Teaching Novices Programming:
A Programming Process Using Goals & Plans with a Visual Programming Environment

Minjie Hu

A thesis submitted for the degree of Doctor of Philosophy at the University of Otago, Dunedin New Zealand

2015
Abstract

It is easy to get novices to understand individual statements of a computer programming language, but it is hard to teach them how to put these statements together into a valid program. This research focuses on the one of the important matters: how to teach novices to construct code. A key constraint is that it aims to develop a new approach for teaching novice programming which is both easy to introduce and effective in improving novices’ learning.

The approach of this study combines three key ideas: using a visual programming environment (VPE); using strategies, specifically the concept of “goal” and “plan”; and having a well-defined programming process. In this study, a visual notation of programming goals, plans and the data-flow relations has been developed and used to represent “hand solution” of programming design. A data-flow framework has also been developed and applied to support implementation of the programming design. A detailed programming process is provided to guide novices programming by using goals and plans in a VPE in order to combine the relevant programming statements into a valid program. Moreover, the data-flow framework provides immediate feedbacks to motivate and engage novices, not only from the unmerged plans, but also from all the rest of intermediate level phases in the programming process till the final program code.

Based on the cognitive load theory, the integration of the above developments has been built up on a visual goal-plan teaching approach. This approach has been evaluated experimentally in a real teaching setting. The evaluation results indicated that the approach has potential to significantly improve the teaching of novices programming.
Acknowledgements

I would like to thank my supervisors, Professor Michael Winikoff, and Professor Stephen Cranefield, for their enthusiasm, advice, support, and patience.

I thank my colleagues and students at Higher Educational Centre, University of Otago, Dunedin, and on Tairawhiti Campus, Eastern Institute of Technology, Gisborne, for their support and encouragement.

I thank my wife, Wen Huang, for overcoming difficulties to look after the family and support my study and research away.
# Table of Contents

**Abstract** ................................................................. i

**Acknowledgements** .............................................. ii

**Table of Contents** .................................................... iii

**List of Tables** ............................................................ vi

**List of Figures** ............................................................ viii

**Glossary** ....................................................................... xi

**Relevant Publications** .................................................... xiv

**Chapter 1  Introduction** ................................................... 1

1.1 Background and Motivation ........................................... 1

1.2 Approach and Research Questions ................................. 2

1.3 Thesis Structure ............................................................ 6

**Chapter 2  Literature Review** ............................................. 9

2.1 Problems and Needs of Teaching Novices Programming .............................................. 9

2.2 A Brief History of Teaching Approaches ................................................................. 17

2.3 Taxonomy of Approaches for Teaching Novices Programming ............................................. 19

2.4 Programming Knowledge ......................................................... 25

2.4.1 Programming Languages and Environments ......................................................... 25

2.4.2 Pragmatic Skills in Programming ................................................................. 34

2.5 Programming Strategies ......................................................... 42

2.5.1 Domain Knowledge ................................................................. 43

2.5.2 Programming by Problem-solving ................................................................. 45

2.5.3 Programming Process ................................................................. 48

2.5.4 Programming Structures, Plans, and Related Tools ......................................................... 52

2.6 Teaching Approaches ............................................................. 65

2.7 Mental Models of Programming ............................................. 69

2.8 Summary ................................................................. 74

**Chapter 3  Overview of the Visual Goal-Plan Approach** ............................................. 80

3.1 Visual Programming Environments (VPEs) ............................................. 81

3.2 The Goal and Plan Approach ......................................................... 82

3.3 The Visual Goal-Plan Approach ......................................................... 83

3.4 The Process of Programming ......................................................... 85

3.5 The Proposed Approach ......................................................... 86
# List of Tables

Table 2-1 Summary of difficulties and proposed solution .............................................. 11
Table 2-2 Comparison of taxonomy in literature based on the revised Bloom’s taxonomy ...... 22
Table 2-3 Comparison of strategies in teaching novices programming ............................. 52
Table 2-4 Summary of the experts’ knowledge chunk approach ..................................... 58
Table 2-5 Key areas of our proposed approach based on experts’ mental representation ...... 61
Table 2-6 Summary of plan tools .................................................................................... 65
Table 3-1 Comparison of related approaches .................................................................. 87
Table 4-1 The Visual Notation of Goals .......................................................................... 100
Table 4-2 The Visual Notation of Plan ........................................................................... 104
Table 4-3 The Comparison of Visual Notation Principles .............................................. 107
Table 4-4 Visual Expressiveness of Diagrams ................................................................. 111
Table 4-5 Graphic Complexity of Diagrams ................................................................. 112
Table 5-1 The Pattern of Process Plan: consuming one dataflow from its in port, but generating only one single datum in dataflow to its out port ....................................................... 131
Table 5-2 The Pattern of Process Plan: consuming dataflow from its in port, and generating a new dataflow to its out port ........................................................................ 135
Table 5-3 The Pattern of Process Plan: consuming multiple dataflow from its in ports, but generating only one single datum to its out port ...................................................... 136
Table 5-4 The Pattern of Process Plan: consuming multiple dataflow from its in ports, and generating a new dataflow to its out port .................................................................. 137
Table 6-1 Janzen and Saiedian’s Evaluation of the TDL Approach ................................. 147
Table 6-2 Test Cases for Example 1 ................................................................................ 151
Table 6-3 Test Cases for Example 2 ................................................................................ 151
Table 6-4 Test Cases for Example 3 ................................................................................ 151
Table 6-5 Desk-check of dataflow in plan ports for Example 1 ...................................... 156
Table 6-6 Summary of Test Schedule for Example 1 .................................................... 157
Table 6-7 Desk-check table of dataflow for computing an average (Example 2) .............. 158
Table 6-8 Desk-check table of dataflow for computing wages (Example 3) ....................... 158
Table 7-1 Summary of CLT teaching approaches in general .............................................. 194
Table 7-2 Comparison of CLT approaches in teaching programming ............................... 205
Table 7-3 Predict Answers of Total Value ........................................................................... 207
Table 7-4 Predict Answers of Average and Name Sub-processes (Sub-goal Labelling) ........ 213
Table 8-1 Course Outline by Years* .................................................................................. 220
Table 8-2 Comparison of possible experimental designs .................................................... 222
Table 8-3 Summary of student results ................................................................................. 230
Table 8-4 The detection of relationship between effect size and sample size for six groups in the ANOVA ............................................................................................................. 231
Table 8-5 Summary of combined student groups and results .............................................. 232
Table 8-6 The detection of relationship between effect size and sample size for three groups in the ANOVA ............................................................................................................. 232
Table 8-7 Summary of hypotheses and prediction of statistical tests ................................. 239
Table 8-8 Post-hoc tests by using M-W U test with Bonferroni-Holm correction ............ 244
Table 8-9 Threats and mitigation to internal validity ............................................................ 246

Table A-1-1 Predicting answers .......................................................................................... 275
Table A-1-2 Desk-checking plan network ......................................................................... 276
Table A-1-3 Test schedule of five phases in the process .................................................... 276

Table A-2-1 Year 2006 Final Examination Marking Schedule .......................................... 278
Table A-2-2 Year 2007 Final Examination Marking Schedule .......................................... 279
Table A-2-3 Year 2008 and 2009 Final Examination Marking Schedule ......................... 280
Table A-2-4 Year 2011 Mid-term Examination Marking Schedule .................................. 281
Table A-2-5 Year 2013 Mid-term Examination Marking Schedule .................................. 282
Table A-2-6 Year 2011 and 2013 Final Examination Marking Schedule .......................... 283
List of Figures

Figure 2-1 Taxonomy used in our literature study of teaching novice programming .......... 23
Figure 2-2 Categorisation of literature in teaching novices programming ...................... 25
Figure 2-3 Schematic of the programming process, adapted from Redmond and Gasen (1989) .................................................................................................................. 52
Figure 2-4 Categorisation of literature in teaching novices programming .................... 76
Figure 3-1 The overview of our visual goal-plan approach .......................................... 87
Figure 4-1 Relationships of plans, modified from Ebrahimi (1992, page 311) (a) Sequential Plans, (b) Nested Plans, (c) Interleaved Plans and (d) Branched Plan ......................... 92
Figure 4-2 Example of merged plans presented in pseudocode .................................. 93
Figure 4-3 Example of plan representation in control flow, modified from Ebrahimi (1992, page 301) ........................................................................................................... 93
Figure 4-4 Example of plan representation in dataflow ................................................. 94
Figure 4-5 Example of branching plan representation in control flow, modified from Ebrahimi (1992, page 301) ..................................................................................... 96
Figure 4-6 Example of rectangle made by three types of goals ..................................... 101
Figure 4-7 Example of three basic goals ....................................................................... 102
Figure 4-8 An example of a goal diagram with multiple process goals ......................... 102
Figure 4-9 An example of goal diagram with multiple input and output goals ............... 102
Figure 4-10 An example of plan network diagram ....................................................... 106
Figure 5-1 Example of plan linkage by plan ports and dataflow ................................ 115
Figure 5-2 Data structure and example of physical model in BYOB for plan linkage .... 116
Figure 5-3 Example of data-flow buffers and plan links ............................................. 119
Figure 5-4 Plan linkage blocks ................................................................................. 120
Figure 5-5 Implementation of sending data to plan out port ....................................... 122
Figure 5-6 Implementation of testing if there is data in a plan in port ......................... 124
Figure 5-7 Implementation of getting a datum from a plan in port .............................. 126
Figure 5-8 Building Input Block ............................................................................. 129
Figure 5-9 Building Output Block .......................................................................................................................... 130

Figure 5-10 Example of *Process Plan* block consuming a dataflow, but generating a single datum ......................................................... 132

Figure 5-11 Similar example of the same pattern for building *Count Plan* and *Maximum Plan* blocks ..................................................................................... 133

Figure 5-12 Another example of the same pattern for building *Search Plan* block ......................................................... 134

Figure 5-13 Example of *Process Plan* block consuming a dataflow, and also generating a dataflow ............................................................................................ 135

Figure 5-14 Example of *Process Plan* block consuming multiple dataflows, but generating a single datum ..................................................................................... 137

Figure 5-15 Example of *Process Plan* block consuming multiple dataflows, and generating a dataflow ..................................................................................... 138

Figure 5-16 Another example of consuming multiple dataflows, and generating a dataflow ......................................................... 139

Figure 5-17 Example of implementation plan diagram by using plan blocks and plan linkage ......................................................... 140

Figure 5-18 Example of dataflow in buffer list for the in port of plan ............................................................................................ 141

Figure 5-19 Example of dataflow transferring between linked plan blocks ............................................................................................ 142

Figure 5-20 Example of feedback from spelling mistake by out port name ............................................................................................ 142

Figure 5-21 Example of feedback from spelling mistake by in port name ............................................................................................ 143

Figure 5-22 Example of plan details working in the Data-flow Framework ............................................................................................ 144

Figure 6-1 Programming process by using goals and plans in our visual programming environment ............................................................................................ 149

Figure 6-2 Three basic goals for *Example 1* ........................................................................................................................................ 152

Figure 6-3 A goal diagram for *Example 2* ........................................................................................................................................ 153

Figure 6-4 A goal diagram for *Example 3* with multiple input-process-output goals ............................................................................................ 154

Figure 6-5 A plan network diagram for *Example 1* ........................................................................................................................................ 155

Figure 6-6 A plan network for *Example 2* ........................................................................................................................................ 155

Figure 6-7 A plan network for *Example 3* ........................................................................................................................................ 156

Figure 6-8 Implementation of plan network diagram by using plan blocks for *Example 1* ........................................................................................................................................ 160

Figure 6-9 Implementation of plan network diagram by using plan blocks for *Example 2* ........................................................................................................................................ 161

Figure 6-10 Using plan blocks from plan library for *Example 3* ........................................................................................................................................ 162
Figure 8-5 Example of P-P and Q-Q plot for the combination of three groups: G1, G2, and G3 .......................................................... 236

Figure B-1-1 Example of predicting required sample size from six groups ................. 284
Figure B-1-2 Example of predicting required sample size from combined three groups .... 285
Figure B-1-3 Example of foreseeing power from current total sample size .................. 285
Figure B-1-4 Example of detecting effect size from samples ........................................ 286
Figure B-1-5 Example of computing the achieved power ............................................. 287

Figure B-2-1 Descriptives for individual sample groups........................................... 289
Figure B-2-2 Descriptives of combination of three groups (Group 1, 2 & 3) .............. 289
Figure B-2-3 Test result of normality of Individual group ......................................... 290
Figure B-2-4 Tests of Normality for Groups in Hypothesis 1 .................................. 290
Figure B-2-5 Tests of Normality for Groups in Hypothesis 2 .................................. 290
Figure B-2-6 Tests of Normality for Groups in Hypothesis 3 .................................. 291
Figure B-2-7 Tests of Normality for Groups in Hypothesis 4 .................................. 291

Figure B-3-1 Results of Hypothesis 1 from Mann-Whitney U test .......................... 292
Figure B-3-2 Results of Hypothesis 2 from Kruskal-Wallis H test ............................ 292
Figure B-3-3 Results of Hypothesis 3 from Mann-Whitney U Test .......................... 293
Figure B-3-4 Results of Hypothesis 4 from Kruskal-Wallis H test ............................ 293
Figure B-3-5 Direct results of post hoc tests for Hypothesis 4 ................................. 294
Figure B-3-6 Pair comparison (G1 vs. G2) of post hoc test for Hypothesis 4 by exact test .... 295
Figure B-3-7 Pair comparisons (G2 vs. G3) of post hoc test for Hypothesis 4 by exact test ..... 295

Figure B-4-1 Results of Hypothesis 4 by one way ANOVA test .................................. 296
Figure B-4-2 Selecting post hoc test from ANOVA .................................................. 297
Figure B-4-3 Post hoc test from ANOVA .............................................................. 297
Glossary

**Cognitive Load Theory (CLT)** — CLT was developed for instructional design in the 1980s based on the experimental observation that the human brain can only handle a limited amount of information at the same time. It mainly concerned instructional design, with the aim of reducing unnecessary cognitive load on the working memory.

- **Extraneous cognitive load** — Extraneous cognitive load is the load that is unnecessary because it does not contribute to schema construction or schema automation.

- **Germane cognitive load** — Germane cognitive load is the load that is directly relevant to the processes of schema construction and schema automation.

- **Intrinsic cognitive load** — Intrinsic cognitive load is the load from some learning material that must be processed simultaneously in the working memory.

**Far Transfer** — Far transfer is the ability to apply knowledge to novel scenarios.

**Goal** — A goal is a certain objective that a program must achieve in order to solve a problem.

- **Sub-goal** — A sub-goal is the purpose of a subsequence of the steps toward achieving a goal.

- **Sub-goal labelling** — This is a teaching method used in the instructional material by requiring learners to give labels for sub-goals to indicate their purpose.

**Learning information elements** — the individual parts of material that must be learned are called elements.

**Long-term memory** — Long-term memory has practically unlimited space to keep large amounts of information, including diverse plans or schemas for solving various problems.

**Near Transfer** — Near transfer is the ability to apply knowledge to a similar scenario.

**Plan** — A plan corresponds to a fragment of code that performs actions to achieve a goal.

- **Plan library** — a collection of plans that can be applied to achieve goals under a certain framework.

**Process** — A process is a set of detailed steps to achieve a solution.
Scaffolding — Scaffolding is support provided at the beginning of a teaching method in order to simplify work on certain tasks. It will be removed for later tasks.

Schema — a schema is a cognitive construct for organising elements of information into a basic unit of knowledge.

  Schema construction — Schema construction is the process of organising information into a unit (i.e. a schema) and transferring it from working memory to long-term memory.

  Schema automation — Schema automation allows schemas to be processed unconsciously rather than in a controlled process, resulting in a reduction of working memory load.

  Schema retrieval — Schema retrieval refers to the retrieving of existing schemas or plans from long-term memory to working memory.

Self-explanation — Self-explanation refers to explanations of the solution that learners give to themselves in order to help learning and understanding.

Worked examples — Worked examples are solutions to demonstrate how to solve problems.

  Fading worked examples — Fading worked examples are a sequence of partially worked examples with decreasing provision of solution steps, which are used to reduce cognitive load by isolating effort within a few limited steps each time.

Working memory — Working memory is the place where all the conscious cognitive processing occurs, such as thinking, reasoning and judging. It has very limited capacity and thus is easily overloaded.
Relevant Publications

Some ideas, figures, and tables have been published in the following papers:

**Full Conference Papers:**
2. Hu, M., Winikoff, M. and Cranefield, S. (2012) Teaching Novice Programming Using Goals and Plans in a Visual Notation, *Proceedings of the Fourteenth Australasian Computing Education Conference ACE2012* (Melbourne, Australia, 30 Jan.- 3 Feb. 2012), CRPIT 123, pp 43-52, Australian Computer Society. This paper introduces our approach for teaching novice programming, which is both easy to introduce and effective in improving novice learning. Our approach combines three key ideas: using a visual programming language; using strategies, specifically using the concepts of “goal” and “plan”; and having a well-defined process. This approach had been evaluated experimentally and the results indicated its potential to significantly improve teaching programming to novices. It covers part of contents, figures, and tables in Chapter 2, Chapter 4, Chapter 5, Chapter 6, and Chapter 8.

**Doctoral Consortium:**
1. Hu, M., Winikoff, M. and Cranefield, S. (2014) Evaluating a New Approach for Teaching Novice Programming: Issues and Lessons, *Proceedings of the Fifth New Zealand Information Systems Doctoral Consortium (NZISDC)*, 25th-26th July, 2014, pp 29-34, Otago University. After developing a new approach for teaching novices how to program, we face the challenge of evaluating the effectiveness of this new proposed approach. Issues include whether the actual sample size is enough; how we choose appropriate statistical tests; and what we need to measure for the strength of
evidence against the null hypothesis. This paper covers part of contents, figures, and tables in Chapter 8.

Chapter 1 Introduction

1.1 Background and Motivation

It has been continually reported that there is a significantly high rate of failing or withdrawing from the first programming course (Ebrahimi & Schweikert, 2006; Facey-Shaw & Golding, 2005; Lahtinen, Ala-Mutka, & Jarvinen, 2005; Sykes, 2007).

There seems to be a broad consensus (de Raadt, 2008; du Boulay, 1986; Robins, Rountree, & Rountree, 2003; Soloway, 1986; Winslow, 1996) that the place where novices struggle is in constructing code to solve a problem, even a very simple problem:

“[An important point] is the large number of studies concluding that novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs. Even when they know how to solve the problems by hand, they have trouble translating the hand solution into an equivalent computer program.” (Winslow, 1996, p. 17; quoted in Robins et al. 2003)

A wide range of approaches have been proposed to improve learning of programming by novices, which include collaborative learning (Rößling et al., 2008), progressive teaching (Djordjevic, 2007; White, 2003; Whittington, 2005), explicitly teaching problem-solving (de Raadt, 2008; de Raadt, Watson, & Toleman, 2006; Pears et al., 2007; Rist, 1995), roles of variables (Sajaniemi, 2002; Stützle & Sajaniemi, 2005), programming visualisation (Ben-Ari, 2001; Crews & Murphy, 2003; Hu, 2004; Naps et al., 2003; Naps et al., 2002; Rößling & Freisleben, 2001), psychological analysis and mental models (Ma et al., 2008; Mayer, 1981; Winslow, 1996; Yehezkel, Ben-Ari, & Dreyfus, 2004), visual programming environment (Guzdial, 2004; Kelleher & Pausch, 2005; Maloney et al., 2008) and game programming (Feldgen & Clúa, 2004; Guzdial & Soloway, 2002; Hu, 2008; Kölling & Henriksen, 2005).

However, the above problem of how to combine diverse programming statements into a valid program is still unsolved in the teaching of novices programming. Therefore, our proposed research focuses on the one of the important matters: **how to teach novices to construct code to solve a problem.** A key constraint is that we aim to develop a new approach for
teaching novice programming which is both easy to introduce and effective in improving novice learning.

1.2 Approach and Research Questions

There are a number of skills and knowledge areas that need to be delivered in order to learn programming. They include:

- an understanding of the notion of a conceptual machine;
- an understanding of the idea of programming such a machine;
- familiarity with the syntax and semantics of a programming language;
- familiarity with the tools and environment (e.g. editor, interpreter or compiler, debugger);
- an understanding of the elements of a development process (e.g. clarify requirements, design, implement, test and debug ... iterate);
- knowledge of a range of programming “patterns” e.g. iteration over an array, use of temporary variables to store flag values etc., and
- debugging skills.

On the other hand, there are a number of reasons that have been proposed for the failure of learning to program, such as:

- “fragile” knowledge of programming concepts (de Raadt, 2008; Lister et al., 2004; McCracken et al., 2001; Pea, Soloway, & Spohrer, 1987; Perkins & Martin, 1986);
- lack of problem-solving strategies and plans (Winslow, 1996; Robins et al. 2003; de Raadt, 2008), and
- lack of detailed mental model (Winslow, 1996; du Boulay, 1986; Robins et al. 2003).

Our proposed approach for teaching novice programming focuses on the heart of the matter: constructing code. We emphasise strategies in problem solving (using goals and plans — discussed below), use a visual programming environment, and develop a pedagogical approach. Specifically, we develop a detailed process for programming, which is inspired by the “Programming by Numbers” (Glaser, Hartel, & Garratt, 2000) approach and the “TeachScheme” (Felleisen et al., 2003) approach.
Recently, researchers have drawn attention to visual programming environments (VPEs) (Kelleher & Pausch 2005), which aim to provide an attractive, easy, and fun way for novices to learn programming using VPEs such as Alice 3 (Dann & Cooper, 2009), Scratch (Resnick, M. et al. 2009), Scratch for Second Life (S4SL) (Harrell, et al. 2008), and Puck (Kohl, 2007). These VPEs are similar in that they use pre-built blocks in which a program is built up by dragging-and-dropping different blocks. The visualisation of debugging in the control-flow visual programming language (VPL) provides static and dynamic visualisation for both variables and control-flow inside the pattern. It provides immediate feedback of design results in general and demonstrates the relationship between variables and control command in detail. At the same time, the ease of using control-flow VPL will have engaged novices in learning and will also build up novices’ confidence from starting, through debugging, to successful execution. Although using VPEs can prevent syntax errors, researchers (Klassen, 2006; Sykes, 2007) have found that the approach is not effective. After using Alice for three semesters, Klassen (2006) found that using Alice did not serve the goal of providing a solid programming concept to students.

“Programming by Numbers” breaks the programming process into a series of well-defined steps and gives students a way of “Programming in the Small” to create the smallest components of functions. “TeachScheme” similarly uses a systematic design method to produce well-specified intermediate products in a stepwise fashion, although its method is more detailed than the “Programming by Numbers” method. Both methods are data-driven, and more suited to functional programming languages. Our detailed process for programming covers the whole programming process, starting from how to analyse the problem, and how to support design activities, including representing and mapping plans (discussed below), visualising and evaluating plans, and merging plans. Finally, it provides steps for coding and debugging to improve the accuracy of program design.

For example, consider the following programming problem:

Write a program that will read in integers and output their average. Stop reading when the value 9999 is input. (Soloway, 1986; de Raadt, 2008)

Our approach provides steps for the novice to identify variables from the problem domain in terms of name, type, and function. Then we introduce steps which guide the novice programmers in how to analyse variables for input integers, calculate the average, and output
the correct result. Next we provide more details on how to present these variables and how to process them by control-flow patterns in the design activities.

As suggested by the list of skills and knowledge areas needed, in addition to defining a detailed process for programming, we also need to define a notation. A key aspect of our proposed approach is that this notation will include direct support for representing goals and plans.

A goal is a certain objective that a program must achieve in order to solve a problem, while a plan (Spohrer, Soloway, & Pope, 1985) corresponds to a fragment of code that performs actions to achieve the goal. Goals and plans (Soloway, 1986) are two key components in representing problems and solutions, which are stereotypical and canned answers. Soloway (1986) also proposed to use a goal/plan “language” to explicitly let a novice construct his/her own plans from a library of plans.

During the last three decades, the programming practice has changed dramatically (see Chapter 2). The focus of research in teaching novices programming has also changed from writing programs and solving problems (McCraeke et al., 2001), to reading, tracing, and explaining code (Lister et al. 2004; Whalley et al. 2006). Subsequently, learning difficulties were considered in terms of programming language, programming strategy, and metal model (Ben-Ari, 2001; du Boulay, 1986; Ko et al., 2004; Perkins et al., 1988; Lister et al., 2004; Robins et al., 2003).

Specifically, goals and plans are based on the top-down and stepwise refinement methods, which had been argued as not being suitable for novices in the 1980s and 1990s (Pattis, 1993; Ulloa, 1980). In order to overcome this issue, Caspersen and Kölling (2009) and Caspersen (2007) advocated using an approach with stepwise improvement and a guided process. Moreover, goals and plans had been explicitly used as strategies in teaching novices’ programming, which include the description of plans together with examples or diagrams (De Raadt, Watson, and Toleman (2009).

As an alternative, we propose using a visual programming language (VPL) to explicitly and directly represent plans and goals in order to reduce teaching time by avoiding syntax errors and reducing the need to learn syntax and then moving on to use of a traditional control-flow VPL. For example, in the programming problem given above, the goal/plan might be
repeatedly inputting data, checking to guard against division by zero, and outputting the correct result. Furthermore, we propose to visualise the VPL’s execution.

It is a difficult activity for a novice to effectively merge the plans that were used to solve to problem. Hence, we aim to systematically introduce goals and plans into the VPL environment, provide support to represent goals in a VPE, map the plans into the control-flow VPL by using patterns, assist the detection of variation from using a pattern, and provide guidance of merging plans to achieve the goals.

Although several researchers have tried to solve the problem of combining different statements into a valid program, the answers are still considered to be “difficult” (Robins et al., 2003; Soloway, 1986; Winslow, 1996). We therefore use a carefully chosen combination of approaches (goals and plans, VPE, and process). Our solution is promising, but it still has issues (e.g. relies on a plan library, which is providing considerable scaffolding). The research questions below are specific questions to be addressed as part of developing a notation and process.

**RQ1:** How can goal and plan representations be integrated into a VPL?

We consider using experts’ knowledge and strategies in terms of goals and plans to solve programming problems, particularly using the dataflow relations proposed by Pennington (1987). We also consider using visual notation to represent goal-plan and dataflow as a “hand solution”, i.e. programming analysis and design for novices.

**RQ2:** How can guidance be provided to novice programmers in using goals and plans and merging plans?

A solution often needs multiple plans and these plans need to be merged into one program. Based on the existing strategies (Soloway, 1986) for gluing plans, our research explores how to effectively merge our plans from data-flow design into control flow code in the VPL environment. Therefore, both a set of steps and a set of principles are applied to provide a guidance to support merging plans.

**RQ3:** How can the teaching process for programming be improved and evaluated?
After finding solutions to the previous two research questions, we will integrate these solutions into a new teaching process. Before we apply the new teaching process in educational practice, we need to assess it against the cognitive load theory (CLT).

Ideally, the evaluation of this new teaching process should be done in a real teaching setting, rather than in a short workshop. However, we need to consider the class size and particularly rule out influences by any plausible factors such as different student cohorts, exam questions, computer languages, and marking standards. We also consider using an appropriate statistical method to analyse the teaching results.

1.3 Thesis Structure

Chapter 2 starts with exploring the problems of teaching novices programming and surveys the history of teaching approaches. It then presents a taxonomy of teaching approaches for our literature study. Following the taxonomy, it explores the essential programming knowledge for novice programmers, comprising programming languages and environments, as well as pragmatic skills in programming. We then explore problem-solving strategies in programming. We conclude that a programming process needs to provide detailed guidance to support novices programming. We also consider experts’ knowledge, strategies, and tools for novices’ programming. Finally, we study the programming mental model to support and guide novices learning.

Chapter 3 overviews our proposed approach which integrates three approaches: 1) using visual programming environments (VPEs); 2) taking advantage of experts’ programming strategies for using goals and plans (Soloway, 1986); and 3) providing a programming process. The details of our implementation of goals and plans in a VPE using a programming process for the development of our approach are presented in Chapter 4, 5, 6, and 7.

Chapter 4 defines the visual notation of goals and plans in the data-flow paradigm. It considers that dataflow is not only a good starting point for programming design for novices (Good, 1999a, 1999b), but is also an appropriate paradigm that enables the plans to be executable before the process of merging plans. Our visual notations for goals and plans, using the dataflow paradigm, were designed and evaluated based on the theory and principles of visual notation (Moody, 2009). By using the visual notation, one can build up a goal diagram to
represent the problem analysis. The goal diagram is then mapped into a plan network that can be implemented subsequently.

Chapter 5 describes the development of a data-flow framework to support the visual plan notation proposed in Chapter 4. The framework was developed by using a visual program language: BYOB. The framework provides a plan library for novices to use. It also provides patterns of plan structure for novices to build their own plans into this library. In order to allow the unmerged plans to be executable in the data-flow paradigm, the framework provides linkage blocks to connect plans by linking plan outputs to plan inputs through data-flow buffers.

Chapter 6 introduces a detailed step-by-step programming process for novices to use goals and plans within the data-flow framework developed in Chapter 5. The detailed programming process uses a “test-early” approach and supports goals, plans and a visual programming language. The process includes six phases. The intermediate-level products in each phase of the process except the first phase are executable and testable, which provides immediate feedback to novices and engages them in the process.

Chapter 7 discusses cognitive load theory (CLT) in general teaching and in teaching programming based on our previous development in Chapters 4, 5, and 6. According to CLT, there are a number of different types of cognitive load in learning programming. The working memory of novice programmers can be easily overloaded. We assess the development in Chapter 4, 5, and 6 by using CLT and integrate them into an approach for teaching novices programming with goals and plans.

In Chapter 8, we evaluate our new approach by collecting final exam results from a real teaching practice in 2011 and 2013. Due to the class sizes being small, the evaluation had to compare results from the group taught using the new approach against those taught using the old approach in previous years (from 2006 to 2009). In the comparison, there are other differences such as different student cohorts, exam questions, computer languages, and marking standards. Therefore, a number of hypotheses were defined to correspond to different plausible explanations for variations in student performance due to these confounding factors. Having small sample sizes, the evaluation also considered relevant statistical issues such as selection of an appropriate statistical method, as well as the values of
p-value, sample size, effect size, and statistical power. The results suggest that the goal of this research to provide guidance for novices to use goals and plans in a VPE has been achieved.

Chapter 9 concludes this study, clarifies the contributions to the field of teaching novices programming, and proposes future work.
Chapter 2 Literature Review

In Chapter 1, we noted that novices struggle when learning computer programming. In this chapter, we explore the literature to identify where the problems are for novices and what differences exist between novices and experts. In particular, we establish a taxonomy of research literature on teaching novices programming. Our literature study then focuses on programming knowledge, strategy, and mental models. Finally, through the comparison of these three areas, we identify the gaps that lead to our further study.

2.1 Problems and Needs of Teaching Novices Programming

In the last three decades, many issues and difficulties in teaching and learning programming were identified and analysed for the purpose of improvement by educators (Du Boulay, 1986; Lahtinen, Ala-Mutka, & Järvinen, 2005; Milne & Rowe, 2002; Pane & Myers, 1996). One of the issues that novices face is the poor design of computer languages and of programming environments for program development (Pane & Myer, 1996). For example, although visual programming languages were expected to be superior to textual languages for novice programming, most visual languages have high “viscosity” so that they require more effort than textual languages to make changes to a program. Some textual languages have violations of consistency in notation, e.g. “The keyword static in C++ has many different meanings depending on context” (Pane & Myer, 1996, p51).

Novices are beginners of learning computer programming. Shneiderman (1976) classified novices as students who are “currently enrolled in a first course in programming” (p124) rather than naive, intermediate, or advanced ones. The problems of novice programmers were summarised by Winslow (1996) as being: 1) novices lack an adequate mental model; 2) novices have fragile knowledge (surface knowledge, rather than strategies); 3) novices use general problem-solving strategies (i.e. work from goal to solution) rather than strategies on particular problem; and 4) novices use control structures, line-by-line and bottom-up approaches in programming. A study claimed that these features were due to the conditions of novices’ learning (Perkins et al. 1989). It indicated that altering novices’ learning condition was promising for improving their performance.
In order to reduce the burden of memorising, the human intellect groups chunks of information without conscious effort. The chunks are believed to be easily handled as units, which are widely possessed by experts in areas such as chess-playing, physics, and computer programming (Chase & Simon, 1973; Larkin et al., 1980; Shneiderman, 1973; Wiser & Shertz, 1983; McKeithen et al., 1981; Bateson, Alexander & Murphy 1987). Winslow (1996) concluded that experts have many mental models, a deep knowledge of their subjects, better problem-solving techniques, better syntactical and semantical knowledge and better tactical and strategic skills.

The differences between novices and experts are generally explained (McKeithen et al., 1981) in the following way as “experts have not only more information, they have that information better organised into useful chunks. Instead of perceiving and remembering individual pieces of information, they process meaningful groups of information, making their perception more efficient and their recall performance much higher” (p307). Adelson (1984) suggested that experts use abstract representation and general information regarding what a program does whereas novices apply concrete representations and detailed information about the program.

Based on research on cognitive processing in programming, Bateson et al. (1987) summarised the differences between novices and experts as: 1) experts organise chunks hierarchically based on a large number of procedural patterns for better performance in quick and accurate recall; 2) experts use a high-level knowledge (e.g. patterns) rather than specific statements to understand a program; 3) experts not only use knowledge of syntax and semantics, but also apply high-level programming plan knowledge in writing programs. The programming plan knowledge proposed by Soloway and his colleagues (Soloway, 1986; Soloway & Ehrlich, 1984; Spohrer, Soloway, & Pope, 1985) is discussed in Section 2.5.4.

The areas of learning difficulty proposed by du Boulay (1986) were widely recognised in the community of computer science education (Ben-Ari, 2001; Lister et al., 2004; Ma et al., 2011; Robins et al., 2003). du Boulay claimed there were five areas of difficulty: 1) problem orientation, i.e. understanding the problem to solve; 2) notional machine, a conceptual model of a computer system by which novices understand the execution of the program; 3) understanding the notation of various formal languages including both syntax and semantics; 4) programming patterns (c.f. plans); and 5) the pragmatic skills of programming (including the lifecycle, testing, debugging, and tool use).
du Boulay also claimed three orthogonal types of mistakes: 1) overuse of an analogy (e.g. a variable is like a box, therefore it can hold multiple values); 2) overgeneralising a rule or principle; and 3) poor handling of complexity and of interactions. He advocated introducing a notional machine as a mechanism for running a program and viewing a program as a mechanism, not just a collection of pieces for a mechanical construction (e.g. cogs, wheels, etc.).

With regard to the examination of difficulties of novices learning to program by du Boulay (1986), we briefly summarise several proposed solutions to improve the teaching of novices programming into three categories, comprising 1) programming knowledge, 2) programming strategy, and 3) mental models (see Table 2-1).

<table>
<thead>
<tr>
<th>Categories of Improvement</th>
<th>Difficulties Identified by du Boulay</th>
<th>Proposed Solutions in Literature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Programming knowledge</td>
<td>• syntax and semantics;</td>
<td>• Overcome barriers of understanding syntactic representation (Ko et al., 2004);</td>
</tr>
<tr>
<td></td>
<td>• Pragmatics</td>
<td>• Feedback (Perkins et al., 1988);</td>
</tr>
<tr>
<td>2. Programming strategy</td>
<td>• Problem orientation,</td>
<td>• Emphasis on problem-solving strategies (Perkins et al., 1988);</td>
</tr>
<tr>
<td></td>
<td>• Acquiring standard structures</td>
<td>• Consider chunks of code (Perkins et al., 1988);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Reveal programming rules (Ko et al., 2004);</td>
</tr>
<tr>
<td>3. Mental model</td>
<td>• Notional machine</td>
<td>• Support learning, and motivation (Ko et al., 2004; Perkins et al., 1988).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Tracing mental representation, using external representation, and understanding programming concepts (Vainio &amp; Sajaniemi, 2007).</td>
</tr>
</tbody>
</table>

Ko et al. (2004) observed novices’ learning of programming using Visual Basic (VB). They identified six barriers in terms of design (i.e. what do I want computer to do), selection (i.e. I don’t know what to use), coordination (i.e. I don’t know how to make them work together), use (i.e. I don’t know how to use it), understanding (i.e. it didn’t do what I expected), and information (i.e. I don’t know how to check). They proposed to provide scaffolding to support the design because of its difficulty. They suggested revealing implicit rules of programming, e.g. strategies for organising different parts of code into a valid program. They advocated overcoming the barriers of understanding syntactic representations which is proposed to make the learning easy.
Vainio and Sajaniemi (2007) proposed that program tracing is an important skill in design, program comprehension, program writing, debugging etc. They argued that novices had ineffective tracing skills. In order to explore the reason for this issue, they interviewed 32 students in a summer course. They claimed that novices had difficulties in tracing (or mental execution) of program mental representation. Particularly, novices had difficulties in coping with tracing values for more than one variable. Novices also were confused by program function (i.e. its purpose and goal) and program structure (i.e. the way of arrangement or construction of program code). Moreover, novices lacked ability to use external representations (e.g. a diagram) and to raise the abstraction level (e.g. tracing through a data-flow representation). They argued that the failure of mental tracing was not only due to novices’ lack of experience, but also because of their cognitive limitations. They suggested that it is important to understand the semantics of program structure to overcome difficulties from confusing about both function and structure. Furthermore, they proposed that novices had difficulties with external representations such as a diagram of a data structure, i.e. being neither able to present program state, nor to trace programs by using diagrams. They argued that “The use of external representations requires not only cognitive capacity (to produce the representation and to compare it with the program text) but also a detailed and non-fragile understanding of the elements to be externalized” (Vainio & Sajaniemi, 2007, p239).

Therefore, they proposed that:

“it may not be helpful to present diagrams to students unless they first understand the concepts that are depicted in the diagrams. Likewise, it may not be helpful to teach students to draw diagrams themselves unless that instruction also teaches students to form a mapping between diagrams and program code” (Vainio & Sajaniemi, 2007, p240).

They advocated that knowledge of dataflow would be helpful in tracing of variables. Finally, they concluded that the tracing difficulties were mainly due to novices’ lack of experience and conceptual understanding of programming.

The research methods regarding novice programming have been developed from traditional experience and surveys to directly collecting novice programming code data online. Brown et al. (2014) had taken the advantage of the BlueJ programming environment by collecting one hundred thousand users’ Java source code from around the world through the Internet. However, they found it is difficult to identify whether or not the code was written by novices or experts. In the continuing study of these collected programmes, Brown and Altadmri (2014)
Informatively categorised the mistakes in the code into eighteen categories, including 11 categories of misunderstanding or forgetting syntax (e.g. mismatched parentheses, spurious semi-colon, etc.), two categories of errors (e.g. mismatching data types in arguments, parameters, and method results), and five other categories of semantic errors (e.g. missing methods and method return statements). They found that many most frequent mistakes were not syntactic except for mismatched parentheses. They challenged that educators may overrate the frequency of different syntactic and semantic mistakes compared to their findings.

Novices were classified into two groups by Perkins et al. (1986): “stoppers” (who are unwilling to explore and stop) and “movers” (who continually try to explore). The major reason for such a classification was that novices have a crucial issue of lacking instructions about what to do next. Sometimes movers can be successful. In other times, they may go around in circles retrying the same idea. Perkins et al. identified that without detailed instructions novices attempt to solve a programming problem by “tinkering” (writing some code and then making small changes). They argued that although “tinkering” promoted “movers” rather than “stoppers”, it often makes their problem worse without sufficiently tracking their program (understanding precisely what the program does).

Furthermore, the reason why novices have difficulties in programming has been suggested to be due to novices’ fragile knowledge rather than missing (forgotten) knowledge (Perkins, Schwartz, & Simmons, 1988). “Missing knowledge alone urges more review, perhaps with drill and practice to consolidate the knowledge base. In the case of fragile knowledge, the students have knowledge to work with and build upon” (Perkins et al. 1988, p310).

Therefore, several instructional strategies were proposed for the purpose of strengthening the improvement of novices’ fragile knowledge. Perkins et al. emphasised stronger problem-solving strategies. They also suggested enhancing the engagement and providing support with feedback. “Interestingly, stoppers were not necessarily especially inept nor movers especially able. With some encouragement, a student who seemed to be a stopper often could solve the problem” (Perkins et al. 1988, p312).

Moreover, Perkins et al. (1988) proposed a “metacourse” or a bridge course for enhancing the learning of programming. The bridge course was designed as a supplement to a normal course
mainly by introducing strategies and mental models. The course promoted mental models by using a visual model in the form of a “paper computer” (using separated papers to show the mental execution in terms of code, variables, and results). It stressed the importance of a “recurrent schema” (or pattern or programming plan) for comprehension and generation of programs. It also extended the problem-solving strategies and mental models for a programming development process to include planning, writing, and testing. Finally, Perkins et al. advocated that “at each level of the planning and coding stages students are encouraged to consider the purpose of chunks of code and the action those chunks actually effect” (p327).

From previous study, we conclude that the theme of all proposed solutions is highly relevant to overcoming novices’ cognitive limitations, i.e. reducing novices’ cognitive load, which is the usage of working memory when solving a problem (cf. Chapter 7). However, individual studies varied from focusing on progressive problem-solving, support learning, motivation and feedback (Perkins et al., 1988), to using scaffolding, teaching beyond textual program code representation, considering chunks of code or patterns, and revealing programming rules (Ko et al., 2004; Perkins et al., 1988).

Over the recent decade, after identifying poor performance in writing program by the majority of students from several universities (McCracken et al. 2001), researchers turned to exploring student performance in tracing program code (Lister 2004) and explaining the purpose of a piece of code (Whalley et al., 2006). Lister et al. (2004) assumed that program reading-related knowledge and skills were precursors to problem solving. Instead of directly assessing students’ abilities to write programs, they evaluated their skills in reading and tracing (manual execution or “desk-checking”), including predicting the possible results from a piece of code and completing a skeleton of code by choosing the correct answer from a small list of possible options. Based on the results of twelve multiple choice questions from 556 students in seven countries, they found that many students were weak in both reading and tracing. Therefore, they concluded that “any research project that aims to study problem-solving skills in novice programmers must include a mechanism to screen for subjects weak in precursor, reading-related skills” (p139).

Whalley et al. (2006) carried further investigation of reading and comprehension skills in novice programmers using the revised version of Bloom’s taxonomy (Anderson et al., 2001) for the multiple choice questions and SOLO (the Structure of the Observed Learning Outcome)
taxonomy (Biggs & Collis, 1982) for the program purpose explanation questions. There were five categories in the SOLO taxonomy: 1) prestructural (unorganised answers that miss the point), 2) unistructural (only one aspect or point), 3) multistructural (multiple unrelated points), 4) relational (logical related answer), and 5) extended abstract (unanticipated extension and generalisation). Considering the minimal chance of getting an “extended abstract” response based on their provided questions, Whalley et al. excluded this category and added a “Blank” category for the question being not answered. Based on 117 respondents to multiple choice questions analysed by the Bloom categorisation, they found that “the level of difficulty of programming assessments at the introductory levels, whether or not inherent in the subject itself, presents a significant and possibly unfair barrier to student success” (Whalley et al. 2006, p.250). From 69 students on the SOLO categorisation, they also found that “weaker students [were] less likely to show performance at higher levels of the taxonomy, and stronger students tending to show higher level capabilities” (p.251). They concluded that “Students who cannot read a short piece of code and describe it in relational terms are not well equipped intellectually to write code of their own” (p.251). Furthermore, regarding the relationship between reading, tracing, and writing skills in novice programming, a recent study found that students’ performance on both tracing and explaining programming code was correlated with their performance in writing the code (Lopez et al., 2008).

The studies on transitioning novices from “stoppers” to “movers” indicate that changing conditions of learning could improve novices’ performance. For example, the report by McCracken et al. (2001) was commonly cited as an example of novices’ failure in writing programs. Ten years after McCracken’s work, McCartney et al. (2013) reran the experiment and found more promising results. In rerunning the experiment, McCartney et al. made a number of changes based on two reasons. Firstly, the original questions in the assessment were considered too hard for novices (Lister, 2011d; McCartney et al., 2013). Next, novice’s cognitive load was not reduced in the original experiment (McCartney et al., 2013). Therefore, McCartney et al. adjusted the conditions of McCracken’s experience by revisiting the previous study. Specifically, they used an Integrated Development Environment (IDE), to provide partially completed programs (e.g. with completed I/O code), and to evaluate students in an “open-book” environment allowing access to online documentation. They concluded that the experiment indicated that the significant improvement results came from the significance of reducing students’ cognitive load.
However, a recent study (Watson & Li, 2014) indicated that the failure rate of novice programming has seen no significant change in the last three decades. Watson and Li (2014) also suggested that the pass rate is unaffected by the programming language over the time, but may be positively affected by teaching smaller groups of students and changing the classroom environment (e.g. teaching in a computer lab).

In order to gain further evidence for the above claims, Vihavainen, Airaksinen, and Watson (2014) quantitatively compared the pass rates from thirteen approaches in 32 articles published from 1980 to 2014. They classified these approaches into five social, engagement, and motivation related teaching interventions: 1) collaboration and peer support, 2) bootstrapping (e.g. using bridge course of CS0, visual programming environment, etc.), 3) relatable content and contextualisation (e.g. media computation\(^1\), games, and real world projects), 4) course setup, assessment, and resources (e.g. adjusting course contents, providing support, changing grading schemas, etc.), and 5) hybrid approaches (e.g. media computation with pair programming and collaboration with games). They concluded that marginal differences existed between the effectiveness of teaching interventions although no statistically significant differences were founded between them. They advocated that both media computation and pair programming are the best practices while using a game theme is the least effective. Furthermore, Porter et al. (2013) and Porter and Simon (2013) proposed that the trio of pair programming, media computation, and peer instruction as a promising approach for increasing social need and support.

Although it is impossible for novices to become experts from only understanding the first programming course, our goal is to make the most use of the first programming course and to fill as many of the gaps in knowledge and skills between novices and experts. Before we consider how to develop our approach to achieve this goal, we explore a brief history of teaching approaches for novice programming, and then establish a taxonomy to further structure our literature study.

---

\(^1\) Media Computation is an approach to teach computing using fundamental manipulations of digital media such as images, music, digital video, and etc. See [http://coweb.cc.gatech.edu/mediaComp-teach](http://coweb.cc.gatech.edu/mediaComp-teach)
2.2 A Brief History of Teaching Approaches

The well-known computer programming paradigms include imperative programming, object-oriented programming, functional programming, logic programming, and concurrent programming (Van Roy & Haridi, 2003). A key question is which programming paradigm should be introduced to novices first as appropriate programming teaching paradigm, such as procedural-first, objects-first, functional-first, algorithms-first, variables-first (Cooper, Dann, & Pausch, 2003; Sajaniemi & Hu, 2006). Since programming teaching paradigm is changed dynamically by educators, it has resulted in much debate.

Since its introduction in the 1970s, the “procedures early” (top-down design and structured programming) teaching approach had been used to teach programming with structured programming languages and it dominated teaching programming until “object-oriented” programming started taking over in the first programming course in the 1990s. After shifting teaching paradigm to objects-first, arguments were raised amongst educators both for and against objects-first (Lister et al., 2006). Although many educators insisted on sticking with objects-first with further refinements (Schulte & Bennedsen, 2006), some of them moved back to the procedural paradigm (Vilner, Zur, & Gal-Ezer, 2007). Others attempted to explore new approaches going beyond objects-first by introducing roles of variables (Sajaniemi & Hu, 2006).

The ACM Computer Science Curriculum (ACM & IEEE, 2013) stated that introductory programming courses were still “paradigm-based” such as procedures-first, objects-first, and functional-first and the debate on their relative merits had been continued since 2001. The ACM curriculum emphasised that introductory courses are programming-focused and are also divided into writing whole programs and completing or modifying existing programs and skeletons.

In the “procedure early” approach, based on the ideas of structured programming (Pattis, 1993), subprograms are introduced after basic control structures (sequence, selection and loop), but before data structures. In structured programming, the main ideas are: 1) designing the problem solution by using procedures in a top-down direction; and 2) coding procedures as a general module block that is connected to outside code by its parameters and result.

In 1990s, when C++ and the “hot” Java language became popular, the object-oriented (OO)
programming approach was shifted from an advanced subject to the first programming course, a paradigm named “objects-early”. Early studies found some advantages in teaching using this approach, but no significant improvement overall in performance. The main reason argued for teaching objects-early is because students find learning OO programming to be very difficult after learning the procedural (or imperative) approach (Kölling, 1999a; Lister et al., 2006). Hence, “if we want to teach object-orientation, we should do it first” (Kölling, 1999a, p1). However, computer science educators argued that “OO programming is more difficult to learn than imperative” (Lister et al., 2006, p148). Moreover, “students who learn objects-first do not learn algorithmic problem-solving” (Lister et al. 2006, p149).

An early study comparing the comprehension of OO and procedural teaching methods (Wiedenbeck et al., 1999) suggested that novices from OO courses understand the function or purpose of a program better than those from procedural courses, while novices from procedural courses understand procedural contents better than those from OO courses. Further comparison (Vilner, Zur & Gal-Ezer, 2007) with 94 students in procedural paradigm course and 76 students in OO course also suggested that students from the OO course did better than those from procedural course in using top-down design for a larger problem.

However, Sajaniemi and Hu (2006) criticised the high drop-out rates and poor skills in the first programming course, and argued that these were due to the object-first approach. They argued that the object-first approach is too sophisticated for novices to build up a valid mental model without first understanding the basic building blocks of programs. In order to emphasise a new notion of “roles of variables” as a transition to objects, they combined procedures-early and objects-first approaches by introducing variables first and then objects. They claimed that the roles of variables captures experts’ tacit programming knowledge. For example, a role of “stepper” is the notion of counting items and a role of “gatherer” is the notion of accumulation. In the variable-first part, the basic control structures were also introduced. They suggested another new notion, “compounds” (methods and procedures in procedural programming or constructors and methods in OO), which was not used for structured programming, but mainly for code re-use such as constructors and methods in OO programming.

A recent survey reported that most institutes still taught procedural programming in the introductory course in Australasia from 2001 to 2010 (Mason, Cooper, & de Raadt, 2012).
Some institutes that taught object-oriented programming in 2001 had moved by 2010 to teach a mix of paradigms by starting with procedural and then moving to object-oriented programming. The use of the functional paradigm was reported to have nearly disappeared in introductory programming courses.

In short, “whilst there is no silver bullet, no teaching approach works significantly better than others, a conscious change almost always results in an improvement in pass rates over the existing situation” (Vihavainen, Airaksinen, & Watson, 2014, p26). Next, we proceed to develop a taxonomy of research in order to better understand and compare prior research.

2.3 Taxonomy of Approaches for Teaching Novices Programming

Before we explore further, we consider the taxonomy of approaches used for teaching novices programming, including both pedagogical approaches and programming approaches, in order to organise the research literature for our study. That is to say our taxonomy is as one about the knowledge and activities needed for programming. A taxonomy is a classification system to arrange different approaches in a certain way. Since there is no unified classification of approaches for teaching novices programming, we study existing taxonomies and build one for our current study.

The most popular taxonomy in education, particularly for curriculum and assessment, is Bloom’s taxonomy (Bloom & Krathwohl, 1956). It has six categories from the lower knowledge to abilities and the higher skills through the cognitive domain, consisting of 1) knowledge, 2) comprehension, 3) application, 4) analysis, 5) synthesis, and 6) evaluation. Bloom’s taxonomy has been applied in computer science course design (Scott, 2003) and the development of ways of assessing the ability of a novice programmer to comprehend programs (Lister, 2000; Lister & Leaney, 2003).

Further studies claimed that the revised Bloom’s taxonomy (Anderson et al., 2001) is better than the original for describing novices’ performance in programming (Fuller et al., 2007; Thompson et al., 2008). The revised Bloom taxonomy (Anderson et al., 2001) provides a two dimensional matrix. One is the cognitive process dimension, consisting of: 1) remember (recalling knowledge or concepts), 2) understand (constructing meaning from external messages), 3) apply (using knowledge and concepts to solve problems), 4) analyse
(decomposing the problem into components to discover the overall structure and relationships), 5) evaluate (making judgments using standards), and 6) create (make a new structure from components). Another is the knowledge dimension, comprising: 1) factual knowledge (or declarative knowledge, basic elements of a subject, e.g. symbols, syntax), 2) conceptual knowledge (principles and theory of a subject, e.g. semantics, rules, concepts), 3) procedural knowledge (knowing how to do a task, e.g. performing skills, algorithms, process), and 4) metacognitive knowledge (high level strategy or cognitive process of learning and thinking).

In the knowledge dimension, a broad range of knowledge is required in learning programming. However, only four categories of relevant programming knowledge were proposed by Brooks (1990). These four categories of knowledge are: 1) application domain knowledge (understanding the properties of applications, such as mathematics, physics, or business), 2) programming structure knowledge (understanding chunks of code blocks, e.g. plans), 3) interpersonal communication knowledge (human understanding and cooperation), and 4) problem-solving strategy knowledge (rules and process).

Instead of the broad range of knowledge proposed by Brooks, the category of programming knowledge focused on programming. Bayman and Mayer (1988) proposed three types of programming knowledge, consisting of: 1) acquisition of syntactic knowledge (language features and facts), 2) conceptual knowledge (actions, locations, and objects relevant to the program), and 3) strategic knowledge (using both syntactic and conceptual knowledge to solve problems). Although some studies included documentation and communication as part of knowledge (Brooks, 1990; Pennington & Grabowski, 1990), we will discuss this sort of study under social and pair programming approaches.

The category of programming knowledge has been extended to be relevant to the cognitive psychology literature. McGill and Volet (1997) integrated the knowledge dimension (declarative, procedural, and conditional (or metacognitive)) from cognitive psychology literature with Bayman and Mayer’s three types of programming knowledge (syntactic, conceptual, and strategic) into a conceptual framework of various components of programming knowledge. They defined the strategic and conditional knowledge as higher knowledge that can be achieved “when an individual is able to use procedural knowledge (syntactic and conceptual) flexibly and appropriately across novel situations and tasks” (McGill
& Volet, 1997, p283). Therefore, they focused on the integration of the interrelations of the rest of two lower level categories into four types of knowledge, comprising: 1) declarative-syntactic knowledge (e.g. the syntax of a language), 2) declarative-conceptual knowledge (e.g. explanation of semantics and fragments of pseudocode), 3) procedural-syntactic knowledge (e.g. ability to apply rules), and 4) procedural-conceptual knowledge (e.g. the ability to design solutions).

Two studies explicitly emphasised applying the cognitive process dimension of the revised Bloom’s taxonomy to programming. Fuller et al. (2007) proposed a matrix taxonomy for the design and assessment of programming courses by separating six phases of cognitive process into two dimensions: producing (apply and create), and interpreting (remember, understand, analyse, and create). They argued that the matrix could be used to trace novices’ learning and guide them to improve. Furthermore, Thompson et al. (2008) showed how the cognitive process dimension of the revised Bloom’s taxonomy was applied to the programming domain.

Moreover, Bower (2008) argued that programming is not only a body of knowledge, but also a practice. He defined a task type as an activity or process for students to perform. He then claimed that the task types include ten levels, comprising: 1) declarative tasks, 2) comprehension tasks, 3) debugging tasks, 4) predicting tasks, 5) providing-an-example tasks, 6) providing-a-model tasks, 7) evaluating tasks, 8) meeting-a-design-specification tasks, 9) solving-a-problem tasks, and 10) self-reflection tasks. In order to support his proposal of having programming processes in the curriculum, he advocated a taxonomy of these ten types of computing tasks in programming processes.

Both knowledge and cognitive process dimensions in the revised Bloom’s taxonomy have been applied to programming. A well-known literature review (Robins et al. 2003) on novice teaching and learning categorised various practical issues into a programming framework. One dimension of this framework is about the process of problem-solving in computer programming consisting of the phases of: 1) designing, 2) generating, and 3) evaluating. Another dimension is about novice’s attributes comprising: 1) knowledge, 2) strategies, and 3) mental models. In order to make novices become effective learners, they emphasised that teaching programming strategy is more important than programming knowledge. Beyond this framework, they also emphasised the factors of motivation, engagement, confidence and emotional responses.
Similarly, Pennington and Grabowski (1990) proposed a framework of tasks of programming. They classified programming tasks into four types, comprising: 1) understanding the problem, 2) design, 3) coding, and 4) maintenance. They also defined the required knowledge and resources for success in these types, comprising: 1) basic process (or strategy, such as composition and comprehension), 2) knowledge (e.g. domain knowledge, such as accounting, physics, management; design strategies and design language; computer language; and programming environment), 3) mental representation, and 4) external representation (documentation or communication).

Most existing taxonomies related to teaching novices programming either focus on programming tasks based on the cognitive process dimension, or concentrate on programming knowledge based on the knowledge dimension in the revised Bloom’s taxonomy. Only two studies cover both dimensions (see Table 2-2). In the dimension of programming tasks, it is the cognitive process based on Bloom’s taxonomy and particularly the revised Bloom’s taxonomy, applied to computer programming. Some taxonomies only covers parts of programming tasks (e.g. only including comprehension, or missing problem-orientation), others include too many details of the tasks (e.g. ten levels of tasks). In order to use the appropriate categories that are suitable to novices, we consider Pennington and Grabowski’s four categories of tasks (understanding the problem, design, coding, testing and debugging) as part of our taxonomy regarding the cognitive process dimension in the revised Bloom’s taxonomy.

<table>
<thead>
<tr>
<th>Authors</th>
<th>Programming Tasks</th>
<th>Programming Knowledge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bayman and Mayer (1988)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bower (2008)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Brook (1990)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Fuller et al. (2007)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>McGill &amp; Volet, (1997)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Pennington and Grabowski (1990)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Robins et al. (2003)</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Thompson, et al. (2008)</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Regarding to the knowledge dimension, we classify both declarative-syntactic knowledge and declarative-conceptual knowledge in the framework of McGill and Volet (1997) as programming knowledge that corresponds to both factual knowledge and conceptual knowledge in the revised Bloom’s taxonomy. We also classify both procedural-syntactic
knowledge and procedural-conceptual knowledge in the framework of McGill and Volet (1997) as programming strategy that Robins et al. and Pears et al. advocated. Moreover, we also consider the teaching approach of applying programming knowledge and strategies into teaching practice.

Therefore, we choose our programming taxonomy based on Robins’ classification of individual attributes of the programmers (knowledge, strategies, and mental models). Considering there are too many interrelations between programming tasks and knowledge that are similar to McGill and Volet (1997) listed, our taxonomy of literature study of teaching novice programming is based on programming knowledge, programming strategies, teaching method, and mental models of programming.

In the taxonomy of our literature study (see Figure 2-1), the first category is programming knowledge, comprising programming languages (e.g. syntax and semantics) and pragmatic skills (e.g. debugging and feedback), which correspond to novices’ cognitive processes of remembering, understanding, and evaluating in the revised Bloom’s taxonomy. We also note that the correspondence between programming tasks and Bloom’s taxonomy is not one-to-one, because some tasks require a range of skills that span different parts of the Bloom’s taxonomy, e.g. debugging typically includes analysis activities. In other words, novices need to remember and understand the basic programming knowledge such as syntax and semantics in a program as well as to apply the skills of execution, comprehension, and debugging of the program.

1. Programming Knowledge
   1) Programming languages
   2) Pragmatic skills

2. Programming Strategies
   1) Domain knowledge
   2) Programming by problem-solving
   3) Programming process
   4) Programming structure, plans, and related tools

3. Teaching Methods
4. Programming Mental Models

**Figure 2-1 Taxonomy used in our literature study of teaching novice programming**

Next, we focus on programming strategies as many educators (Davies, 1993; Perkins & Martin, 1986; Robins et al., 2003) broadly emphasised. Perkins and Martin (1986) interviewed novices about difficulties and claimed that novices may possess knowledge of a programming language (e.g. language structure or structure of a plan), but they were unable to master this knowledge
because they had limited ability to solve problems by using strategic knowledge. Subsequently, our category of programming strategies need to cover contents from the beginning of understanding the problem to further both general rules and detailed process, to practice support, comprising four sub-categories: 1) domain knowledge (familiarity with the problem), 2) programming by problem-solving strategy (the rule and approach to solve the problem), 3) programming process, and 4) programming structure, plans, and tools (chunks of code in certain patterns and tools of using programming structure and plans to support programming).

We argue that our categories of orientation and problem-solving strategy relate to novices’ cognitive processes of analysing and understanding in the revised Bloom’s taxonomy. Novices should firstly learn how to decompose the problem from the problem domain. Our categories of programming process as well as programming structure, plans, and tools correspond to novices’ cognitive processes of applying and creating in the revised Bloom’s taxonomy. Novices should not only learn programming structures, plans and tools, but also study a programming process for using these structures and tools.

Thirdly, our category emphasises adopting appropriate teaching approach to apply the above programming knowledge and strategies into pedagogical practice. We focus on exploring teaching approaches that are both easy to learn and effective in supporting novices’ learning. These teaching approaches would also engage novices in their learning.

Finally, our category of programming mental models spans all the categories of novices’ cognitive processes in the revised Bloom’s taxonomy. Novices should use mental models from remembering and understanding the notional machine and programming concepts to apply them for analysing the problem towards the representation of mental model. Eventually, they should create and evaluate their solution.

We now proceed to survey the literature by following the taxonomy of Figure 2-1. During the search process, the literature that we explored was further expanded to include more details as in Figure 2-2.
2.4 Programming Knowledge

In order to improve teaching of novices, computer science education research had moved from early cognitive and psychological analysis and empirical studies of novices’ programming knowledge (Linn, 1985; Sheil, 1981; Soloway, Adelson, & Ehrlich, 1988; Soloway & Ehrlich, 1984; Winslow, 1996) to more recent work on multi-national and multi-institutional studies of programming reading and writing related skills (Fincher et al., 2005; Lister et al., 2004; McCracken et al., 2001; Naps et al., 2002). In this section, we start exploring this literature starting with work focusing on students’ knowledge of programming languages and environments (Section 2.4.1), and then shift to work focusing on students’ knowledge of pragmatic skills (Section 2.4.2).

2.4.1 Programming Languages and Environments

Guzdial (2004) overviewed several programming environments from the last forty years and the work on making programming easier for novices. He highly valued Logo (developed in the mid-1960s) and its descendants (such as Smalltalk-72) in 1980s, which originated from the artificial intelligence programming language, Lisp. He argued that the Logo family of programming environments had influenced the development of modern programming
environments towards making it easier to read and write programs. He advocated that new programming environments should support development for compelling domains such as games, multimedia, and simulations. He also expected the new environments to provide immediate feedback on novices’ work. Furthermore, he suggested the option of a “new face”: a higher-level of representation for design activities, such as using GPCeditor, a goal-plan-code programming environment for mapping from a program goal and then to plan code to program code in the Pascal language (see Section 2.5.4).

Kelleher and Pausch (2005) proposed a taxonomy of programming environments and languages for teaching novices. They aimed to lower the barriers to novices programming. Based on literature study of various programming environments and languages, they summarised several approaches for making programming languages more approachable to novices. One approach was to simplify the language by making the syntax more intuitive and easier to remember (e.g. teaching novices by using “x” instead of “*” for multiplication and using “output” instead of “system.out.print” for displaying results). Remembering syntax is a significant challenge to novices. One of the attempts to bypass and avoid syntax errors is to develop graphical objects to represent commands that can be moved and put together to form a program. For example, LogoBlocks² was developed as an extension of Logo³ by MIT Media Lab. It provided graphical blocks representing commands which can be dragged from a tool palette and dropped next to each other on the work area in order to form programs. Alice 2⁴, a programming environment for 3D animations and games, was developed by Carnegie Mellon University in 2003. Its commands can also be dragged and dropped, and parameters for each command must be selected from relevant drop-down menus.

Kelleher and Pausch also studied the programming languages that support and organise different structures of code in different paradigms of programming. For example, Pascal⁵ was widely used for teaching the structured programming. In teaching novices object-oriented programming, BlueJ⁶, and Karel J Robert⁷ were used as stepping stone for Java-style syntax

³ http://el.media.mit.edu/logo-foundation/logo/programming.html
⁴ http://www.alice.org/index.php
⁵ http://www.pascal-programming.info/index.php
⁶ www.bluej.org
while Karel++\(^8\) was commonly used for C++ style syntax. They advocated making multiple styles of programming accessible to novices.

Finally, Kelleher and Pausch concluded that there were three complementary strategies to make programming accessible to novices. The first way was to simplify the mechanics of programming by removing unnecessary syntax and providing immediate feedback. The second way was to provide social support to novices, for example, supporting students working together and providing examples. The third way was to provide motivations for novices to learn programming, such as building robots and games.

Although Java, Python, C++, and C# have been widely used for teaching novices programming, the selection of the first programming language is always contentious because it is a trade-off among different factors. Farooq et al. (2014) proposed an evaluation framework and comparative analysis of the first programming languages based on both technical features (such as consistent language constructs rules, type checking, enforceability of good program writing habits, and less effort for writing simple programs) and environmental ones (such as demand in industry, contemporary features, readable syntax, and user friendly IDE). For each feature, they discussed its importance to the students’ first programming language and defined evaluation criteria. They further divided each feature into a few sub-features and rated each evaluated language against each defined sub-feature. They applied this framework to evaluate a list of leading first programming languages taught from 1994 to 2011, including Ada, C, C++, C#, Java, Pascal, and Python. They concluded that, based on technical features, the top languages were Python, Java, Pascal, and Ada. Based on environmental features, the top languages were Java, Ada, Python, and C#. Java and Python were identified as the top first programming languages based on the overall score for widely used programming languages.

The papers mentioned above were concerned with certain features of novice programming environments such as simpler syntax and immediate feedback. We now turn to finding out what the practice is and start from programming languages used in Australasia. In the last decade, the most popular languages for teaching novices programming have changed from Java, VB, C++, and C in 2001 to Java, Python, C, and C# in 2010 in Australia and New Zealand (De Raadt, 2007; De Raadt, Watson, & Toleman, 2004; Mason et al., 2012). Based on their surveys, Mason et al. (2012) proposed that the major reason for the changes were the shift

\(^8\) [http://csis.pace.edu/~bergin/karel.html](http://csis.pace.edu/~bergin/karel.html)
from focussing on industry relevance and students’ marketability in 2001 to focussing more on pedagogical benefit in 2010.

Regarding programming environments and tools, Mason et al. (2012) also reported that during these ten years, there were changes from many (45%) to fewer (20%) institutes not using an IDE (Integrated Development Environment, e.g. Microsoft Visual Studio, or Eclipse) or tool (e.g. BlueJ, or Alice). They advocated that pedagogical reasons were the most frequent reason for selecting the programming environment.

In the late 1990s, Java was widely used in many universities and institutes because of its advantages in supporting object-oriented programming, concurrent programming, and Internet development (Hadjerrouit, 1998). In order to select the first programming language, Hadjerrouit (1998) advocated pedagogical consideration such as language simplicity and programming paradigm as well as industry requirements. He evaluated the suitability of using Java as the first programming language for teaching novices programming. After three years’ experience of teaching novices both Simula and C++ languages, he evaluated curricular and pedagogical issues based on his observations and discussion with students as well as students’ evaluation and their performance. He found that students had many of the same basic programming problems in Java as in C++, for example, the issues of using semicolons, loops, conditionals, etc. He also found that students had additional problems using the debugging and compiling tools of Java. Moreover, he found that the intended Internet development was one of the motivations for novices to learn Java. Finally, he concluded that:

“Learning Java as a first language to support introductory programming turned out to be more difficult than originally anticipated. Java is a relatively difficult language for students with no programming background. It is more suitable for teaching students with some programming knowledge, particularly in C/C++... But teaching Java as a first programming language is not a problem of technology, it is a pedagogical problem as well” (Hadjerrouit, 1998, p47).

With similar concerns as Hadjerrouit (1998) regarding the programming environment for novices to learn object-oriented programming, Kölling (1999b) argued that “a suitable programming environment is crucial for the success of an introductory course. Of all the problems reported by educators connected to teaching object-orientation, problems with the environment used were the most frequent and the most severe” (p6). After analysing the
suitability for teaching and support for object-orientation from the existing programming environments, he concluded that “Those environments which offer good support for object concepts are too complex to be effectively used in a first year teaching course. Problems with environments have been identified as the most common problems with teaching object-orientation to undergraduates” (p12).

Farag, Ali, and Deb (2013) studied the influence of using different programming languages in the introductory programming course. They changed their online introductory programming courses from C++ to Java. They applied indirect and direct assessment for the evaluation. In the indirect assessment, they evaluated students’ perception of course efficacy by using a survey of the students’ perceptions of each of the course components including syllabus, tools, projects, and exams, as well as a survey of students’ online experience of support and learning. They also compared students’ satisfaction levels when using the different languages. In the direct assessment, they measured achievement of Intended Learning Outcomes (ILOs) including effective use of the IDE, the ability to develop programs, the ability to debug and test, and the use of language structures. They also compared students’ success such as the chances of achieving the highest grade and a failure grade as well as the overall grade percentage. However, none of the comparisons showed a significant difference between using C++ and Java to teach novice programming.

Further to the above concerns regarding programming environments, Kölling et al. (2003) argued that “Our hypothesis is that teaching object-orientation is not intrinsically more complex, but that it is made more complicated by a profound lack of appropriate tools and pedagogical experience with this paradigm” (p249). They then introduced BlueJ, a Java IDE specifically developed for introductory object-oriented programming. With BlueJ, students can use build up a Unified Modelling Language (UML) class diagram and create code for classes and methods from the provided templates based on the class diagram. An object inspector and visualisation were also provided for students to understand OO concepts. Moreover, in order to provide an effective pedagogical environment for progressive learning, they also provided guidelines for learning in the order of executing, reading, modifying, and using provided examples and a sequence of assignments. They used a fill-in-the-blanks educational pattern to let students complete method bodies under provided method signatures. Although students can start seeing objects at the beginning by using BlueJ and its pedagogy, they still have to deal with the difficult issues of Java syntax and language structures during working.
Although Java and Python are widely used in introductory programming, they still pose challenges to novices in their syntax. Therefore, people have still done work on environments, as well as explored alternative languages (like Scratch\(^9\)). Following the previous development of Logo (in the 1980s) and LogoBlocks (in the 1990s), the Scratch visual programming environment was launched in 2007 (Maloney et al., 2010; Resnick et al., 2009). It provides coloured command blocks, which are categorised in order to help novices to identify them. Each command or program fragment can be executed independently to provide immediate feedback without waiting to complete the entire program. Resnick et al. (2009) and Maloney et al. (2010) argued that Scratch is “tinkerable”, since it allows the user to execute snippets of code: “Tinkerbility encourages hands-on learning and supports a bottom-up approach to writing scripts where small chunks of code are assembled and tested, then combined into larger units” (Maloney et al. 2010, p16:4). It also makes command execution visible by highlighting the currently executed command and showing the value of variables on the “stage”. However, it only supports three data types: Boolean, number, and string. Numbers and strings can be converted automatically during processing. It also does not support customised procedures.

To make up for the notable disadvantage of Scratch lacking procedures, an extended reimplemention of Scratch was developed with the ability to “Build Your Own Blocks”, namely BYOB\(^10\) (Harvey & Mönig, 2010). BYOB also provides nestable lists, meaning a list can be embedded as an element in another list. Harvey and Mönig (2010) proposed that other data structures (e.g. hash tables, trees, etc.) can be implemented as a list of lists and defined in libraries for further applications.

Scratch and BYOB have been applied in teaching in various contexts. Maloney et al. (2008) studied urban youth learning programming with Scratch at a Computer Clubhouse. They found that Scratch simplified the learning of programming by preventing syntax errors and providing immediate feedback in the practice. They also emphasised that it was important to provide support (e.g. through social infrastructure) and assist engagement (e.g. with multimedia) to novice programmers, which supports the arguments of Kelleher and Pausch (2005), and of Guzdial (2004).

---

\(^9\) https://scratch.mit.edu/
\(^10\) http://snap.berkeley.edu/
Fincher et al. (2010) compared three well-known “Initial Learning Environments”: Alice\textsuperscript{11}, Greenfoot\textsuperscript{12} and Scratch. They all support novices making animations and games. Both Alice and Scratch prevent novices from making syntax errors by providing command blocks in a drag-and-drop programming environment while Greenfoot provides Java code templates. Both Alice and Greenfoot support object-oriented programming concepts while Scratch works for procedural programming study. It seems that Scratch has the advantages of preventing syntax errors and supporting the procedural programming paradigm using a simple 2D (vs. 3D in Alice) graphics.

Fesakis and Serafeim (2009) studied the influence of learning Scratch on programming. For non-computer major university students, they used Scratch as a programming environment and performed surveys of students’ attitude about ICT education pre and post experience. They also did surveys for the assessment of Scratch as well as for the attitude about Scratch and programming. They concluded that “after the familiarization with Scratch students show increased self confidence in exploitation of ICT in education. There are more students that do not declare anxiety and stress about their ability to use ICT as well as students that wish to develop their own educational software applications... The research results validate the educational decision of using Scratch in a course of computer programming for future teachers” (p261).

Meerbaum-Salant, Armoni, and Ben-Ari (2013) investigated teaching computer programming concepts at high school by using Scratch. They evaluated their teaching by combining parts of both the revised Bloom taxonomy (Anderson et al., 2001) with SOLO (the Structure of the Observed Learning Outcome) taxonomy (Biggs & Collis, 1982). For the evaluation, they used three categories from the revised Bloom taxonomy (understanding, applying, and creating) and another three from the SOLO taxonomy (unistructural, multistructural, and relational), combining them into a nine-level taxonomy from the lowest level of unistructural understanding to the highest level of relational creating. They found that students were able to understand computer concepts at a reasonable level except for the concepts of loop, and variables. They argued that these difficulties could be solved by explicitly teaching in detail the relationship between concepts and code in Scratch. They also argued that without close and

\textsuperscript{11} http://www.alice.org.
\textsuperscript{12} http://greenfoot.org.
effective mentoring, many students could only use Scratch to create media, and learn very little of programming.

Furthermore, Malan and Leitner (2007) introduced Scratch in higher education as a transition to Java at the beginning of summer school for students with no previous background in computing. Through a survey of 25 students, they found that most students believed that Scratch had positively influenced their experience with Java. They advocated that the simplicity of Scratch (e.g. no syntax distraction) had engaged and encouraged the students’ learning from Scratch to Java.

Gibbs and Coady (2010) used Scratch for the first programming assignment of CS1. Through the survey, they found that most students successfully developed a working program in the Scratch environment and even explored further than the assignment required. By analysing students’ performance, they also found that many students were not able to abstract their solutions effectively, for example, their solutions were either too complicated, or too general. They concluded that “though they were most likely successful with the given task, they did not connect with any deeper intellectual merit to the activity, and did not obtain an appreciation of the power of abstraction” (p173).

In previous study, Scratch had shown advantages for overcoming the difficulties of using a programming environment by preventing syntax errors, providing immediate feedback, and helping to understand programming concepts. However, Lister (2011a) had concerns about students’ learning as similar to those raised in the previous study. He argued that

“I am convinced that students learning to program via tools like Scratch and Alice will make better early progress than students who must fight a compiler. However, many students, irrespective of how they are first introduced to programming, will still reach a point where they hit a cognitive wall...Inevitably, there will be a backlash against these tools, just as there was a backlash against teaching objects early. People will say ‘the students are still having problems’, and these people will be right, but the fault will not be with these tools. The fault will lie with the absence of a pedagogical rethink of what should happen after these tools” (Lister, 2011a, p21).

Lister (2011c) also argued that novices may be motivated by using the new programming environments, but motivation and time on task is not sufficient to ensure novice learning. He
advocated combining these environments with deliberated learning practice, which is highly structured and monitored to provide feedback for further improvement, although it requires effort and engagement.

Recent experience supports Lister’s (2011a, 2011c) argument of pedagogical consideration when using these visual programming environments. Inspired by using Scratch, particularly BYOB, Giordano and Maiorana (2014) trialled a new approach that combined three aspects: process, guided inquiry, and a visual programming environment in a small class size at high school. They used a series of short questions to guide students’ knowledge building by the process-oriented guided inquiry learning approach. They also applied an instructor-guided inquiry approach to foster feedback in all class discussions. They concluded that the preliminary lessons should be a “procedure-first” approach starting with simple blocks in BYOB to learn the procedural abstraction and then translate the BYOB program to another language, e.g. C. They argued that it is promising to combine pedagogical strategies, content, and visual programming environments such as Scratch or BYOB.

App Inventor\(^\text{13}\) is a Scratch-like visual programming environment for developing Android mobile application. Unlike Scratch, it supports mapping from blocks into Java code. Robertson (2014) studied the usage of App Inventor to teach first-year students programming. She found that students were frustrated by bugs with the environment. Although students believed that App Inventor was easy to use for mobile application development, they considered that it was not as flexible as the professional Java IDE, e.g. Eclipse. She suggested using App Inventor only for a short period of time, e.g. six weeks, and then moving to a fully featured IDE.

Papadakis et al. (2014) compared Scratch and App Inventor for novice programming, considering both strong and weak points in the two programming environments. They concluded that Scratch was appropriate for introducing programming to young students, rather than teaching them deep knowledge. They proposed that after Scratch was introduced in the early class, App Inventor could be used in the next class because it generated Java code.

In summary, various programming languages have been used in teaching novices programming in the last three decades. The selection of a first programming language for novices is a trade-off between pedagogical considerations and industrial demands. Recent literature studies have

\(^{13}\text{http://appinventor.mit.edu/explore/}\)
shown that Java and Python are recommended as programming environments for novices even though they do have weaknesses and issues (e.g. syntax). However, considering pedagogical benefits, some educators have started moving towards visual programming environments to attract and motivate young students. Scratch, particularly its extended reimplementation, Build Your Own Blocks (BYOB), has advantages in terms of starting with simplicity, preventing syntax errors, providing immediate feedback, and helping to understand programming concepts, which make it a promising introductory platform, before transitioning to Java or Python.

2.4.2 Pragmatic Skills in Programming

The programmatic skills for novices in computer programming that we considered in this study are editing, compiling, testing, and correcting the program. The editing and compiling skills largely depend on the features (e.g. visual and textual, interpreted and compiled) of the programming language and programming environment used. The main focus of this section is therefore debugging skills. A bug is regarded as an error in a program which causes the program being unable to achieve the predicted results. Debugging is also considered as a process to locate and correct bugs (Xu & Rajlich, 2004).

Debugging was considered as one of the most pervasive tasks in the programming process (Vessey, 1989), a central part of programming (Gugerty & Olson, 1986), and a bottleneck in programming (Ducasse & Emde, 1988). Debugging is one of the essential skills for successful programmers (Fitzgerald et al., 2010). However, Ahmadzadeh, Elliman, and Higgins (2005) have argued that “The skills and abilities that distinguish experts from novices in program debugging and the affect [sic] of those competencies on the ability to progress in programming seem to have been largely ignored in recent years” (p84). Further research discovered that debugging was difficult for novices (Fitzgerald et al., 2010; McCauley et al., 2008; Murphy et al., 2008).

McCauley et al. (2008) studied the literature in the areas: 1) why do bugs occur; 2) what types of bugs occur; 3) what kind of debugging process is used; and 4) how to improve the teaching and learning of debugging.
We now focus on the first question that McCauley et al. proposed: why do novice programs have bugs? Answers to their questions can be found in the earlier work by Soloway and his colleagues (Pea et al., 1987; Spohrer & Soloway, 1986; Spohrer et al., 1985) who had carried large numbers of studies about novices programming bugs in 1980s. Bonar and Soloway (1985) argued that bugs produced by novices were because of mismatch between “natural” thinking (as in natural language) and programming languages/models after interviewing novice programmers. Their initial findings were that bugs were due to inappropriately using a step-by-step procedure from natural language into programming course. For example, novices easily were confused between the word “then” in natural language for a following step and “if-then-else” structure in programming language for one of two options under the specified condition as well as between the word “while” in natural language for a continuously active test and the “while” loop in programming language for the loop condition getting a test once per iteration. They advocated the linkage of programming knowledge from step-by-step natural language to programming plan knowledge. This linkage is in a goal-plan setting with some classification and leads naturally to next paper of discussion about goals and plans.

In further study from the point of view of goals and plans, Spohrer and Soloway (1986) analysed the frequent and less frequent mistakes that novices are likely to make. They argued that “bugs seem likely to occur when students are unable to coordinate and integrate the goals and plans that underlie program code” (p632). They advocated teaching novices programming composition strategies by using goals and plans rather than only constructing plans as in the above study.

Pea (1986) claimed that there were language-independent conceptual “bugs” that exist in novice programming. He classified these conceptual bugs into: 1) parallelism bugs (novices assumed that different statements in a program can be processed at the same time without considering the order of these statements); 2) intentionality bugs (novices wrote program code beyond the provided information about the problem); and 3) egocentrism bugs (novices assumed that computer can do more than that was told by the program). From the literature study, he concluded that the conceptual bugs are attributed to diverse misconceptions that plague novices.

McCauley et al. (2008) concluded that most errors were caused by a chain of novices’ cognitive breakdowns in skills, rules, or knowledge. They argued that errors in programming plans were
caused by breakdowns of novices’ cognitive rules. The broken down rules might be caused by fragile knowledge. Therefore, they suggested that knowledge breakdowns might be attributed to both fragile knowledge and incorrect programming plans. Based on the literature study, they also concluded that both computer language errors and missing something or malformed elements of program structure caused most of the novices’ bugs. They advocated for investigations into the use of new programming languages and environments such as Alice and Scratch.

We now turn to types of novice programming bugs that have been identified in the literature. The errors were classified as syntax errors, run-time semantic errors (errors that passed compilation, but failed at run-time, e.g. dividing by zero), and logic errors (errors that did not trigger a run-time error, but produced incorrect results) (McCauley et al., 2008).

According to the distribution of bugs in different parts of programs, Johnson et al. (1983) categorised a total of 783 bugs in 206 novices’ programs into eight categories: 1) inputs (incorrect usage of input statements), 2) outputs (incorrect usage of output statements), 3) initialisation (setting variables with incorrect initial values), 4) updates (incorrect updating of variables), 5) guards (incorrect testing conditions of statements), 6) syntactic connectives (incorrect syntax for delimiting a block structure, e.g. begin, and end), 7) complex plans (incorrect constructions of plans when plans have more than one plan component, e.g. initialisation, updates, and outputs), and 8) declarations (incorrect setting name and data type for variables). They argued that these bugs were not random occurrences, but were systematic due to novice misconceptions. They attempted to relate the bugs to the misconceptions and to program constructions given missing plans, misplaced plans, spurious plans, or malformed plans.

Ahmadzadeh et al. (2005) categorised programming bugs among novices into compiler errors and logical errors. They also divided the compiler errors into: 1) syntax errors relating to grammar, order of tokens (e.g. operators), or symbols missing (e.g. semicolons); 2) semantic errors relating to consistency in the program (e.g. misuse of local and global variables); and 3) lexical errors with unknown tokens. They modified a Java compiler to store the compiling messages from students’ Java programs. From more than one hundred-and-thousand (108,652) error records collected during one semester, they found that there were 36% syntax errors, 63% semantic errors, and only 1% lexical errors.
In summary, novices’ programming bugs can be categorised as syntax errors, semantic errors, and logic errors. Semantic errors and logic errors were considered as the major types of bugs that novices had in their programming practice by Ahmadzadeh et al. (2005). Syntax errors are mainly from incorrect usage of programming language, which can be detected by compilation. However, both semantic errors and logic errors cannot be detected directly by compilation. Subsequently, strategies are needed for the debugging of both semantic errors and logic errors.

Now we turn to look at how novices debug, i.e. what strategies they use, and how effective these strategies are. Vessey (1985) investigated debugging processes by both experts and novices. She recorded their problem solving process using a verbal protocol. From her analysis, she found that experts were proficient at using chunks of programs and used breadth-first approaches for debugging (e.g. checking on clues at the same program structure level and then moving to the next level) while novices with less proficiency at chunking programs used depth-first approaches for debugging (e.g. checking on clues in the details of program structure).

Furthermore, Vessey (1989) claimed that program comprehension played an important role in debugging, particularly when working on a program written by other people. Through an empirical test of program bugs, she advocated paying substantial attention to programming processes. She emphasised that in the programming process it is important to have a mental model of correct program functioning for debugging of logical errors.

Ducasse and Emde (1988) proposed a classification of debugging knowledge, strategies, and the corresponding techniques to support the strategies. Based on 18 automated debugging systems and a dozen cognitive studies on debugging, they identified several knowledge types for debugging, including knowledge of the intended and actual program, knowledge of programming language and programming expertise, knowledge of the application domain, and knowledge of bugs and debugging. They also identified four strategies for debugging that can be used as single or combined strategies, consisting of: 1) filtering by tracing and slicing the program; 2) checking computational equivalence between the intended and actual programs; 3) checking whether the program is well-formed; and 4) recognising stereotyped errors from the known bugs.
Following Ducasse and Emde’s study, Murphy et al. (2008) investigated novices’ debugging strategies using interviews of 21 students in a survey-programming-debugging-survey pattern. They found that the most commonly used strategies were tracing, testing, understanding the code, using resources, using tools, and isolating the problem. However, these strategies were not always employed effectively by all the students. Although some strategies (e.g. gaining domain knowledge, pattern matching, considering alternatives, and environmental changes) were effectively supporting students to successfully debug, other strategies (e.g. working around rather than facing the problem, doing unnecessary things, or tinkering) were employed with less effect. They concluded that even though students used many debugging techniques, they applied these techniques ineffectively or inconsistently. They advocated tracing and testing strategies such as:

- “If you have to track more than one or two variables or there’s a loop involved then you should trace on paper.
- If the bug can’t be determined from the input and output then you need to add print statements.
- Make sure that your print statements are well-placed and print meaningful information.
- If you’d have to use many print statements, your program ‘hangs’ or has an infinite loop, use a debugger” (p166).

Recently, Fitzgerald et al. (2010) restudied all of the above thirteen novices’ strategies proposed by Murphy et al. (2008). They assumed that the debugging strategies developed by students as a by-product of programming would not be effective. They interviewed 21 students and observed them debugging. They argued that pattern matching by using similar examples for writing and debugging code is an important technique for novices. They advocated debugging strategies specific for students from Murphy’s entire strategy list except for three items (isolating the problem, taking advantage of programming environment, and working around the problem). During the interview, they found that the strategy of isolating the problem was considered as the common practice of commenting out part of the code by students. They considered that environmental strategies for simply taking advantages of the features of programming and debugging environments (e.g. using the undo command for recovery of correct code and using comments for inline documentation). They also considered that working around the problem required knowledge of programming plans or schemas (see Section 2.5.4) by novices to replace the code with a known schema. Moreover, based on the
interviews with students, they added a new “think” strategy of pondering and reflecting on the possible solution when debugging.

Based on the past 20 years of work, Fitzgerald et al. (2008) advocated that domain knowledge and program comprehension are the key factors for the success of debugging. They also found that students were able to fix bugs after they located them. They argued that:

“Similar to Soloway and Spohrer’s 1986 findings, our subjects had more difficulty with language-independent bugs rather than bugs related to misunderstanding or confusion about the language. Arithmetic errors were particularly troublesome debugging problems. Malformed statements were the most difficult bugs to fix. Loop conditions, conditional logic, arithmetic errors, and data initialization and updating were difficult bugs to find“ (Fitzgerald et al., 2008, p115).

Inspired by Bonar and Soloway’s (1985) consideration that many bugs result from a mismatch between students’ pre-programming knowledge expressed in natural language (e.g. English) and their programming knowledge conveyed in Pascal, Simon et al. (2008) argued for identifying novices’ pre-existing abilities relevant to debugging and troubleshooting experiences before teaching programming. To address the gaps in their debugging strategies, they asked 305 students from six institutes to answer four real-life questions that were similar to debugging scenarios. These four questions were: 1) “light bulb”, which required students to write instructions to a visitor if the light in the bedroom did not come on; 2) “telephone”, which attempted to find out why a sentence differed from the original one after being whispered by students in a circle; 3) “coffee”, which asked students to describe how to locate the nearest Starbucks coffee shop in a foreign country where students did not understand the language; and 4) “real life”, which required them to describe a process of troubleshooting in their daily life from identifying the problem, through learning the problem to solving the problem. They intended to explore how some debugging strategies were used by students such as using domain knowledge, using deep knowledge to locate errors, seeking help, and employing problem-solving strategies. From the analysis of answers, they found that:

“students often provided a structured process for solving the problem, indicating that they were generating and following a plan as they worked ... Students have an ability to test, but they do not consider it a strategy that is mandatory for troubleshooting ... Neither detailed use of domain knowledge nor employing a strategy for gaining an understanding of the system (the first step of generalized troubleshooting) was evident
in student response ... they rarely used deep knowledge of the domains ... seeking help from others was only common in the coffee shop scenario” (Simon et al., 2008, p128).

Although they found that students’ pre-existing strategies were less relevant to debugging, they argued that the real world based scenarios could help novices understand debugging issues and processes.

To support teaching of debugging, Simon et al. (2007) proposed that video vignettes of the debugging process could provide “scaffolding” to motivate novices’ debugging rather than them getting “stuck” by feeling difficult, confusing, and lonely. They made 30 video clips covering the debugging process from locating to fixing 26 common Java bugs. The debugging videos were categorised into: 1) compile time errors (e.g. missing semicolon, or assignment operator (=) used instead of an equality operator (==)), 2) runtime errors (e.g. no class definition found, or index out of bounds), 3) logic errors (e.g. empty a while loop body, infinite loop, incorrect iterator update, missing input statement, trying to return a value through a parameter with a primitive data type), and 4) process modelling (e.g. using a debugger to set a breakpoint and then using the “step into”, “step out”, and “set over” functions).

McCauley et al. (2008) reported that bugs were rooted from preconceptions, misconceptions, and fragile knowledge of programming and programming language. They advocated combating these “bug” roots, building program comprehension skills, and explicitly teaching debugging skills and tools.

Turning now to tools to support debugging, Ko and Myers (2004) argued that in the last 30 years debugging tools provided similar ways of locating bugs by using breakpoints, code-stepping, and print statements for the observation of control flow.

“Commercial debugging tools are notorious for hidden dependencies: code stepping tools show runtime data on the call stack, but not the data that depends on it or that it depends on. Print statements reveal relevant runtime data, but hide the context from which the data was printed. Another issue is viscosity, or resistance to local changes. For example, inserting ten print statements to observe runtime events requires removing them later; trying to use a code stepping tool to show data from the past requires reexecution. These hindrances to exploration may lead to debugging errors” (Ko & Myers, 2008, p152).
They developed an interrogative debugging interface for novices by asking “why did” and “why didn’t” questions. They assumed that the “why did” questions explained the occurrence of an unexpected runtime action while the “why didn’t” questions described the absence of an expected runtime action. The interrogative debugging interface, named Whyline — a Workspace that Helps You Link Instructions to Numbers and Events, was embedded in Alice.

“The idea is simple: rather than requiring people to translate their questions to code queries, the Whyline allows developers to choose a why did or why didn’t question about program output and then the Whyline generates an answer to the question using a variety of program analyses. This avoids the problems noted above because developers are much better at reasoning about program output, since unlike the execution of code, it is observable” (Ko & Myers, 2008, p301).

From testing in the Alice environment with nine students in Master’s program who had experience of other programming languages, they found that the Whyline significantly reduced debugging time (an average factor of 7.8) by asking questions about mapping from question to related code.

Subsequently, Ko and Myers (2008) implemented the Whyline for Java based on their experience in the Alice environment. Through empirical evaluation, they found that novices with Whyline were able to debug twice as fast as experts without it. However, because Whyline was a trace-based approach, it cost more time than other debugging tools due to loading the Whyline trace. The programming language could also affect the precision of the Whyline’s “why didn’t” answers.

“Answers to why didn’t questions are not intended to be a definitive explanation for why something did not occur, but rather a set of potential explanations. After all, there are many possible fixes to any given problem, and only the developer is capable of choosing the appropriate modification” (Ko & Myers, 2008, p309).

In summary, program errors have been mainly classified as programming language related errors, i.e. syntax errors, and language-independent errors, i.e. semantic errors and logic errors, although there were many types of error in each category. The cause of these bugs was considered from both novices’ errors in using a programming language and novices’ preconceptions, misconceptions, and fragile knowledge of programming and the programming language. Debugging is one of the most important skills and a problem-solving task rather than a by-product of programming because debugging includes not only programming knowledge,
but also its own pragmatic skills. This is because debugging required application domain knowledge, knowledge of programming language, knowledge of program comprehension, programming process, and a mental model of programming. It also needed knowledge of bugs and debugging. The strategies of teaching debugging included starting from real-world scenarios, tracing on paper, adding print statements, using code stepping tools, using Whyline tools, and using videos about the debugging process.

2.5 Programming Strategies

“In teaching programming, I believe we are trying to teach a skill, and that means teaching programming principles and strategies that we use in developing programs...If you are a programming instructor, ask yourself whether you teach simply facts, or whether you teach students about the program-development process; if you are a textbook author, look at your text and see which principles/strategies of program development you actually explain and illustrate” (Gries, 2002, p7).

Davies (1993) argued that many studies are only concerned with the content and structure of programming knowledge, but fail to consider how to use and apply the knowledge. He advocated the role of strategy in programming (particularly in program comprehension and generation). Based on literature studies, he claimed that “studies have been explicitly concerned with the kinds of difficulties experienced by novices, and in particular with those difficulties that arise because of an absence of elementary problem-solving strategies or because of a reliance upon inappropriate strategies” (p239).

Moreover, de Raat, Toleman, and Watson (2004) observed that experts applied strategies derived from previous problem-solving experience to write programs. In order to take advantage of the kind of problem-solving strategies employed by experts, they adopted Soloway’s (1986) theory (cf. Section 2.5.4) by using a goal and plan based framework in their curriculum.

Before problem-solving, the first step is to understand the problem to be solved. Two critical features of understanding the problem were proposed by Pennington and Grabowski (1990). One is application domain knowledge; another is the mental representation of the problem (cf. Sections 2.5.4 and 2.6).
We start focusing on the problem (application) domain to understand the problem to be solved (see Section 2.5.1). We then explore programming by using problem-solving strategies (see Section 2.5.2). Following this, we consider details of the program-development process (see Section 2.5.3). Finally, in order to choose an appropriate strategy and an effective programming process, we study various problem-solving strategies employed by experts (see Section 2.5.4).

2.5.1 Domain Knowledge

Gaining domain knowledge (information related to the problem to be solved or problem orientation) was identified as the first difficulty faced by novice programmers (du Boulay, 1986). This point is backed up by Pennington and Grabowski (1990) who observed that programming involves solving problems in another application problem domain (such as accounting, physics, or management), and therefore a first step is to understand the problem in the application domain. Letovsky (1987) proposed that the knowledge base for programmers includes both programming knowledge and application domain knowledge. Guindon (1990) advocated that “There is little hope of understanding this [design] process without identifying the domain of knowledge designers bring to bear, and how designers exploit that knowledge in searching for a satisfactory solution” (p285). In order to understand the problem to be solved, the possession of application domain knowledge was emphasised by many educators (Adelson & Soloway, 1985; Brooks, 1990; Pennington, 1987; Pennington & Grabowski, 1990; Letovsky, 1987; Wiedenbeck, 1999).

Prieto-Díaz (1990) proposed a domain as a field or application area for programming development such as banking, payroll, or a control system. He argued that domain analysis was mainly associated with accumulated experience. He advocated developing a domain analysis process by collecting information in the existing systems. With this process, knowledge and abstractions were organised and managed into domain models, development standards, and reusable components (e.g. a library). In order to explain domain analysis, Rugaber (2000) emphasised application domain models with real-world objects such as tax rate tables. He argued that “a domain model can act as a schema for controlling the program understanding process and template for organizing its results” (p146).
A variety of programming knowledge and strategies in familiar and unfamiliar application domains have been analysed in the literature (Adelson & Soloway, 1985; Shaft & Vessey, 1995). Adelson and Soloway (1985) argued that a designer would not have the same knowledge and skills of objects in different application domains because the designer had different degree of understanding of various domains. They conducted a protocol analysis using a “thinking aloud” approach with three expert designers and two novice designers by using verbal reports from designers while watching a video record of their words, actions, etc. during their problem-solving session. They concluded that when having sufficient application domain knowledge, the designers would use simulation of design-in-progress (a dynamic representation of design progress) by integrating familiar material in novel ways as well as making notes for systematic expansion. Alternatively, when having insufficient application domain knowledge, the designers would develop constraints on the design in order get enough specificity with the object before using the simulation. Finally, they concluded that the designers would use a plan instead of the above methods after having an appropriate problem-solving plan from experience.

Shaft and Vessey (1995) investigated the relationship between application domain knowledge and program comprehension. They argued that experienced programmers who had more application domain knowledge would conserve effort by using a top-down comprehension process because the top-down problem-solving process was more parsimonious than the bottom-up process. On the other hand, programmers who were unfamiliar with the application domain would engage in an in-depth bottom-up comprehension of program details. They confirmed their hypothesis by using their coded verbal protocol data to analyse 24 professional programmers’ comprehension process of two COBOL programs, each having more than 400 lines of code, one in the familiar payroll domain and the other in an unfamiliar hydrology domain.

Furthermore, Khatri et al. (2006) argued that application domain knowledge did not influence the performance of syntactic and semantic comprehension tasks (e.g. using syntax of an entity-relationship (ER) model), but influenced the solution of problem-solving tasks by using knowledge represented in schemas (e.g. an ER model). These types of tasks were also named as schema-based problem-solving tasks. They examined 81 students in the familiar application domain of sales (e.g. an order-processing application) and in the unfamiliar application domain of hydrology (e.g. a ground water application) by using ER models. They found that
Information Systems (conceptual modelling) domain knowledge such as representations, methods, techniques, and tools affected the solution of all types of schema-understanding tasks. However, application domain knowledge only affected the solution of schema-based problem-solving tasks.

In summary, problem orientation requires understanding the problem application domain because it is important to problem solving. Previous studies of problem-solving in both familiar and unfamiliar application domains show that familiarity with the application domain has benefits to program design, program comprehension, and program solution. Although application domain knowledge can be accumulated from experience, we believe that it would be beneficial for novices to solve problems in a familiar application domain when learning programming.

2.5.2 Programming by Problem-solving

“Competent programmers need both well-organized knowledge of a programming language and problem-solving skills. We refer to programming-specific versions of more general problem-solving approaches as design skills. They include generating alternative solutions to a problem, comparing the alternatives, implementing solutions one piece at a time, testing the solution to a computer problem, debugging the solution when the tests reveal deficiencies, and understanding existing code” (Linn & Clancy, 1992, p 121).

The problem-solving approach in the programming context was mainly adapted from the idea by Polya (1957) of solving a mathematical problem in a four-phase process (understand the problem; devise a plan; carry out the plan; and look back) (Abboud, 1994; Barnes, Fincher, & Thompson, 1997; Gomes & Mendes, 2007a). Abboud (1994) argued that problem solving is an iterative process from a first solution to a new solution through the use of testing. She suggested that the process of problem solving consisted of the following four steps: 1) analysing the problem; 2) attempting a “solution”; 3) evaluating the solution; and 4) if it is not a solution, making another attempt and checking the new solution. She also advocated testing a solution by tracing the execution of the algorithm (viz. simulation) before writing the code.

Influenced by Polya’s idea, Barnes et al. (1997) proposed a four step structured process of “understand, design, write, and review”. Specifically, in the understanding phase, they
suggested collecting sample inputs and expected outputs as a test data set. In the designing phase, they emphasised using problems similar to those the students had already solved as assignments to maximise learning impact. A test data set collected from previous phases was used in this phase to evaluate the design. The writing phase was the implementation of the design using the target language. The final reviewing phase was to reflect and look back on the learning process.

In a multi-national study that found many students did not know how to program, McCracken et al. (2001) advocated that “a fairly universal expectation is that they should learn the process of solving problems in the domain of computer science, in order to produce compilable, executable programs that are correct and in the appropriate form” (p126). They proposed five problem-solving steps for the success of programming, 1) abstracting the problem from its description, including identifying the relevant aspects of the problem and modelling those elements; 2) generating sub-problems (or functional decomposition); 3) transforming sub-problems into sub-solutions (implementation and tests in modularised and standardised forms); 4) re-composing (putting the sub-solutions together to generate the solution to the problem); and 5) evaluating and iterating (testing and revisiting the earlier steps). In this framework, the claim of testing the execution of an algorithm before coding by Abboud (1994) is also backed by emphasising testing early (i.e. testing from step 3) in the process.

Based on a literature study, Gomes and Mendes (2007b) proposed six stages of programming through problem solving and identified several types of abilities that novices needed to solve problems and to become experts in programming. In the programming problem-solving process, they began with five steps similar to those proposed by McCracken et al. (2001), and also added the last step of communication of the problem solution by selecting appropriate media and representation to exchange ideas to an outside audience. They proposed that novices need numerous abilities in the programming problem-solving process. Particularly, various reasoning abilities are essential to solve a problem and communicate the result. Finally, they advocated teaching experts’ problem-solving procedures to novices.

“So, in order to improve problem solving among students, we think that it is important to understand the advantages that expert problem-solvers have and transform these advantages into problem-solving directions. Perhaps by teaching expert’s problem-solving procedures to novice students, they will be able to improve their abilities, approaching the knowledge framework of experts” (Gomes & Mendes, 2007b, p222).
Also based on a literature study, Ismail, Ngah, and Umar (2010) argued that problems in the programming process were mainly in the problem-solving phase (such as analysis and design) and the implementation phase. Based on analysis, they proposed that the problems were due to the lack of problem-solving skills. They proposed that the problems in the design phase were due to inefficient tools for representing the solution, lack of semantic knowledge, and weakness in testing the design. They proposed that the problems in the implementation of a detailed solution were due to a lack of syntactical and semantical knowledge, as well as an ineffective approach to coding and testing. By interviewing five expert lecturers, they also found types of problems similar to those that were identified in previous studies of problem solving.

Deek, Kimmel, and McHugh (1998) argued that students got mired in the difficulties of syntax concepts. They proposed teaching problem-solving first and then programming. They also emphasised the importance of teaching problem-solving independently from programming language. In this approach, students can concentrate on problem solving and testing strategies. After the solution is constructed, the language syntax is then used to translate the solution into code and to test the final program. They proposed an interactive pedagogical process, comprising: 1) instructors presenting the problem; 2) students progressing from initial understanding of the problem to a precise formulation; 3) students setting up solution plan using goal decomposition into subgoals as well as tasks for accomplishing each subgoal; 4) students developing a solution design from a high-level design in a framework to a detailed design by transforming subgoals into corresponding algorithms; 5) students working on an algorithmic walkthrough to identify the exact language structure required for implementation; 6) instructors teaching programming language syntax; 7) students implementing the detailed design into code; and 8) students testing the program. Through a trial semester, they found that the results from students who received the experimental method were skewed towards the higher grade (e.g. A or B), indicating students’ retention of acquired knowledge and their ability to apply that knowledge. They also found that the results from students who received traditional method were skewed towards the lower-end grade (e.g. D or F).

Although problem-solving strategy was advocated to be a critical part of novice programming in the previous study, there was no consensus about teaching problem-solving in introductory programming. A survey of 85 programming courses from 39 universities in Australia and 8
universities in New Zealand in 2003 (de Raadt, Watson, et al., 2004) revealed that instruction about problem-solving strategy varied greatly in the courses. Some courses did not include problem-solving strategy because the instructors believed that the problems were not big enough to explicitly fit the strategy. A subsequent survey (Mason et al., 2012) showed that the percentage of class time spent on problem solving remained similar from 2003 to 2010 in the universities in Australasia. The survey indicated that problem-solving strategy had been either moved into a separate course or taught implicitly rather than explicitly.

In a literature study, Robins et al. (2003) concluded that there is a close relationship between programming strategy and programming knowledge in pedagogy. They proposed that programming strategy includes how programming knowledge is applied to solve a problem. In this point of view, we agree with their argument that teaching programming strategy is more important than programming knowledge. We also see that it is possible to teach novices programming strategies.

In summary, although various strategies were proposed, problem-solving strategy has been widely applied in programming. In particular, it is promising to use expert’s problem-solving strategy in teaching. However, the problem-solving strategy is too general for novices to apply to their practice. It only indicates the general direction of programming. There is no more detailed guidance to each phase of the problem-solving process. Before we go to further study about experts’ knowledge and strategy, we turn to understand what a programming process is.

2.5.3 Programming Process

Problem-solving indicates a general strategy (or direction) for novice programmers to develop a program. In contrast, a programming process provides a detailed step-by-step guidance as scaffolding to support novices’ development of programs. Based on the arguments for and against minimal guidance during instruction proposed by Kirschner, Sweller, and Clark (2006) in the Educational Psychology community, Guzdial (2009) was concerned that teaching novices programming by having them programming with minimal guidance was ineffective.

Initially, Kirschner et al. (2006) proposed that “Direct instructional guidance is defined as providing information that fully explains the concepts and procedures that students are
required to learn as well as learning strategy that is compatible with human cognitive architecture” (p75). For teaching novices, they advocated using worked examples as rich guidance for the solution of problem solving rather than minimal guidance based on the discovery of learning. They also advocated using process worksheets as a way of guiding instruction. They argued that:

“Such worksheets provide a description of the phases one should go through when solving the problem as well as hints or rules of thumb that may help to successfully complete each phase. Students can consult the process worksheet while they are working on the learning tasks and they may use it to note intermediate results of the problem-solving process” (Kirchner et al., 2006, p80).

Next, a debate about teaching novices with minimal guidance arose. Hmelo-Silver, Duncan, and Chinn (2007) argued that both problem-based learning and inquiry-based learning were not minimally guided approaches because both approaches included scaffolding and guidance. They also argued that scaffolding embedded expert information and guidance.

Finally, the debate had a consensus that it is ineffective to teach either problem-based learning or inquiry-based learning without scaffolding (Sweller, Kirschner, & Clark, 2007). Sweller et al. (2007) argued that “the only scaffolds they seem to ignore are providing learners with a problem and a problem-solving procedure that can be used for generating this solution. In other words, both a fully worked out example of a solution (i.e., task support) and the process-related information used to reach the solution is necessary for the design of suitable learning tasks and the associated instructional support and guidance structures” (p117).

We now turn to explore the relevant issues about using strategies, particularly considering for and against minimal guidance and guided methods in computer education. In order to overcome the difficulties in OO programming, Kay et al. (2000) had adopted the problem-based learning (PBL) approach. PBL is “a pedagogy that centers student learning around open-ended, student-driven problems facilitated by an instructor in order to achieve the learning outcomes of a course” (Fee & Holland-Minkley, 2010, p129). The key ideas are: authentic large projects, self-paced group work, replacing lectures with tutorials and labs, and focusing on generic and metacognitive skills in terms of critical thinking, planning, problem solving, research skills, and communication skills. Kay et al. (2000) showed that PBL yielded a
significant improvement in programming competence, but it also required substantial effort to introduce. The negative aspects of PBL are a lack of structure, guidelines, and feedback.

Following the method of minimal guidance, Ben-Ari (1998) introduced constructivism from learning theory into computer science education and believed that students can actively construct knowledge rather than passively receive and remember knowledge from textbooks and teachers. He argued that there is a “perception that computer science is ‘hard’ ... due to the fact that models must be self-constructed from the ground up” (p259). He suggested having group assignments and controlled labs instead of individual homework exercises. He proposed that these kinds of social interaction enabled students to construct a good model of computer.

Lister (2011b) proposed applying another type of constructivist theory, neo-Piagetian, in the teaching of novices programming. Inspired by the study of children’s constructivist education by Piaget (1970), neo-Piagetian theory holds that “people, regardless their age, are thought to progress through increasingly abstract forms of reasoning as they gain expertise in a specific problem domain” (p10). Lister claimed that novices’ development of programming included three reasoning stages from less sophisticated reasoning to formal operational reasoning about writing programs. He defined the first stage as preoperational reasoning. At this stage, novices could trace a program and explain code details without abstract meaning (e.g. programming concepts). They struggled in using diagrams effectively. The second stage was defined as concrete operational reasoning. Novices could only reason in a specific program context. The third stage was called formal operational reasoning. At this stage, novices wrote programs by following problem-solving strategies as discussed above. He argued that we could not expect that novices immediately start programming by jumping directly to the last stage because novices’ working memory was easily overloaded due to the lack of knowledge structures such as a “chunk” of program code (see Section 2.5.4 for a discussion of “chunks” of code and see Chapter 7 for working memory).

Through historical review of constructivist teaching methods in three decades from 1960s to 1980s, Mayer (2004) argued that the guided discovery method was more effective than the pure discovery method in teaching. The pure discovery method in teaching promotes learners to be highly behaviourally active through searching, analysing, developing, and evaluating rather promoting hands-on activities. While the pure discovery method gives students maximal freedom to explore, the guided discovery method provides students systematically
guidance on their learning objective. Based on the literature, he claimed that guided discovery helps students to learn and transfer problem-solving rules to new problems in the literature in 1960s. He found that the guided discovery method required the most learning time, but performed best on solving transferring problems among the pure discovery method, guided discovery method, and expository method (providing both question and answer). He also claimed that minimal level of guidance by using corrective feedback (e.g. simply comments of yes or no) helped students’ learning better than self-discovery of conservation strategies in the literature in 1970s. The conservation concept was based on Piaget’s (1970) classic vision of discovery learning in children’s ability of logical thinking. According to Piaget’s theory, conservation refers that when applying self-discovery method without teachers correcting children’s ability to determine a certain quantity of material (e.g. liquid) remains the same despite changing its appearance (e.g. under different container shapes). Mayer claimed that pure discovery failed to help students’ learning of programming concepts by using the Logo programming environment in the literature in 1980s. From the literature, he found that students who were taught by a guided discovery method, specifically proving a worksheet with a basic programming and debugging process, performed better than those without such a guidance.

The advantage of providing students with a programming process had been found in the study of guided discovery in the literature. However, “Fewer studies have focussed in on the process of programming, per se” (Redmond & Gasen, 1989, p697). Although a teaching programming process has been advocated, it still has not been widely applied in the introductory programming curriculum (Caspersen & Kölling, 2009).

Redmond and Gasen (1989) described programming as a flow program (see Figure 2-3). They emphasised feedback at the evaluation points by testing the loop backs indicated with dashed lines in Figure 2-3 during the programming process. They also claimed that “data about changes in code will increase our knowledge concerning the nature of how people think and develop their programs during the programming process” (p700).

In summary, although problem-solving strategies are important in teaching novices programming, it is critical to find out how to effectively teach these strategies to novice programmers. The teaching strategies have been focused on the way of using either guided
learning, or unguided (or minimal) learning. A comparison of the two strategies in teaching novices programming is shown in Table 2-3.

![Diagram of programming process]

**Figure 2-3 Schematic of the programming process, adapted from Redmond and Gasen (1989)**

<table>
<thead>
<tr>
<th>Strategies</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimal or unguided learning (PBL) (Kay et al. 2000)</td>
<td>• significant improvement in programming</td>
<td>• requires substantial effort to introduce;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• lack of structure, guideline, and feedback</td>
</tr>
<tr>
<td>Guided learning</td>
<td>• effective</td>
<td>• need feedback</td>
</tr>
<tr>
<td>Programming Process (Redmond &amp; Gasen 1989; Mayer 2004; Caspersen &amp; Kölling 2009)</td>
<td>• better performance than without guidance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• engagement</td>
<td></td>
</tr>
</tbody>
</table>

Some of the constructivist approaches such as PBL could improve programming learning, but they need significant effort. Other constructivist approach, e.g. neo-Piagetian, could improve learning, but need support by using “chunks” of knowledge advocated in the literature. On the other hand, the guided learning methods such as providing a programming process seem effective and promising for supporting problem-solving strategies. We now turn to explore the idea of “chunks” of knowledge from experts in next section.

### 2.5.4 Programming Structures, Plans, and Related Tools

Teaching novices programming is usually based on examples in a particular computer language together with the explanation of programming concepts. This approach requires students to solve problems based on their knowledge of language and concepts. In this general teaching model, three types of programming knowledge (syntactic, conceptual, and strategic) were proposed (Bayman & Mayer, 1988). Bayman and Mayer (1988) argued that this conceptual
teaching model enhanced mental models and problem-solving performance, especially for weak students. They also argued that students’ problem-solving performance is closely associated with their conceptual knowledge. However, this model neither provides a detailed process for using the knowledge, nor takes advantage of knowledge possessed by experts. We now turn to: 1) finding experts’ knowledge of programming; 2) understanding of how experts work to teach novices; and 3) exploring how to apply in practice.

1. Various Formats of Experts’ Knowledge
The concepts of goals and plans were introduced by Soloway and his colleagues (Letovsky & Soloway, 1986; Soloway & Ehrlich, 1984; Spohrer et al., 1985). A goal is a certain objective that a program must achieve in order to solve a problem (Letovsky & Soloway, 1986), and a plan (Spohrer et al., 1985) corresponds to a fragment of code that performs actions to achieve a goal. Letovsky and Soloway (1986) proposed “the term goal to denote intentions and the term plan to denote techniques for realizing intentions” (p41). Soloway (1986) advocated that:

“There is a lot of knowledge and strategies that experts use that need to be made explicit and taught explicitly to students in introductory programming courses. There are two general categories of concepts: knowledge and strategies for using knowledge” (Soloway, 1986, p851).

He argued that goals and plans are the basic building blocks for analysing problems and generating programs.

The notion and construction of a programming plan are the key elements of comprehension and generation of programs. For the notion of a programming plan, Soloway and his colleagues (Soloway et al., 1988; Soloway & Ehrlich, 1984) claimed that it directly corresponds to the notion of a schema: a generic knowledge structure for interpretation, inference, expectation, and attention on text comprehension in artificial intelligence and psychology. In order to construct plans, Rist (1989) proposed a model of plan generation beginning with a focus on a single line of code, moving on to fragments consisting of lines of code, and then to a complete plan by code fragments. The plans can be then merged to produce a full program.

Soloway and his colleagues (Soloway et al., 1988; Soloway & Ehrlich, 1984) suggested that expert programmers have at least two types of programming knowledge: 1) programming plans, and 2) rules of programming discourse. They argued that the plans were used to construct the program for a specific problem and the rules of programming discourse were
applied to guide the composition of those plans into a program. For the rules of programming discourse, Joni and Soloway (1986) claimed that there are seven discourse rules for novice programmers, for example, the discourse rule for initialisations: having your program initialise each variable precisely once.

Soloway (1986) proposed four general strategies for the composition of plans into a program. The strategies were: 1) abutment (placing two plans in sequence), 2) nesting (embedding one plan inside another), 3) merging (combining plans by interleaving them), and 4) tailoring (modifying plans to fit the needs of the problem).

Moreover, based on the study of plan composition and the relationship of plans, Ebrahimi (1994) argued that “programming errors are related to the mismanagement of Plans and the programming knowledge being used” (p460). The four major categories of errors were attributed to: 1) missing plans, 2) misplaced plans, 3) malformed plans, and 4) misused irrelevant plans.

When using plan knowledge for program comprehension, Letovsky and Soloway (1986) advocated that “Delocalized plans could be defined as plans with data flow links spanning widely separate parts of the code; data flow analyzers, by making such link explicit, could be very useful in countering the comprehensibility problems associated with these plans” (p47). In order to prevent novices’ comprehension errors for programming plans, they aimed to provide the program reader with information about the code and a convenient way to access the information. Therefore, they proposed two strategies: 1) document goals and roles for variables; and 2) document by in-line comments.

Regarding the general constructs of program from goals and plans, Rist (1986) proposed that:

“A program may be considered as a plan tree that relates problem plans to dominant goals. Top-down program design would view a program as growing from the top description node in the tree down to the individual leaves. In such a tree, the program goal represents the highest node. Below this are the standard global plans, such as input, process and output. At each level, goals are split into plans and the process continues until the plans at the lowest level can be translated into code” (Rist, 1986, p29).
Using goals and plans, the programming process was proposed to start from constructing a goal chain (Rist, 1986). The goal back-chaining method works backwards from the required final goal to be achieved for the problem solution to find out the prerequisite goals that are needed to realise the final goal. Similarly, starting with these prerequisite goals, back-chaining is used to find out their prerequisite goals to produce the provided information. For example, this method starts from a goal for output works backwards to its goals for processing and then backwards to its goals for input. With this method, a goal chain is constructed in terms of goals and their connections. After the analysis of the goal chain, program development is a process of top-down design and implementation using existing code from a library (Rist, 1986):

“The first version of a new program, a new problem, is written by a process of goal back-chaining. The program goal creates one or more subgoals which in turn create other and more detailed goals. Once learned, this goal chain can be isolated and retrieved from a plan library for use in writing later versions by top-down design” (p30).

Regarding the change from a bottom-up development model to using goals and plans, Rist (1989) argued that “if knowledge can be found to guide program design, top-down and forward design will be seen” (p410). This argument suggested that when the plan is provided in a plan library, novices can develop their solution of goals in a forward and top-down model. He divided the structure of a program into four levels, comprising: 1) a single line of code from pieces of knowledge including symbols, variables, and operators; 2) a simple plan combined from lines of code; 3) a complex plan from merged simple plans to achieve the goal of a problem; and 4) a program from merged complex plans. He argued that although the complexity increases in the development from the first to the last level, novices can retrieve the chunks of knowledge at each level instead of creating them. “As expertise develops, detailed planning at one level disappears and is replaced by retrieval and planning at the next level of design” (Rist, 1989, p398).

Furthermore, Soloway (1986) argued that:

“Expert programmers know a great deal more than just the syntax and semantics of language constructs … They have built up large libraries of stereotypical solutions to problems as well as strategies for coordinating and composing them. Students should be taught explicitly about these libraries and strategies for using them” (p850).

He proposed four strategies for constructing and explaining program from goals and plans:

1) Stepwise refinement, which breaks a problem into several smaller sub-problems.
“Break down a problem into sub-problems, on the basis of problems that you have already solved and for which you canned (or almost canned) solutions....Break the new problem down so that you can use those canned solutions...Canned solutions are programming plans—stereotypical methods for achieving goals” (Soloway, 1986, p855).

This strategy provides a guideline of decomposing a goal into sub-goals. The sub-goal decomposition can stop when it matches a relevant plan in the plan library.

2) Plan composition methods, which are applied for gluing plans together (see Chapter 4).

3) Rules of programming discourse, which direct how plans should be realised.

4) Simulation, which provides feedback and enables novices to rework their designs in order to develop an effective design. Soloway advocated that “the simulation strategy should be explicitly taught to students” (Soloway, 1986, p858).

Recently, De Raadt et al. (2009) advocated teaching and assessing programming strategies explicitly rather than implicitly using goals and plans that experts have in programming practice. They described a plan as a small, independent strategy that has been used in previous solution. They used the description of plans as strategies together with examples or diagrams. They successfully integrated Soloway’s programming knowledge in the traditional curriculum and developed a strategy guide consisting of 18 strategies. They argued that it was possible to teach and assess programming strategies in the curriculum with such a new vocabulary. They also claimed that students learned and applied programming strategies more frequently when these were explicitly taught.

A similar idea, using the term “template” was proposed by Linn (1985):

“Templates are stereotypic patterns of code that use more than a single language feature. Templates perform complex functions such as sorting names alphabetically or summing a set of numbers. Templates can be used each time a given task is encountered. A large repertoire of templates enables the programmer to solve problems without creating new code. Well chosen of templates facilitate good programming” (p15).

A schema-based approach aligned to control flow was proposed by Détienne (1990) for program understanding. She claimed that a schema (or plan) is represented as a knowledge packet that has variables. Consequently, understanding a program can be started by evoking
schemas (or program plans) stored in memory and instantiating the schemas with values.

Similar to the goals and plans approach proposed by Soloway and his colleagues, a program can be represented hierarchically using decomposed subgoals. Détienne advocated using mental simulation of this hierarchical representation so that the programmer can infer the goal of the process based on the intermediate values of variables from mental execution. Although the schema-based approach supports the control flow based approach, she argued that “plans formalise information on data flow and functions whereas syntactic constructs reflect more the structure of the program as described with control-flow reflection” (Détienne, 1990, p219).

Although previous studies of plans and templates investigated providing novices with chunks of programming knowledge, further studies discovered that novices still had difficulties in mapping the abstract plans into program code (Rist, 1989; Wallingford, 1996). Instead of teaching novices abstract plans, computer language, and how to convert between them, Wallingford (1996) proposed teaching novices programming patterns by providing the problem, the solution, and the implementation details. He also proposed using programming patterns as a methodology for the entire introductory course and providing a complete set of programming patterns.

“A pattern approach offers the novice programmer a new kind of tool. Rather than viewing programming language statements as the building blocks out of which to construct programs, students can be taught to use patterns as the basic unit of analysis, design, and programming. These patterns provide a mapping from a type of problem to an effective algorithm and an effective implementation in code. In this way, small piecework that the students would otherwise have to continually redo is standardized into a larger unit that can be reused in various contexts” (Wallingford 1996, p28).

Further studies considered how to use programming patterns in the teaching process. Porter and Calder (2003) proposed the process of solving a program by using patterns for novice programmers. The process consisted of: 1) detecting unsolved problem parts in order to add new patterns; 2) finding the pattern from a provided pattern language diagram (a chart of the context for the next pattern to be applied); 3) applying the pattern details to the program; and
4) repeating the previous three steps to apply them to the next unsolved part of the problem.

In summary, several approaches have attempted to use the chunks of knowledge that experts possess to improve teaching novices programming. The concepts of programming plan, templates, schemas and patterns were used variously (Détienne, 1990; Linn & Dalbey, 1985; Porter & Calder, 2003; Soloway et al., 1988; Soloway & Ehrlich, 1984; Wallingford, 1996). A summary of these approaches contrasted with the conventional approach in the domain of novices attributes that Robins et al. (2003) classified is in Table 2-4.

Table 2-4 Summary of the experts’ knowledge chunk approach

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Main Author</th>
<th>Knowledge</th>
<th>Strategies</th>
<th>Mental Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Experts’ Chunk Knowledge</td>
<td>Soloway (1986)</td>
<td>Goals and plans</td>
<td>Rules of programming discourse, strategies of plan composition</td>
<td>Simulation (mental execution)</td>
</tr>
<tr>
<td>Approach</td>
<td>Linn (1986)</td>
<td>Templates</td>
<td>Procedural skills</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Detienne (1990)</td>
<td>Schemas</td>
<td>N/A</td>
<td>Simulation (mental execution)</td>
</tr>
<tr>
<td></td>
<td>Wallingford (1996),</td>
<td>Patterns</td>
<td>Process</td>
<td>N/A</td>
</tr>
</tbody>
</table>

If using the structure of experts’ knowledge to teach novices programming is better than the conventional method, then the next critical issue is how to use that knowledge. Gilmore (1990) claimed that “it seems that possessing knowledge is not the only problem that novice programmers have. In a number of cases they show that they have the knowledge, but also that they do not know how to adequately use it” (p232). In other words, providing novices with support and detailed guidance is a critical issue in the development of an effective teaching approach to improve processes for teaching novices’ programming. Although previous studies had proposed strategies and process for using expert’s knowledge (Soloway, 1986; Linn, 1986; Porter & Calder, 2003), there was no detailed programming process of constructing a program starting from the problem analysis to goals and plans and then to the final program. Before we proceed to development of the process, we turn to explore the mental representations of a program in the way presented by experts.

2. Representation of Experts’ Programming Knowledge

Based on an empirical study of verbal protocol data from professional programmers, Letovsky (1987) proposed a knowledge-based cognitive model of the program understanding processes. The model includes a set of knowledge assimilation processes for combining information from
program code and documentation from a knowledge base. The knowledge base conglomerated results from previous studies of conventional programming knowledge and experts' knowledge, including programming language semantics, goals, plans, efficiency knowledge (detecting inefficiencies and evaluating costs), domain knowledge, and discourse rules. Therefore, he concluded that a complete mental model of a target program can be then constructed by the combination of collected information from the knowledge base. The components consist of: 1) the specification (description of goals), 2) annotation (explanation of implementation of each goal), and 3) implementation (description of actions and data structures). Finally, he advocated that the assimilation process plays the most important role in constructing the mental model.

Based on investigating the role of programming knowledge in programming comprehension and mental presentation, Pennington (1987) backed up Rist's (1986) back-chain between goals using dataflow in an opposite direction to the back-chain. She proposed to construct the mental representation at the “macrostructure” level (or abstract level), focusing on two types of programming knowledge: text structure knowledge (procedural relations or control flow structure consisting of sequence, iteration, and selection) and plan knowledge (functional relations or patterns of program segments). Through studies of programming comprehension by professional programmers, she concluded that control or procedural relations dominated the mental representation. She argued that a functional representation is not constructed as fast as a control or procedural representation and needs extensive involvement and explanation. Regarding mental representation of plan knowledge, Pennington concluded:

“Plan representations of a program are primarily based on data flow relations. This is because much of the control structure in a program that is mandated by data flow requirements is arbitrary ... in terms of the multiple abstractions of programs, data flow and function information should be readily available; sequence and detail operations should be less accessible” (p308).

The role of both language notation and knowledge representation in the determination of programming strategy was studied by Davies (1991). Based on a study of plan generation using different computer languages by novices, intermediates and experts, he argued that features of language notation could assist novices using programming strategies during the beginning stages of their development. However, he also claimed that “As [novices'] programming skill increases, the role of notation appears to take less precedence as a determinant of strategy”
He advocated that the role of both language notation and knowledge representation had to be considered together as the determinants of programming strategies. We consider that this result could be the reason why previous studies of using visual programming languages or simplified (or subset) languages showed benefits only for novices.

Furthermore, Wiedenbeck, Fix, and Scholtz (1993) proposed that expert’s mental presentation of a program includes five abstract characteristics: 1) hierarchical multi-layered structure, 2) explicit mappings, 3) incorporation of basic recurring patterns, 4) well-connected representation, and 5) a foundation in the program text.

Firstly, hierarchical structure had already been widely used in top-down decomposition and step-wise refinement in program design. It was also applied to the decomposition of a goal into sub-goals hierarchically. Secondly, they argued that experts and novices had almost no difference in their understanding of goals and sub-goals. However, experts had a stronger ability than novices in mapping between sub-goals and segment code. In other words, novices need detailed guidance to map sub-goals to the relevant plan code. Thirdly, they advocated the development of skills for recognising and labelling recurring patterns. Fourthly, regarding to understanding the interaction between one plan and another, they argued that dataflow represented the major connections between plans. However, novices had a poor understanding of data connections. That is to say, there is a need for notation to represent goals and plans in terms of data-flow connections. Fifthly, they argued that there were differences between experts and novices in locating various program units in the program, which referred to the understanding of how the program unit corresponded to program text.

Guindon (1990) proposed that:

“A library of reusable software design schemas, which could easily be retrieved by giving high-level descriptions of the problems may help bridge the gap between novices and experts ... The library of reusable design schemas could also be used as a training tool” (p301).

That is to say, a plan library could be used to directly support our teaching of programming development.

Wiedenbeck et al. (1993) proposed that:
“These differences in the mental representation may provide a partial explanation of why novice performance is poorer than expert performance on tasks ... When a programmer exercises these skills, a good representation which supports comprehension-related programming tasks is likely to emerge... However, we do expect that maintenance programmers who work with a large program over a period of time eventually develop a multi-faced representation similar to what we found” (p808).

In summary, teaching novices programming by taking advantage of expert knowledge in terms of goals and plans is promising. However, in order to match these experts’ mental representations, five key areas should be considered (see Table 2-5).

Table 2-5 Key areas of our proposed approach based on experts’ mental representation

<table>
<thead>
<tr>
<th>Experts’ mental representation</th>
<th>Goal of Proposed Development</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) hierarchical multi-layered structure</td>
<td>how to organise goals;</td>
</tr>
<tr>
<td>2) explicit mappings</td>
<td>how to map from goal to final program;</td>
</tr>
<tr>
<td>3) incorporation of basic recurring patterns</td>
<td>how to represent goals and plans;</td>
</tr>
<tr>
<td>4) well-connectedness</td>
<td>how to connect plans and to be associated to control flow connected program code;</td>
</tr>
<tr>
<td>5) foundation in the program text</td>
<td>how to support program implementation by using plans</td>
</tr>
</tbody>
</table>

3. Tools for Using Experts’ Approaches

Several tools have been developed to support teaching novices programming. PROUST was a tool that particularly used expert’s knowledge, i.e. programming plans, to analyse and understand buggy programs developed by novices (Johnson & Soloway, 1985). In PROUST, a goal is decomposed into sub-goals, which are organised hierarchically. A mapping from sub-goals to plans implements the program. Johnson and Soloway (1985) claimed that “Buggy programs are either derived from incorrect goal decompositions or from incorrect implementations [or mapping] of correct decompositions” (p269). They proposed that there was an issue of choosing from several alternative plans to achieve the same goal. They also discovered that the organising correct plans might still result in bugs. Therefore, they advocated developing a way of relating plans to other plans using goal decomposition. They also advocated making the programming process explicit by goal decomposition from problem specification to plans and then to code.

The most significant tool that used goals and plans was Bridge (Bonar & Cunningham, 1988) and its visual programming component, BridgeTalk (Bonar & Liffick 1990). Bonar and Liffick (1990) argued that novices’ syntactic strategies were similar to those of novices who studied
physics by matching knowns and unknowns against formulas, whereas experts utilised intermediate concepts and techniques from past experience. The *Bridge* was designed to bridge the gap between novices and experts, particularly in the conceptual distance between programming language (syntax and semantics) and the goals or purpose implemented by the program. “The key principle of Bridge is: Teaching elementary programming with intermediate representations provides novice programmers with a deeper understanding of the programming language and process than[sic] is possible with conventional approaches” (Bonar & Cunningham, 1988, p14).

In order to make use of intermediate representations, *Bridge* presented the user with three phases (Bonar & Cunningham, 1988). In phase I, simple natural language phrases (e.g. “read in numbers”, “count each integer”) were used to present a step-by-step informal statement of a problem solution. These English language phases were supported by selecting informal plans from defined menu items.

In phase II, a visual plan-based programming language, *BridgeTalk*, was used to present formal plan components from the informal plans identified in phase I and to express their interrelationships (Bonar & Cunningham, 1988). Although the early version of *BridgeTalk* had a visual representing of both control flow and dataflow, they decided to use control flow to present the relationships between plans (Bonar & Liffick, 1990). Considering the difficulties of representation of plans in several dispersed lines of code within a program, Bonar and Liffick (1990) maintained a plan as an atomic element without further support for mapping from the visual plan-based (or plan-like composition) program to programming language code.

In phase III, Bonar and his colleagues required students to transfer the visual plan structure from phase II into a computer language. They only provided pseudocode from individual plans without support for merging these plans.

Bonar and his colleagues had significantly contributed to the development of using expert’s knowledge of goals and plans with intermediate representations. They established an approach for combining the conceptual knowledge of goals and plans with a visual programming language. However, no further development was reported, such as mapping from the visual plan-based program into final program code. Perhaps the icon usage in the visual programming language was not popular in teaching novices programming when Bonar
and Liffick (1990) had such a pilot exploration. Or maybe when mapping from these visual plans to a programming language, it was too hard to overcome the buggy programs when merging plans.

The further study of mapping from plans to programming language code was based on text-based representation of plans in menu items rather than from visual plans. Guzdial et al. (1998) developed GPCeditor (Goal-Plan-Code editor) to support decomposition of goals and composition of plans a programming language code. The tool was designed based on the idea of CAD (computer-aided design) to provide scaffolding for using goals and plans by choosing from menu items. The menu items provided mappings from goal to plans and then from plans to plan code details. However, for program generation by plan composition, GPCeditor only implemented three of four Soloway’s strategies for gluing plans: abutt (abutment), nest (nesting), and cut (tailoring). They avoided merging plans by interleaving where parts from different plans were blended. The significant contributions of this tool are that it provided a plan library to support decomposition of goals and integrated plan decomposition and composition. It also provided a programming process through the order of the programming activities, i.e. goal-plan-code. They proposed that the scaffolding by using GPCeditor had a feature of fading, which refers to the concepts used in the traditional apprenticeship (Collins, Brown, & Holum, 1991).

“Scaffolding is the support the master gives apprentices in carrying out a task. ... Fading is the notion of slowly removing the support, giving the apprentice more and more responsibility” (Collins et al., 1991, p2).

Finally, they concluded that “the GPCeditor was well-suited to a student with little or no programming experience, but, after a few months of using it, it no longer met their needs” (Guzdial et al., 1998, p176).

Similarly, ADAPT (Ada Packages Tool) was designed mainly to support novices in program development from planning to code following a hierarchical top-down development approach (Fix & Wiedenbeck, 1996). In ADAPT, a program was developed on several levels by selecting menu items at each level. The highest level is a strategic plan (e.g. Input Plan, with the function of inputting data). The next level is a tactical plan with more details, but still language-independent (e.g. Message “enter a number”, Input a number). When selecting from the menu, the higher level plan was replaced by the details until all the language-independent items were replaced by Ada code from templates. Fix and Wiedenbeck (1996) claimed that this
multi-level plan replacement is a top-down approach from high-level plan using pseudocode to the implementation using programming code. Although the replacement of higher level plans can be in any order, they advocated that it is useful to provide immediate feedback during the process. They also suggested that “the language describing plans was the weakest point of the prototype” (p82). In other words, using a visual notation for plans as in BridgeTalk would be better than the text-based plans.

Furthermore, many hypermedia tools were developed as scaffolding to support mapping and organising of plans (or temples) (Linn, 1992). Linn (1992) argued that:

“a [teaching] model ensures that students are exposed to appropriate goals for knowledge building, but do not guarantee that students will follow the model in their own knowledge organisation. Scaffolding increases the completeness of the knowledge that is organised but does not guarantee that students will organise their knowledge according to the ideal model” (p137).

She reported five different tools by hypermedia links to provide support for retrieving templates from a library as well as for learning (or understanding) individual templates. For example, one tool (List Template Library) organises templates through a hierarchical chart. Another tool (HyperComments) prompts for information to support the selection of a template.

de Barros et al. (2005) developed a programming learning environment using pedagogical patterns, ProPAT. It was built as an Eclipse plug-in to teach Java. Students could select and add provided pedagogical programming patterns into the editor to construct a solution. It seems that ProPAT is similar to previous tools (such as GPCeditor, ADAPT) so that the user can start programming from provided code segments rather than from the scratch.

In summary, most tools surveyed in this section support mapping from goals to plans (see Table 2-6). However, not all of them support mapping from plans to programming code. Only GPC partially supported plan code composition (except for interleaving). Although the rest of the tools support retrieving individual plan details in programming code or pseudocode, there is no support for how to merge these plan details. One tool (BridgeTalk) had a visual programming language at the plan level rather than at programming code level. Other tools (GPCeditor and ProCAT) had a simple programming process rather than a detailed process.
Table 2-6 Summary of plan tools

<table>
<thead>
<tr>
<th>Tools</th>
<th>Mapping from Goals to Plans</th>
<th>Mapping from Plans to Language Code</th>
<th>Visual Programming Language</th>
<th>Programming Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bridge &amp; BridgeTalk (Bonar &amp; Cunningham, 1988; Bonar &amp; Liffick, 1990)</td>
<td>Using informal plans</td>
<td>N/A</td>
<td>Using a visual plan-based language</td>
<td>N/A</td>
</tr>
<tr>
<td>GPCeditor (Guzdial et al. 1998)</td>
<td>Using menu items</td>
<td>Partial support except interleaving</td>
<td>N/A</td>
<td>weak</td>
</tr>
<tr>
<td>ADAPT (Fix &amp; Wiedenbeck, 1996)</td>
<td>Using hierarchical menu items</td>
<td>Support for retrieving plan code, but not for plan compositions</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Hypermedia tools (Linn, 1992)</td>
<td>N/A</td>
<td>Support for organising and retrieving plan code, but not for plan compositions</td>
<td>Hypermedia links rather than a visual language</td>
<td>N/A</td>
</tr>
<tr>
<td>ProPAT (de Barros et al., 2005)</td>
<td>N/A</td>
<td>Support for identifying pattern code with a general composition process</td>
<td>N/A</td>
<td>weak</td>
</tr>
</tbody>
</table>

2.6 Teaching Approaches

Many teaching approaches have been proposed to support novices’ learning of programming. These approaches vary from using a full language to a reduced size language, to beyond computer execution (e.g. manual execution, or mental execution), to beyond the focus on computer language (e.g. using pseudocode or symbolic language).

Brusilovsky et al. (1994) had analysed three different approaches in teaching novices programming, which consisted of: 1) the incremental approach, 2) the sub-language approach, and 3) the mini-language approach. In the incremental approach, a programming language is divided into a sequence of subsets. The whole programming language is taught progressively from one concentrated subset to another. Instead of teaching the whole programming language, the mini-language approach is to use a small and simple language to support novices’ learning. Brusilovsky claimed that Logo and Karel the Robot were examples of mini-languages. Students use a small set of commands to make a program in order to control an actor or object, such as turtle, a robot, a cherry or any other object to move in a microworld.

Although the "mini-language" makes teaching and learning simple and attractive, it limits the learning of problem-solving to a synthetic environment and special purpose language rather than a real programming language. The sub-language approach aims to combine the advantages of both previous approaches. It includes a subset of features from a real programming language. They argued that this approach supports the first stage of novices’ learning and helps to prepare them for further study. Finally, they advocated that teaching...
novices programming should start with using a small, simple language subset together with visualisation. Programming concepts should be embedded with problem-solving. Similar ideas were adopted by Hu (2004) to combine a small language subset and visualisation in the teaching practice.

Our focus on teaching approaches now turns from the role of language size to programming pedagogy. Fincher (1999) argued that traditional teaching approaches rely too much on the execution of programs rather than on the rationale for using these approaches. She explored four proposed approaches that were based on conceptual models and methodologies for teaching programming. They were: 1) the “syntax–free” approach, 2) the “literacy” approach, 3) the “problem-solving” approach, and 4) computation as interaction.

Firstly, in the “syntax-free” approach, pseudocode (plain English) and pencil-and-paper are used for students to prepare their knowledge and skills for real language programming. It was argued that this approach reduced students’ load from learning the programming language at the beginning in order to focus on the design of the solution.

Secondly, in the “literacy” approach, real language programs were used for learning through reading and comprehension rather than by writing. It was claimed that this approach was regarded as a sort of apprenticeship model. Students read incrementally from simple to more complex real-world examples, and then studied to extend the programs. However, there was concern that this approach might take a long time for some students to understand the programs. Other students might fail to find their own way to explain the examples and to write new programs.

Thirdly, the “problem-solving” approach (cf. Section 2.5.2) was concerned about its pedagogical effect and advocated a pedagogically-based approach to programming, comprising a problem-solving cycle of different phases: understand (analysis), design, write (implement), and review (test and debug), which can also be applied in various domains.

Fincher argued that although some of the approaches might be used either concurrently or sequentially, the common idea of these four approaches was the separation of coding from programming. She also claimed that “these approaches start from a position of identifying the
acquisition of other skills [e.g. analysis, design, problem-solving] as the ultimate objects and support student learning directly to this end [i.e. coding]” (p5).

Considering the ordering of teaching topics, Pattis (1993) argued that novices needed to learn programming language features first rather than design skills first. He suggested an approach for learning at an early stage by reading, modifying, and writing programs designed by the teacher.

Regarding the top-down design of procedural decomposition, Pattis argued that it is necessary to provide feedback on designs before students proceed with further development. However, at the time of his study, he claimed that only experts could mentally obtain the feedback without execution of the program because experts are more experienced in mental tracing. Therefore, for top-down design, instead he proposed using model programs, e.g. a simple model of Input-Process-Output as a divide-and-conquer variant of the above methods for the purpose of stepwise refinement. He also warned when introducing procedural decomposition, some important language features (e.g. difference of a variable representation between a storage location and a value; difference between abstract parameters and argument values) are critical because “students have an inadequate context for understanding them [those language features], or are not sophisticated enough to appreciate them. If forced to learn these features too early, students will incompletely understand them and avoid them” (Pattis, 1993, p125).

Interestingly, Pattis introduced a “natural” approach of teaching subprograms by using a subprogram library. In this approach, students firstly learn how to use subprograms from the library based on the subprogram interface comprising header and parameters. Next, they learn how to design their program by using different subprograms from the library. They then learn how to write their own subprograms. Finally, they can develop more complex programs. He argued that using subprogram headers according to names and comments can help students to understand them. The most significant advantage of this approach is that after understanding subprograms in a library, students can design complicated programs by using these prewritten subprograms in a stepwise-refinement way of iterative development rather than that in procedural development. The advantages of this approach are analogous to building up a skyscraper by using premade parts such as walls, doors, windows, or units rather than by using raw materials such as bricks, wood, and glass. However, there is no clear
guideline or detailed process to organise and design a program by using these subprograms. Also, there is no guideline for writing individual subprogram.

With the benefit of programmers working collaboratively, pair programming from industry had been applied to computer science education (Hanks, 2008; Salleh, Mendes, & Grundy, 2011; Williams et al., 2000; Williams & Kessler, 2001). Pair programming means that two people work collaboratively side-by-side by using one computer to develop one program under the problem-solving strategies described above. One person who directly operates the computer is called the “driver” or “operator”. The other who observes the operation and provides ideas is called the “navigator” or “observer”. The pair also exchanges their roles regularly. Hanks (2008) proposed studying whether there were the same types of problems that were faced in both pair programming and solo programming as well as whether there were the same frequency of problems met in both ways of programming. He used similar methods to those that Robins, Haden, and Garner (2006) had used, but he had pair programming for 30 students learning BlueJ rather than Robins’ solo programming for 470 students learning Java. He found that students with pair programming had the same types of problems as well as the same ratio of problem categories as those working alone. He concluded that “pair programming is not a panacea; students still need help and it is important to provide an environment in which that help can be given” (Hanks, 2008, p9).

In a systematic literature study of 73 papers about pair programming, Salleh, Mendes, and Grundy (2011) proposed that the advantages of pair programming were “improving design quality (fewer defects), team communication, and rapid solution to problems, enhancing the learning process, and increasing enjoyment in learning” (p509). They argued that since pair programming involves social interactions between both members, the effectiveness of students’ performance could be influenced by their compatibility. Based on a literature study, they identified 14 compatibility factors that might affect the effectiveness of pair programming. They concluded that the most important factors for compatibility were personality type, actual skill level, and perceived skill level. They also found that students would rather work with someone who had similar skills, abilities, and motivations.

In summary, most teaching approaches in the literature either focus on programming knowledge, e.g. using a sub-language or mini-language, or investigate programming strategies, e.g. using a problem-solving approach, divide-and-conquer and an incremental approach,
stepwise-refinement, a syntax-free approach, or a “literacy” approach. One approach that was shown to be promising by using a subprogram library for teaching subprograms. However, there is no clear guidelines for developing programs by subprograms. Recently, pair programming has shown its advantages in improving design quality, team communication, learning, and enjoyment. However, this approach needs additional effort and time. Particularly, this approach is largely affected by a pair’s compatibility. Now, we turn to discuss the mental representation of programming.

2.7 Mental Models of Programming

A mental model is described as cogitative structure to represent knowledge of a real-world artefact or phenomenon (Ben-Ari, 2001; Gentner & Stevens, 1983). It is proposed to be used for explanation and for prediction of behaviour. The mental model of the program as a notional machine has been widely applied in teaching novices programming (Mayer, 1981; Rist, 1986; du Boulay, 1986; Berry & Kölling, 2014). Milne and Rowe (2002) investigated the difficulties caused by computer language features and carried out an online survey on concepts and topics of the C++ language, surveying both students and teachers. They reported that the topics that students and teachers agreed were most difficult were related to understanding information in the computer memory, an issue related to understanding the notional machine. Therefore, they advocated to develop a visualisation tool to support students’ understanding and to help them to develop a model of the notional machine.

Similarly, Lahtinen, Ala-Mutka, and Järvinen (2005) developed a web-based visualisation to improve novices’ learning of programming concepts and language features. They conducted an international survey of more than 500 students and teachers about their background, course contents, and learning aspects. They concluded that the most difficult issues were how to design a program for problem-solving such as decomposition and debugging. They argued that the biggest problem for novices was to learn how to apply the basic concepts rather than to understand them. Consequently, they claimed that practical learning was more important than understanding the course contents because students needed practice to understand the concepts. Finally, Lahtinen et al. concluded that the difficulties in novices’ programming were attributed to the issues related to abstract programming concepts and program construction. Therefore, they advocated developing learning material and approaches to support the construction of novices’ knowledge and skills.
Recently, novices’ mental models of programming concepts, particular of variables, have also been brought into several investigations (Ma et al. 2011). Initially, Dehnadi and Bornat (2006) observed the mental models that novices used when reading assignment statements. Based on the mental models, they attempted to predict novices’ success in their final examination. They gave twelve questions similar to those on the exam to 61 students at week 3 of the course. Each question included several initialisation and variable-to-variable assignment statements for two or three variables (e.g. int a = 5; int b = 10; a = b;), ending up with a combination of diverse possible answers that represented a wide range of mental models. They reported that consistent use of similar models for almost all of the questions were associated with the passing the examination. They argued that the mental model test focused on variables in the early weeks can predict the success of the programming course.

Further, Ma et al. (2007) were intrigued by the results from Dehnadi and Bornat. They extended the range of programming concepts to object reference assignment (e.g. person a = new person (“John”); b = new person (“Mary”); a = b). At the end of a 20 week course, the results from 90 students supported “those obtained by Dehnadi and Bornat in that the consistent group performed significantly better than the inconsistent group. However, the results also show that the separation is not clean, particularly so in the case of the more advanced concept of reference assignment. Many participants in the inconsistent group still passed the examination” (p503).

However, Caspersen, Bennedsen, and Larsen (2007) found a lack of correlation between consistent mental model application and programming performance when investigating 142 students by using the original 12 questions used by Dehnadi and Bornat. They argued that the variation in performance could be attributed to the variances between two universities in terms of course contents, course structures, examinations, teachers, cohort of students, etc. They proposed that further investigation such as cross-institutional investigation was required.

Bornat, Dehnadi, and Simon (2008) expanded their investigation to more than 500 students at six institutes in three countries. Their results failed to find a significant correlation between consistency of the mental model and examination performance. However, they still believed that there would be an association between them.
Finally, Bornat (2014) issued a retraction for his original claims in 2006 about “camels and humps” in the prediction of students’ success or failure using a test for programming aptitude. He had earlier noted that the distribution of students’ results formed two peaks like camel’s humps: a large success hump and a small failure hump. In this paper, he withdrew the earlier claim that they could predict students’ success or failure based on an aptitude test. However, he still believed that Dehnadi had revealed evidence of an important phenomenon in novices’ programming which had been supported by several later papers.

Visualisation and animation of programming has also been applied to improve novices’ understanding (Ben-Ari, 2001; Naps et al., 2002). Ma et al. (2011) integrated programming visualisation into their previous mental model investigation. They argued that

“Program visualisation provides a potential solution to this problem by simulating how a programming concept operates through the use of graphics and animation, providing students with something close to a concrete model of program execution” (Ma et al., 2011, p63).

They concluded that using a visualisation-oriented learning environment supports novices to construct viable mental models.

Regarding the concept of notional machine introduced by du Boulay (1986) in introductory programming education, a recent literature study (Sorva, 2013) extended and amalgamated several threads of research in order to emphasise that this is an important aspect of knowledge for novice programmers. Sorva (2013) defined a mental model as “a mental structure that represents some aspect of one’s environment” (p8:7) and explained that mental models can offer simplified explanations to complex phenomena and allow the programmer to simulate the program mentally in order to predict the system’s behaviour. He defined a notional machine as an abstraction of computer hardware that is a mental model for understanding the executed program.

Sorva argued that misconceptions of the notional machine by novices were due to their deficient or inadequate understanding of programming practice, such as partial understandings, difficulties, and bugs. He explored the various efforts in the literature on discovery and reducing misconceptions, such as using plain English (Bayman & Mayer, 1983) and visualising the state of program execution (Sajaniemi, Kuittinen, & Tikansalo, 2008). Based on a literature study, he concluded that the misconceptions by novices were associated with
inadequate understanding of the notional machine and the implicit processes of the program code.

Sorva also used the quality of mental models based on the understanding of the notional machine to differentiate experts and novices as well as their abilities to mentally trace programs. Again, from a literature study, he claimed that experts had robust mental models while novice programmers’ mental representation of the notional machine were initially based on analogies from surface features of computer language, e.g. regarding an assignment statement as a mathematical equation. He emphasised that novices’ tracing of programs by mentally simulating program execution with the mental machine integrated the program and the notional machine.

Furthermore, Sorva connected three conceptual theories to emphasise the importance for novices to learn a notional machine. Firstly, according to the emphasis in constructivism on the learner’s prior knowledge, the knowledge of the real computer rather than the notional machine had been proposed as a prerequisite for novices to learn programming. Sorva claimed that the notional machine can be used instead as it is less abstract than a formal model (how things should work according to an assumption or standard) for understanding variables and computer, which is critical for novices to form.

Secondly, according to phenomenography theory, people experience the same phenomena differently depending on the person, time, and context. For example, an object in programming can be understood by students in an incremental development way with different categories such as a piece of code, an active entity during program execution, or a model (Eckerdal & Thuné, 2005). Sorva believed that novices’ learning of a notional machine could be supported by the incremental progress from program text to program runtime activity.

Thirdly, according to threshold concept theory, a threshold in education is defined as a troublesome barrier to students’ understanding (Meyer & Land, 2003). Sorva argued that a program run by a notional machine was considered as a dynamic execution-time entity or the program dynamics. In this point of view, program dynamics were regarded as a threshold concept that novices must cross. Finally, based on the literature, Sorva concluded that using conceptual models led to better performance than giving instructions in programming
language and that using a conceptual model of a notional machine could support teaching program tracing.

Moreover, Berry and Kölling (2014) proposed a graphical notation to represent the notional machine and developed a tool for novices to use this notation. Mental models not only support program development, but also help program comprehension. We now turn to briefly exploring how mental models support program comprehension.

Two literature studies summarised various theories in program comprehension, particularly by using mental models (Schulte et al., 2010; Storey, 2006). Storey (2006) defined a mental model as a mental presentation of a programmer’s program comprehension. Brooks (1983) advocated that program can be comprehended in a top-down, depth-first manner. He introduced the concept of “beacons” for recognising and identifying certain structures and operations, for example a section of code for interchanging elements in an array. Brooks proposed that the principles of program comprehension were as follows:

“1. The programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain.
2. Comprehending a program involves reconstructing part or all these mappings.
3. This reconstruction process is expectation driven by the creation, confirmation, and refinement of hypotheses” (Brooks, 1983, p544).

In summary, literature studies have shown that explicitly teaching mental representation of programming knowledge could support novices learning. A conceptual model of a notional machine can supply multiple benefits to novices in terms of reducing misconceptions, supporting program tracing, and comprehending programs.
2.8 Summary

In our literature study, we have explored many issues and difficulties in teaching and learning programming during the last three decades. One of the important issues is that novices face the poor design of computer languages and programming environments. Many difficulties that du Boulay (1986) identified are widely encountered by novices, such as lack of problem domain knowledge, misconceptions about the notional machine, misunderstanding notation of various formal languages, lack of acquiring standard structures, and weak pragmatics of programming.

Although many methodologies and tools have been proposed, the difficulties of novices learning programming were reported as remaining relatively unchanged over the years. The proposed solutions for reducing the difficulties can be summarised as: supporting learning, progressive problem-solving, providing scaffolding, motivation, feedback, program patterns, visual environments, program tracing, and revealing programming rules.

The differences between novices and experts were mainly that experts had chunks of knowledge, strategies, and abstract mental models. The reason why novices have difficulties in programming were explained by Perkins and his colleagues as being due to novices’ “fragile” knowledge. They proposed that changing the learning conditions could improve novices’ performance, e.g. providing a bridge course such as CS0.

The paradigms commonly used in introductory programming courses are still procedures-first, or objects-first. Although debate on their relative merits continues, the courses still emphasise writing and completing or modifying programs. A recent study has shown that some institutes still teach procedural programming and others moved to teach a mixture of paradigms by starting with procedural programming and then moving to object-oriented programming. Another study suggested that changing programming languages had no effect on novices’ pass rate.

In order to organise the research literature for our study, we defined a taxonomy of approaches for teaching novices programming. Our taxonomy consists of programming knowledge, programming strategies, and programming mental models. It is based on literature studies, Bloom’s taxonomy, and the revised Bloom’s taxonomy. It emphasises programming
strategy as many educators advocated. The literature that we explored is summarised in Figure 2-4 based on our taxonomy.

In the category of programming knowledge, we firstly explored modern programming languages and environments that were appropriate for teaching novices. This is mainly concerned with making it easy for novices to read and write programs and also with providing motivation (e.g. supporting games and multimedia) and immediate feedback.

The selection of a first programming language for novices was a trade-off between pedagogical considerations and industrial demands. Although Java and Python were identified as widely used programming languages, they do have weaknesses and issues (e.g. remembering syntax and debugging). On the other hand, visual programming environment, e.g. BYOB, an extended reimplementation of Scratch, simplifies the learning of programming by preventing syntax errors and providing immediate feedback. It is a promising introductory platform before transitioning to Java or Python. However, as Lister advocated, pedagogical consideration is a critical issue although visual programming environment can attract students at the beginning.

Under the category of programming knowledge, we also consider pragmatic skills in programming, particularly focusing on novice’s debugging skills. Program errors were classified as mainly programming syntax errors, semantic errors and logic errors. Debugging was suggested as one of the essential skills or a bottleneck in programming. Debugging requires similar knowledge as programming, such as application domain knowledge, knowledge of programming language, knowledge of program comprehension, programming process, and a mental model. It also needs knowledge of bugs and debugging. The strategies for teaching debugging included real-world scenarios, tracing on paper, adding print statements, pattern matching, using code stepping tools, using the Whyline tools, and using videos for studying debugging process.
1. **Programming Knowledge**

- **Language selection**
  - Fesakis and Serafeim, 2009; Gibbs and Coady 2010; Harvey and Möning, 2010; Kölling et al., 2003; Maloney et al., 2008; Maloney et al., 2010; Meerbau-Salant et al., 2013; Papadakis et al. 2014; Resnick et al., 2009.

- **VPE**
  - Ahmadzadeh et al., 2005; Bonar and Soloway, 1985; Ducasse and Emde, 1988; Fitzgerald et al., 2008; Fitzgerald et al., 2010; Gugerty & Olson, 1986; Johnson et al., 1983; Ko and Myers, 2004, 2008; McCauley et al., 2008; Murphy et al., 2008; Pea et al., 1987; Simon et al., 2008; Simon et al., 2007; Spohrer and Soloway, 1986; Spohrer et al., 1985; Vessey, 1985, 1989; Xu & Rajlich, 2004.

2. **Programming Strategies**

- **Domain knowledge**

- **Programming by problem-solving**
  - Abboud, 1994; Barnes et al., 1997; De Raadt et al., 2009; De Raadt, Watson, et al., 2004; Deek et al., 1998; Gomes and Mendes, 2007a, 2007b; Ismail et al., 2010; Linn & Clancy, 1992; McCracken et al. 2001.

- **Programming process**
  - Ben-Ari, 2001a; Fee and Holland-Minkley, 2010; Guzdial, 2009; Hmelo-Silver et al., 2007; Kay et al., 2000; Kirschner et al., 2006; Linn, 1986; Sirjani, 2011b; Mayer, 2004.

- **Programming structure, plans, and related tools**
  - Expert’s knowledge
    - Goals and plans
    - Templates
    - Schemas
    - Patterns
    - Representation of experts’ knowledge
      - Tools for using experts’ approaches
    - Bonar and Cunningham, 1988; Bonar and Liflick 1990; de Barros et al., 2005; Fix and Wiedenbeck, 1996; Guzdial et al., 1998; Johnson and Soloway, 1985; Linn, 1992.

3. **Teaching Approaches**

4. **Programming Mental Models**

- **Notional machine and variables**

**Figure 2.4 Categorisation of literature in teaching novices programming**
Programming strategies refer to how to use programming knowledge to develop programs. It was suggested that programming strategies are more important than programming knowledge. Programming strategies consist of four parts: 1) understanding the program application domain; 2) applying problem-solving strategy; 3) having a process for programming; and 4) using programming structures, plans, and tools.

Understanding the program application domain was considered as the first step of programming. Literature studies show that programmers use a more top-down comprehension process in a familiar application domain while using a more bottom-up comprehension process in an unfamiliar application domain. However, if the designers had an appropriate problem-solving plan from experience, they would use the plan instead of other methods. Although application domain knowledge can be accumulated from experience, we believe that it would be beneficial for novices to solve problems in a familiar application domain.

Most of the problem-solving approaches in the programming context were mainly adapted from idea of problem solving in a four-phase process in mathematics education, which has been interpreted in various formats, including: 1) understanding the problem; 2) designing a plan; 3) implementing the plan, and 4) testing and debugging. Although problem-solving strategy was advocated to be a critical part of novice programming in the literature, there was no consensus about teaching problem solving in introductory programming. Problem-solving strategy had largely been either moved into a separate course or taught implicitly rather than explicitly.

Based on the arguments for and against minimal guidance during instruction proposed in the Educational Psychology community, Guzdial (2009) was concerned that teaching novices programming by problem-solving was ineffective. The debate in the Educational Psychology community reached a consensus that it is ineffective to teach either problem-based learning or inquiry-based learning without scaffolding. PBL was shown to yield a significant improvement in programming competence, but it also required substantial effort to introduce. Moreover, a neo-Piagetian approach may improve learning, but it needs support of “chunk” of knowledge. On the other hand, although problem-solving strategy is commonly used in programming, it only suggests the general direction of the programming process. In other words, problem-solving strategy only tells novice what to do. There is no detailed guidance for novices to learn
how to solve the problem. Guided methods such as providing a programming process was effective and promising to support problem-solving strategies.

The concepts of goals and plans were introduced by Soloway and his colleagues in 1980s. A goal is a certain objective that a program must achieve in order to solve a problem and a plan corresponds to a fragment of code that performs actions to achieve a goal. Goals and plans are the basic building blocks for analysing problems and generating programs. Soloway (1986) proposed four general strategies for the composition of plans into a program: abutment, nesting, merging, and tailoring. Plan linkage by dataflow had been proposed by Letovsky and Soloway (1986) and by Pennington (1987). Soloway (1986) advocated building up a plan library and strategies to use the library for programming. He also advocated explicitly teaching novices both these libraries and strategies. Concepts similar to plans, such as templates, schemas, and patterns, have also been used in the literature. The programming code is linked by control flow while the plans are connected by dataflow. The key challenge is how we make a transition from dataflow related plans into control flow dominated programming code. Most tools in the literature supported mapping from goals to plans. However, not all of them supported mapping from plans to programming code.

Subsequently, when teaching novices goals and plans, it is necessary to make further development by taking advantage of these tools. We propose rethinking the use of the intermediate representations that Bonar and his colleagues applied, for example the way of identifying an informal plan as well as the way of representing plan relationships. Their experience inspired us to recognise that we need to develop more intermediate representations between the visual plans and programming code. Moreover, we also propose providing scaffolding in terms of additional support and guidance to novices to effectively use this expert knowledge. We now turn to consider the study of programming mental models in order to effectively represent them to novices.

Approaches to teaching programming address both general teaching strategies and particularly problem-solving strategy in programming. Many general teaching strategies make a trade-off between using a full language to a reduced size language, to beyond a real programming language. For example, the commonly used approaches included the sub-language approach, the mini-language approach, and the “syntax-free” approach. Although most approaches make teaching and learning simpler, they limit the learning of problem-
solving to a synthetic environment rather than a real programming language. On the other hand, the “literacy” approach based on real-world examples may take a long time for some students to understand the programs. A “natural” programming approach, introduced by Pattis (1993), uses a prewritten subprogram library. However, there is no clear guideline or detailed process to organise and design a program by using these subprograms.

Mental models not only support program development, but also help program comprehension. A mental model of the computer as a notional machine has been widely applied in teaching novices programming. Novices’ mental models of programming concepts, particular of variables, have also been brought into several investigations. More attention has also been paid to the role of variables in novices’ programming.

In essence, experts have programming knowledge of goals and plans. They also have strategies of using their knowledge. Mental models represent how the knowledge is used under the strategies. In order to fill the gaps towards experts’ mental representations proposed by Wiedenbeck et al. (1993), in this thesis we would consider the following five key areas:

1. how to organise goals;
2. how to map from goal to final program;
3. how to represent goals and plans;
4. how to connect plans and to be associated to control flow connected program code; and
5. how to support program implementation by using plans.
Chapter 3 Overview of the Visual Goal-Plan Approach

The literature study in Chapter 2 has revealed that although a wide range of approaches have been proposed to improve novices’ learning of programming (Kay et al., 2000; Pears et al., 2007; Robins et al., 2003), there continues to be a high rate of failing or withdrawing from the first programming course (Lahtinen, Ala-Mutka, & Jarvinen, 2005). A number of reasons have been proposed for this, such as “fragile” knowledge of programming concepts (Lister et al. 2004; McCracken et al. 2001), lack of problem-solving strategies and plans (de Raadt, 2008; Winslow, 1996), and lack of detailed mental models (Du Boulay, 1986; Winslow, 1996). There seems to be a broad consensus that “novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs” (Winslow, 1996, page 17). Therefore, the purpose of this chapter is to provide an overview of the development of an efficient approach that focuses on the heart of the matter: how to teach novices to construct programs. It starts from recent use of visual programming languages and then looks at the previously advocated goal-plan approach as well as a recently proposed programming process. Finally, it proposes a new visual goal-plan approach based on the combination of these three ideas.

As explored in Chapter 2, teaching computer programming is thought to be “hard” and a challenge. Students feel scared and try to avoid programming (Jenkins, 2001). Even if programs are written in English, programming is still believed to be as difficult to learn as a foreign language. According to Cognitive Load Theory (CLT) (Paas, Renkl, & Sweller, 2003; Van Merriënboer & Sweller, 2005) (see Chapter 7), the human brain has a limited capacity in working memory for thinking, reasoning and judging. Therefore, only a limited amount of information can be handled at the same time without overloading working memory and then be easily lost due to overload.

Further study about poorly performing students in computer programming suggested that the failure of students is due to spending large amounts of their cognitive resources in the working memory not only on learning concepts and understanding problem statements, but also on learning the syntax of a computer language and coping with the programming environment (Mason & Cooper, 2012). That is to say, inadequate cognitive resources are left free in these students for their development of programming.
Having reviewed the existing research in cognitive load theory (CLT) and its implications for teaching computer programming, we decided to provide novices with a process for using goals and plans in a visual programming environment. The use of goals and plans helps novices to reduce the extraneous cognitive load of organising the details for a plan and to increase resources for germane cognitive load. In a visual programming environment (such as Scratch), novices do not have to remember syntax and to learn the use of a compiling tool, which helps to reduce intrinsic cognitive load. Moreover, the use of a process for designing goals and plans reduces extraneous cognitive load and increases the germane cognitive load of having to decide "what do I do next?".

3.1 Visual Programming Environments (VPEs)

Using visual programming environments (VPEs) is considered as a way to reduce cognitive load on memorising programming language syntax. Recently, researchers have drawn attention to VPEs (Kelleher & Pausch, 2005), which are also intended to provide an attractive, easy, and fun way for novices to learn programming. However, VPEs such as Alice (Dann & Cooper, 2009), Scratch (Resnick et al., 2009), Scratch for Second Life (S4SL) (Harrell et al., 2008), and Puck (Kohl, 2007) have not turned out to be effective in improving the learning of programming. These VPEs are similar in that they use pre-built "blocks" representing programming language constructs and a program is built up by dragging-and-dropping different blocks. Although using VPEs can prevent syntax errors, researchers have found that the approach is not effective (Klassen, 2006; Sykes, 2007). After teaching with Alice for three semesters, Klassen (2006) found that it did not serve the goal of providing a solid programming concept to students.

Conversely, Lister (2011a) concluded that using Scratch and Alice will allow novice programmers to make better initial progress compared to using other languages, but that without a “pedagogical rethink of what should happen after these tools”, there would still be issues when students are required to perform tasks that require transitive inference, such as realising that checking whether an array is sorted is equivalent to checking whether each pair of consecutive items are in order. In other words, visual programming languages may support novices to start programming. However, there must be a well-defined pedagogical method to help novices to bridge the gap to becoming experts.

Therefore, we propose to construct a detailed programming process. In the process, we mainly
apply the previously advocated goal and plan approach (Soloway, 1986) in the phases of decomposition and composition (or analysis and design). We also use “test-early” and “test-driven learning” approaches in the process. Moreover, the combination of these ideas for the detailed process motivates us to implement a framework to support early feedback from using a goal-plan approach within VPEs. Before proceeding to discuss the detailed process, we start from the concepts of goals and plans and then turn to the construction of their visual model.

### 3.2 The Goal and Plan Approach

Recall that a goal is a certain objective that a program must achieve in order to solve a problem (Letovsky & Soloway, 1986), and a plan (Spohrer et al., 1985) corresponds to a fragment of code that performs actions to achieve a goal. Goals and plans are key components in representing problems and solutions (Soloway, 1986). Plans or schemas are proposed to be provided to students to solve problems as they are constructed as “canned” solutions in the form of programming language templates (Soloway, 1986; Rist, 1989). Furthermore, plans can be provided in a library and the combination of them can be then applied to solve problems. We argue that using the provided plans from a plan library, novices can reduce their cognitive load for constructing these plans. By enabling students to focus on merging these plans to create a problem solution, the germane cognitive load is increased. Although strategies of merging plans have been proposed (Soloway, 1986), there are insufficient instructions on how to “put the pieces together”. In other words, there is no detailed process for manipulating these plans.

Moreover, a tool, GPCeditor, (Guzdial et al., 1998) was created that supported novices to write a program based on the decomposition and composition of goals and plans. However, this approach paid much more attention to low level plan code details rather than to effectively support the design at the plan level, i.e. providing feedback before merging all the plan segments. Although the tool can generate segments of plan code, there was not a detailed process for performing the composition of those pieces of plan code, and, furthermore, the tool’s evaluation did not clearly demonstrate a significant advantage.

Similarly, pedagogical programming patterns were advocated by Porter and Calder (2003) who proposed using small pieces of code segments in teaching novice programmers. A tool, ProPAT, was inspired by the idea of programming patterns (de Barros et al. 2005) allowing
novices to insert code from the pedagogical patterns. However, there was weak support for how to apply these patterns in the goal analysis. Furthermore, there was not a detailed process of programming. Earlier, Bieliková and Návrat (1998) attempted to teach students a set of standard structures (or program schemata) as well as a method for how to apply them, but there was only a weak description of the goals achieved by the schema, and testing and debugging were not supported.

Recently, the goal and plan concepts have been taught as programming strategies in curricula by de Raadt (2008). Each strategy was also called a plan, which was basically pattern-like program code with examples. This approach attempted to integrate plans to build the program code after explicitly introducing goals and plans. However, it lacked a detailed programming process for the development from goals to program via plans. Firstly, there were no clear guidelines for performing goal analysis. Secondly, there was no well-defined process for merging plans. Thirdly, there are no tools to support its strategies of programming implementation.

In summary, a number of approaches that have used goals and plans have been proposed, but have failed to provide detailed processes and no tool has been developed to support their use at design time. This implies that even if students understand each plan segment they still do not know “how to put them together”. Therefore, we have decided firstly to develop a framework to support working with goals and plans in a visual programming environment (VPE) and secondly to develop a clearly defined programming process for the new visual goal-plan approach.

3.3 The Visual Goal-Plan Approach

In this section, we propose to design a visual model of goals and plans in the data-flow paradigm. Our model allows the programming problem to be analysed using a goal diagram by using the visual notation of a group of goals and their data-flow links. Subsequently, the solution can be presented as a plan network of the set of corresponding plans using a visual notation. After developing a data-flow framework to implement the plan networks in a visual programming language, specifically BYOB, we can not only provide a plan library in the framework, but also offer certain patterns for students to create their plans and to add to the plan library. Within the data-flow framework, the network of unmerged plans is executable.
This allows early testing to provide feedback on the decisions made in the earlier phases of analysis and design.

In order to develop a visual framework for our new goal-plan approach, first of all, we need to develop a visual notation to represent goals and plans so that novices could straightforwardly and successfully develop their designs in a diagram. A key question in developing a notation is control flow programming paradigm versus dataflow programming paradigm.

Whereas the control flow programming paradigm deals well with programming code details at a low level, dataflow is a good starting point for programme design at a high level (Good, 1999a, 1999b). Good (1999a, 1999b) suggested that changing paradigms may be useful for program comprehension. Based on the analysis of plan combination goal-plan analysis by Soloway and his colleagues (Letovsky & Soloway, 1986; Soloway, 1986; Spohrer et al. 1985), particularly by Ebhrahimi (1992), we decided to use the data-flow paradigm to design our visual notation of goals and plans and then apply the control flow paradigm to program code details (see Chapter 4 for the discussion of considering this choice). Specifically, we are motivated to develop our visual notation only in the data-flow paradigm because plans can be executed in this paradigm before they are merged. Consequently, early feedback can be collected immediately from the execution of unmerged plans after the analysis and design are completed at the plan level.

Using the data-flow paradigm, we designed the visual notations for goals and plans based on the theory and principles of visual notation (Rumbaugh, 1996). The development of the visual notation of goals enables a problem to be described by a goal diagram showing a group of related goals. These goals can be achieved by relevant plans identified from the existing plan library, where the dataflows between them are represented as links between goals and between plans. All the plans and links build up a plan network that conform to the goal diagram and become a solution of the program at the design level.

After designing visual notation for goals and plans, using a data-flow paradigm, we model goals and plans as consuming and producing dataflow, i.e. sequences of data. The dataflow is the linkage between goals as well as between plans. In principle, these interleaved plans are executable through the data-flow linkage before being merged into a single program in the control flow paradigm. We then turn to develop a framework by using BYOB which allows plan
networks to be executed without being merged. The framework also provides an environment for the transition from plan networks to the final program. (See Chapter 4 for more details of design for the visual model of goal-plan approach and Chapter 5 for the implementation for the data-flow framework.)

By using the data-flow framework, we need to develop a detailed programming process for the visual goal-plan approach from analysis in the data-flow paradigm to the final program in the control paradigm. In order to engage novices within the process, we also need feedback to be provided from each phase of the process.

3.4 The Process of Programming

Recall that problem-solving strategy only indicates the direction of programming rather than providing support by guided learning. The process of programming fits into this gap by providing detailed guidance of developing programs. The idea of using an explicit process of programming in computing education is not new. An early programming process was developed based on the classical software life cycle model and inspired by a process for writing English essays (Gantenbein, 1989). The commonly used programming process includes five steps: 1) defining the problem; 2) selecting an approach; 3) designing a solution; 4) implementing the solution; and 5) testing the implementation. Providing feedback from each step and a “test-driven learning” approach are also proposed as important parts of the programming process for novices (Pattis, 1990; Janzen & Saiedian, 2008). However, in these work there is no detailed process for programming, and no structures that experts use such as schemas or plans, are applied in the process.

There are more approaches that have provided detailed processes, but they also tended not to use the concepts of goals and plans. For example, “Programming by Numbers” (Glaser, Hartel, & Garratt, 2000) provides a clear process to create the smallest components of functions. It breaks the programming process into a series of well-defined steps and gives students a way of “programming in the small”. A similar, but more detailed, systematic design method was applied by Felleisen et al. (2004) to produce well-specified intermediate products in a stepwise fashion called “TeachScheme”. However, although both approaches emphasize the detailed process, the goal and plan concepts are not included. Additionally, both approaches are data-driven and more suited to functional programming languages than to mainstream procedural
languages. More recently, a stepwise improvement process, STREAM (Caspersen, 2007; Caspersen & Kölling, 2009), was developed as a conceptual framework for teaching novice object-oriented programming. However, it did not use a VPE specifically designed for novices. Moreover, none of the above programming process approaches has tools to support them, especially in a VPE.

Some researchers considered using a programming process and goals to reduce cognitive load without using a visual language. A programming process is considered as scaffolding that supports temporarily achieving a solution, and the scaffolding can be eventually removed (Caspersen & Bennedsen, 2007). Other researchers considered using visual programming language to reduce intrinsic cognitive load, but did not use plans (Margulieux, Guzdial, & Catrambone, 2012). In order to reduce extraneous cognitive load, the combination of several techniques is also proposed such as the use of worked examples, sub-goal labels (the key words of tasks), and scaffolding. However, this work did not propose using plans in a visual programming environment and providing immediate feedback from each phase of the process to effectively motivate and engage novices in the process.

Therefore, we decide to develop a detailed programming process that uses a “test-early” approach and supports our visual goal-plan approach working in the data-flow framework. The process is divided into several phases. Each phase includes several steps and is designed to provide feedback to engage students with the process. The details of the process are described in Chapter 6. The teaching method of this approach is covered in Chapter 7, and finally, the evaluation of this approach is reported in Chapter 8.

### 3.5 The Proposed Approach

We conclude with the hypothesis that what is needed is a programming process that students can follow along with a structured means of representing the parts of a solution using an easy-to-use notation. Specifically, we conjecture that combining goals and plans with a detailed process will yield an effective means for teaching programming. Therefore, our proposed approach (see Figure 3-1) thus combines three ideas: 1) using a Visual Programming Environment (VPE); 2) using goals and plans; and 3) having a well-defined process. This combination is novel and carefully motivated.
Table 3-1 summarizes the related work in terms of whether it uses goals and plans; whether a detailed process is provided; and whether it has been supported in a VPE. As can be seen, existing work tends to use one or two of these features, but none of them has all three features.

Table 3-1: Comparison of related approaches

<table>
<thead>
<tr>
<th>Literature Study</th>
<th>VPE</th>
<th>Goals /Plans /Patterns /Schemata</th>
<th>Detailed Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dann and Cooper, 2009</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Harrell et al., 2008</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Klassen (2006)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Kohl, 2007</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Resnick et al. 2009</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Bieliková and Návrat (1998)</td>
<td>No</td>
<td>Schemata</td>
<td>Weak</td>
</tr>
<tr>
<td>Barros et al. (2005)</td>
<td>No</td>
<td>Patterns in Code</td>
<td>Weak</td>
</tr>
<tr>
<td>Soloway and his colleagues (1980s &amp; 90s)</td>
<td>No</td>
<td>Plans in Code</td>
<td>Weak</td>
</tr>
<tr>
<td>Glaser et al. (2000)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Felleisen et al. (2004)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Caspersen and Bennedsen (2007)</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Caspersen and Kölling (2009)</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Margulieux, Guzdial, and Catrambone (2012)</td>
<td>Yes</td>
<td>Only Goals</td>
<td>Weak</td>
</tr>
</tbody>
</table>

This Research: a “visual goal and plan” approach

Yes | Yes | Yes
Chapter 4 The Design Notation of Goals and Plans

In Chapter 3, we presented an approach for teaching novice programming that combines three approaches: working in a visual programming environment (VPE), using the ideas of goals and plans, and having a clearly defined process for programming with goals and plans in a VPE. First of all, we need to develop a design notation to represent goals and plans. In this chapter, we introduce a visual design notation to represent goals and plans. Firstly, we describe the general principles for designing a visual notation that were applied in this approach (see Section 4.1). After comparing the ability of different plan representations to support the desired process in both dataflow and control flow paradigms (see Section 4.2), we decided to develop our design notation of goals and plans using a data-flow paradigm at the programming design level (see Section 4.3 and 4.4). We then move to a control flow paradigm at the programming code level. And finally, we evaluate the design notation against the design principles (see Section 4.5).

4.1 Visual Notation Design Principles

“**A visual notation** (or visual language, graphical notation, diagramming notation) consists of a set of **graphical symbols** (visual vocabulary), a set of **compositional rules** (visual grammar) and definitions of the meaning of each symbol (visual semantics). The visual vocabulary and visual grammar together form the **visual** (or **concrete**) syntax....A valid expression in a visual notation is called a **visual sentence** or **diagram**. Diagrams are composed of **symbol instances** (tokens), arranged according to the rules of the visual grammar.” (Moody 2009, page 757)

Visual notation has a long history of being widely used to improve human communication. After the diagram creator encodes information in a visual form, the diagram user decodes (interprets) the visual document (Moody, 2009). Visual notations have two spatial dimensions (i.e. horizontal position and vertical position) and are processed in parallel by the human visual system, in contrast to textual representations that are linear one dimensional and are processed serially (Bertin, 1983; Larkin & Simon, 1987; Rumbaugh, 1996). Rumbaugh (1996) also argued that using graphic symbols can make a notation more understandable than just text keywords because “the symbols minimise the perception time for the basic constructs by
We aimed to develop a visual notation of goals and plans for novices so that they could easily and effectively develop designs that support the process of programming. In order to design an effective notation for goals and for plans we take into account three sets of principles for the design of graphical notation (Moody & Van Hillegersberg, 2009; Moody, 2009; Rumbaugh, 1996). Although these principles are similar, we believe Rumbaugh’s principles are more appropriate to directly guide our design of the visual notation, while Moody’s principles and Moody and van Hillegersberg’s principles are more suitable for evaluating the developed notation (see section 4.5).

Rumbaugh’s principles (Rumbaugh 1996) are:

1. “Clear mapping of concepts to symbols”: one concept would be mapped to one and only one graphical symbol. If a concept is represented by more than one symbol, it produces symbol redundancy, which is potentially confusing to users of the notation. For example, in the Unified Modelling Language (UML) class diagram, a class interface is represented by either a rectangle or a circle, whereas only one of the two different symbols should be used to represent the interface;

2. “No overloading of symbols”: one symbol should represent one and only one concept rather than two different concepts. Conversely, if a symbol represents different concepts, it results in symbol overload. For example, in the UML class diagram, a rectangle symbol is used to represent both class and object. However, two different symbols should be used to represent classes and objects respectively;

3. “Uniform mapping of concepts to symbols”: the same symbol should be applied to the same concept at different places in the model. In the UML, symbols are used consistently in different models within object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented software engineering (OOSE). For example, an object concept is represented by a rectangle in a class diagram, in a sequence diagram, or in a collaboration diagram;

4. “Easy to draw by hand”: the symbol used in design should be represented by a basic shape, or a combination of basic shapes. However, when drawing by hand some

---

distinctions are hard to make, for example, line thickness, or the difference between a rectangle and a round edge rectangle;

5. “Looks good when printed”: the size, font and colour of all the symbols must be clearly visible in printing;

6. “Must fax and copy well using monochrome images”: some lines or distinctions between colours and between fonts may fade or become lost when a diagram is faxed or photocopied. However, since 1996, when Rumbaugh’s criteria were published, copying and faxing have become less important;

7. “Consistent with past practice”: if it is possible, reuse existing symbols from previous practice. For example, the symbol of rectangle for Class in the current Unified Modelling Language (UML) was inherited from the symbol of rectangle for Entity in the traditional Entity-Relationship Diagrams\(^\text{15}\) (ERD);

8. “Self consistent”: a symbol must present the same concept consistently. For example, a primary key symbol must be consistently applied in every entity in an ERD. While Principle 3 refers from concept to symbol, Principle 8 here discusses from symbol to concept;

9. “Distinctions are not too subtle”: the notation should avoid subtle distinctions between symbols. For example, in the UML’s sequence diagram notation, the distinction between synchronous and asynchronous messages is indicated by the shape of the arrowhead on the message, which is a subtle difference that can be missed.

10. “Users can remember it”: a notation should use a limited number of symbols in order to be easily remembered by users. ERDs are good examples of using a small limited number of symbols in database design. Rumbaugh argued that symbols should be “mnemonic”, i.e. attributes of the notation should relate to its meaning, and thus make it easier to remember. For example, in the UML, a stick figure is used to denote an actor;

11. “Common cases appear simple”: making common cases simple is highly desirable although it may lead to certain redundancy. For example, in the UML diagram type e.g. a class diagram, it can have a hollow or filled diamond to indicate aggregation or composition relationships at the ends of association. However, in the more common case of pure association, the notation is simply a line (with no diamond);

\(^{15}\) http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.123.1085
12. “Suppressible details”: some information can be shown at different levels of detail. For example, detailed information about non-key attributes can be omitted at the initial design of an ERD. Instead, only primary key and foreign keys are described in each entity.

These principles informed the design of our visual notation for goals and plans. They were used in the development of visual notations of goals and plans in Sections 4.3 and 4.4 as well as in the evaluation in Section 4.5. However, before we proceed to develop the details of our notation, a key decision that needs to be made is whether to use a dataflow or control flow based representation.

### 4.2 Dataflow versus Control Flow

The notation that we develop needs to support a novice programmer in designing a program in terms of goals and plans, and then proceeding, in a systematic manner, to an implementation. We therefore begin by considering the various plan composition strategies, taking into account how well they can be used in a data-flow based notation and in a control flow based notation to support the desired process, including early feedback before merging plans. First of all, we look at the early research on plan composition.

Soloway and his colleagues (Letovsky & Soloway, 1986; Soloway, 1986; Spohrer et al., 1985) proposed that the programming process begins from a tree structure of goals resulting from goal analysis and decomposition. This tree is developed by stepwise refinement and each goal is achieved by a predefined plan from an existing library. Ultimately, all the goals in the tree structure are realised as a combination of plans. Originally, Soloway (1986) proposed three ways of combining plans. These three basic relationships are presented as (a), (b), and (c) in Figure 4-1: (a) sequential (Plan B begins after Plan A finishes, and Plan C begins after Plan B finishes); (b) nested (the sequential Plan B and Plan C are used as two of the steps within Plan A); and (c) interleaved (the steps of Plans A, B and C are merged with each other).

Subsequently, Ebrahimi (1992) proposed an additional way of combining plans (see (d) in Figure 4-1): branching (Plan A uses either Plan B or Plan C, depending on a condition).

We now consider how well these ways of combining plans can be supported when using both control flow and dataflow based notations. A key issue is early feedback from the plan combinations. It is desirable to give early feedback to help novices and to assist them in early detection of errors. In other words, the plan combinations need to be executable for the
purpose of providing feedback before they are merged. Otherwise, novices could make errors in plan compositions (Ebrahimi, 1994) such as missing plans, misplacing plans, using malformed plans, and misusing plans. Hence, if the errors can be caught earlier using feedback from unmerged plan combinations, novices would be encouraged to explore and be more confident to merge the details of those plans that lead to the solution.

However, as we will see, not all plan combinations can be executed directly: some require plan merging to be performed. Therefore, a key question to be considered is whether the plan combinations are directly executable when using control flow and when using dataflow. Specifically, interleaved plans are an issue, since they need to be merged to form an executable single-threaded program (see Figure 4-2). As can be seen in Figure 4-3, the representation is unable to support unambiguously capturing the order of execution for three interleaved plans (Read All Number Plan, Count All Number Plan, and Sum All Number Plan). More importantly, it does not provide support for the sequencing that the executable (procedural) program needs to read each input, and then process it, including updating both the sum and the count, before the next input value is read (see Figure 4-2). In other words, the interleaved plans, depicted in a control flow based notation, are not executable until they are merged.

Figure 4-1 Relationships of plans, modified from Ebrahimi (1992, page 311) (a) Sequential Plans, (b) Nested Plans, (c) Interleaved Plans and (d) Branched Plan
Procedure Average
sum := 0; // Sum All Numbers Plan
count := 0; // Count All Numbers Plan
read (number); // Read All Numbers Plan
while (number <> SENTINEL) do // Read All Numbers Plan, Sum All Numbers Plan, and Count All Numbers Plan
begin
    sum := sum + number; // Sum All Numbers Plan
    count := count + 1; // Count All Numbers Plan
    read(number); // Read All Numbers Plan
end
if count <> 0 then // Guard Plan
begin
    average := sum / count; // Compute Plan
    result := “Average is” + str(average); // Compute Plan
end
else // Guard Plan
begin
    result := “Error in data”; // Guard Plan
end
write(result); // Output Plan

Figure 4-2 Example of merged plans presented in pseudocode

Figure 4-3 Example of plan representation in control flow, modified from Ebrahimi (1992, page 301)

On the other hand, by adopting a dataflow based representation, we can avoid the need for merging the interleaved plans before any execution can be done. We model goals and plans as consuming and producing sequences of data. A plan can be executed to completion by storing
results in a data buffer. For the above example, dataflow\(^{16}\) starts from the Read All Numbers Plan (see Figure 4-4). The same dataflow, "Numbers", has two streams from Read All Numbers Plan to both Count All Number Plan and Sum All Numbers Plan. Afterwards, another two streams, “Count” and “Sum”, flow respectively from Count All Numbers Plan and Sum All Numbers Plan to Division Plan (which is similar to the Guard Plan in Figure 4-3, but has the Compute Plan embedded). Finally, the dataflow “Result” moves from Division Plan to the last Output Plan. In other words, by using a data-flow model, we can execute unmerged plans. This is a significant difference between control flow and data-flow models when using goals and plans, because it enables novices to receive feedback before plans are merged into a final (procedural) program.

![Figure 4-4 Example of plan representation in dataflow](image)

We now consider the other cases of plan composition. Firstly, for the sequential plans, both dataflow and control flow representations are natural and support the process. For example, in the control flow paradigm (see Figure 4-3), the Output Plan is executed in the sequence

\(^{16}\) Note that the implementation (described in Chapter 5) makes use of data buffers. For instance, the Read All Numbers Plan executes to completion, reading all numbers, and these numbers are stored in a buffer.
after its prior nested plans (the Guard Plan and the Compute Plan). Similarly, in the data-flow paradigm (see Figure 4-4), the Output Plan is also executed after its prior plan (the Division Plan).

Next, for the nested plans, it is clear how to present them in control flow. For example, the Compute Plan is nested in the Guard Plan (see Figure 4-3). However, in the dataflow paradigm, nested plans are not always executable without merging. This is because the nested plan (for example, the Compute Plan, being used within the Guard Plan) is not used on a whole dataflow (see Figure 4-4). It is executed only if the dataflow contains the relevant items (for example, when the dataflow for divisor is not equal to zero). Otherwise, it is not executed (for example, when the dataflow for divisor is equal to zero). Therefore, it is arguably less natural to use dataflow for nested plans. Finally, for the branching plans, control flow also deals with them well. For example, the Guard Plan in the Average program can also be represented by using branching plan composition, which can be naturally modelled using a notation based on the control flow paradigm (see Figure 4-5). On the other hand, using dataflow representation, a conditional split needs to be introduced, which has a condition, a single in flow and two out flows, which is similar to the “Filter” and “Split” operators in Yahoo Pipes. This works for cases where there is a single value, and can be extended to deal with multiple values. Overall, both dataflow and control flow can support branching. However, it is arguable that branching is a less common or less important construct as evidenced by its exclusion from the original forms of plan combination by Soloway (1986). Therefore, for simplicity to students, we exclude branching by assuming that instances of branching have been combined into a single plan when using a notation based on the data-flow paradigm.

---

17 Yahoo Pipes is an online service for creating data mashups. For more details, see http://pipes.yahoo.com/pipes/docs?doc=modules

18 This extension requires some complexity in buffer implementation in order to deal with the issue of ordering information.
In summary, the key difference between control flow and dataflow is the handling of interleaved plans. In the data-flow paradigm, there is no need to merge plans that are interleaved (such as the above Read All Numbers Plan, Sum All Numbers Plan and Count All Numbers Plan in Figure 4.4), whereas in the control flow paradigm, plans with the interleaved relationships must be merged with each other before they can be executed. Turning to the other plan combination types, both dataflow and control flow paradigms can support sequential plans, and branching plans. Although the support is somewhat better in control flow for nested and branching plans, the benefit is remarkable in dataflow for the interleaved plans. Therefore, using the data-flow paradigm has a significant advantage in that it does not require merging for interleaved plans.

Another advantage of a dataflow based notation is that it is possible to reuse plans within the same program. In the control flow paradigm, data is passed from plan to plan by using shared
variables (at the global or class level). However, this prevents a plan from being used more than once. For example, to display the sum of even numbers and the sum of odd numbers separately, two sum plans must be used. Unfortunately, they only give the total result of all the numbers rather than separated sums for odd and even numbers. This is because, when using a control flow representation, the two sum plans use the same shared variable. On the other hand, in the data-flow paradigm, data is passed from plan to plan through named dataflows, which allows the sum plan to be used twice, giving correct results.

We also considered the results of comparing both control flow and data-flow approaches by Good (1999a, 1999b), who evaluated how participants’ comprehension was affected by the language paradigm. She found that participants using a control flow visual programming language (VPL) scored significantly higher on questions about control flow (the sequence or the order of events in a program), operations (actions in a program such as assignment) and state (current situation of objects and events under certain conditions), while others using a data-flow VPL scored higher on questions about dataflow (data movement and transformations) and function (the purpose or the goal of a program). Furthermore, the study also suggested that changing paradigms may be a useful teaching aid for program comprehension.

Although dataflow has advantages, it also has disadvantages, and control flow is complementary in terms of its advantages and weaknesses. Using a control flow paradigm is closer to program development at the code level. In other words, novices can understand programming details better in control flow paradigm. Conversely, using a dataflow paradigm is closer to program development at the design level. In other words, novices can understand program structure or relationships of different parts of in the program better in dataflow paradigm. In order to take advantages of both design and programming details, we propose to start with dataflow because it allows early feedback (as discussed earlier) and also because Good’s results suggest that it is better for understanding the purpose of program, which is what design is about. We then shift to control flow because Good’s results also suggest that it is better for dealing with details of program in terms of sequence, process, and condition. Accordingly, we also believe the shifting of paradigms is better than remaining in the same paradigm.

Based on the above comparisons, we concluded that neither control flow nor dataflow is
sufficient on its own. Dataflow is a good starting point of design at a high level, whereas control flow deals well with code details at a low level. We acknowledge the additional complexity associated with using both paradigms. Therefore, in the process defined, we only use a single paradigm in each phase at a time. We also provide effective support (such as a detailed process, visual notation, and tool) in both paradigms as well as in the transition from one paradigm to another. In an initial trial with a small group of the University Foundation Year students, we found that students were able to use the process. We concluded that the complexity of using two paradigms is manageable. Therefore, we decided to use the data-flow paradigm for goals and plans and then the control flow paradigm for program code details. Significantly, the unmerged plans in our approach are executable using the data-flow paradigm, which makes it possible to provide feedback to students at the design level. Although plan merging is deferred to the code level, a well-defined process is discussed in Chapter 6 in order to support the merging of pattern-based plan segments. Moreover, feedback from further intermediate steps can also support our transition from the data-flow paradigm to the control flow paradigm.

4.3 Visual Notation of Goals

We now present a dataflow based visual notation for goals to be used at the design level. We propose a notation where goals are represented by icons and are linked by “data-flow” arrows (see Table 4-1). Recall that a goal is a certain objective that a program must achieve in order to solve a problem. We categorise goals into three related groups: input, output, and processing. An Input Goal is a goal for getting values from the keyboard or a file; a Process Goal is a goal for processing values; and an Output Goal is a goal for displaying results. For example, when a program needs to display the sum of a sequence of values, the goal of displaying the result is classified as an Output Goal. However, before the result can be displayed, the program also needs to achieve the goal to input this sequence of values and the goal of processing these values to compute the sum. The goal to input the values is identified as an Input Goal. And the goal for processing the total of these values is a processing goal, specifically, one named Sum Goal.

Goals are linked by dataflows, sequences of values that “flow” between goals. The direction of dataflow is from the source goal to the destination goal. In the typical case, a dataflow has a single source and single destination to link two goals, and is called a basic data flow or
In the example of displaying the sum of a sequence of values, there are two basic dataflows among the above three identified goals. The first dataflow is from Input Goal to Sum Goal, which means after achieving the input goal of obtaining a sequence of values, the dataflows to the Sum Goal to calculate the sum of these values. The second dataflow is from Sum Goal to Output Goal.

However, it is also possible for a dataflow to have more than one destination forming a “fork”. For example, in order to calculate the average of a sequence of values, the first goal, Input Goal, is to input these values, and then two goals compute respectively the total and the count of these values (Sum Goal and Count Goal). The data from the Input Goal has data-flow streams to both the Sum Goal and the Count Goal (the “fork” dataflow can also be seen in Figure 4-4). Both streams of dataflow convey the same values.

A goal sends or receives dataflow through its ports. We make the assumption that each goal has a single out port and we use arrows (denoting dataflow between goals) to show the direction of the flow. Thus, for the goal with a single in port, a port can be identified by the combination of the goal’s name and knowing whether the port is an “in” or an “out” port rather than using a port name in the goal diagram, which aims to simplify the initial design stage for students. For the case of multiple in ports, a number is added such as in1 and in2 to distinguish the ports (see Section 4.4).

By convention, data flows from left to right. We assume that the in ports should be on the left side of a goal and the out port on the right side. Subsequently, the combination of goal name and the port name of dataflow source (on the right) uniquely represents the output of a goal while the combination of goal name and the port name of dataflow destination (on the left) identifies the input of the goal. For the case of multiple in ports, the different data-flow arrows (on the left) from different other goals make each in port uniquely identified. Therefore, we can hide the port name and its function (in or out) from the goal name. Instead, we use the combination of both goal and data-flow icons to identify each port. We argue that the suppression of ports in goal notation is significant because it makes simple and easy for students to start.

In accordance with the “good notation principles” (Rumbaugh, 1996), every concept is clearly mapped to a single distinct symbol without overloading. These concepts and symbols are
represented in a one-to-one relationship to match the first two principles. For instance, each type of goal has a unique icon which is only used for that goal type (clear mapping [principle 1] and, there is no overloading of symbols [principle 2]). Furthermore, the same icon is consistently used (uniform mapping, [principle 3]). A goal is, in a draft design, compound, but has not yet been decomposed into sub-goals. Eventually the compound goal will be decomposed, or changed into an atomic goal (if a plan in the plan library can realise it). Initially, we had two different textures for compound and atomic goals: dashed line and solid line. However, through the pilot study, we found that although using a dashed line to indicate compound goals made figures easier to read, students did not use it because they could not easily work out when to use a compound and an atomic goal. In other words, the distinctions between dashed line and solid line are too subtle for students. Thus, we decided not to use the dashed line icons (distinction not too subtle, [principle 9]).

Table 4-1 The Visual Notation of Goals

<table>
<thead>
<tr>
<th>Concepts and Definitions (Visual Semantics)</th>
<th>Symbols (Visual Vocabulary)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input Goal</strong>: a goal for inputting values from the keyboard or a file.</td>
<td>![Symbol]</td>
</tr>
<tr>
<td><strong>Process Goal</strong>: a goal for processing values.</td>
<td>![Symbol]</td>
</tr>
<tr>
<td><strong>Output Goal</strong>: a goal for displaying the result(s).</td>
<td>![Symbol]</td>
</tr>
<tr>
<td><strong>Basic Dataflow</strong>: a linkage between two goals. The direction of dataflow is from one source goal to one destination goal.</td>
<td>![Symbol]</td>
</tr>
<tr>
<td><strong>Fork Shape Dataflow</strong>: the dataflow has one source goal and many destination goals. Every destination goal linked at the end of the fork shape icon receives the same dataflow from one common source goal.</td>
<td>![Symbol]</td>
</tr>
</tbody>
</table>

**Rules (Visual Grammar)**

1) A program must include at least one Input Goal and one Output Goal. The order of goals is generally from Input, to Process, to Output with data flowing from left to right.

2) Each input and processing goal must be linked to either one goal by a basic data-flow symbol, or many goals by a fork shape data-flow symbol.

3) Each processing goal has to be linked from at least one other goal.

4) Each output goal must be linked from exactly one other goal.

5) Input goals must not have incoming dataflows and Output goals must not have any outgoing dataflows.

In order to satisfy principle 3 of uniform mapping, we choose the same data-flow symbols for
both goal and plan visual notations. All the symbols selected are easy to draw by hand (easy to draw, [principle 4]). The goal notation does not have any issues with printing (looks good, [principle 5]), faxing or copying (using monochrome images, [principle 6]). The notation using nodes and arrows is consistent with past practice (e.g. UML) (consistent with past practice, [principle 7]). However, there is no widely adopted and used notation for goals. The three shapes are self-consistent (self consistent, [principle 8]) to match each other in that the three types of goals together can build up a rectangle (see Figure 4-6), which helps to make the notation memorable (users can remember, [principle 10]).

![Figure 4-6 Example of rectangle made by three types of goals](image)

Since we only use a limited number of symbols (five in total), there are no similar symbols using subtle distinctions between them such as using texture, brightness or colour (distinctions are not too subtle, [principle 9]). Meanwhile, it would not be difficult for novices to remember such a small number of symbols (users can remember, [principle 10]). The common case of a dataflow with one source and one destination is simple, and we have also specific notation for the case with multiple destinations. For the common cases of both “basic” dataflow (having one source and one destination) and “fork shape” dataflow (having one source and multiple destinations), we keep the two symbols for single arrow and fork shape multiple arrows in order to make the two cases simple (common cases appear simple, [principle 11]). The name of a dataflow can be suppressed as long as the arrow line connects both a source (out) and one or many destinations (in) of goals or plans (suppressible details, [principle 12]).

Having chosen symbols and icons to depict the goal-related concepts, we now give an example of developing a diagram of goals for a program. In general, a program will have at least one input, one output, and one processing goal. Recall the above example of displaying the sum of a sequence of values: three goals with two dataflows are represented in Figure 4-7. The initial goal, Input Goal, reads a sequence of values from the user. Through the first dataflow from Input Goal to Sum Goal, these values are sent to achieve the Sum Goal for processing the sum. After realising the Sum Goal, the result sum is sent to the final goal, Output Goal, to be displayed.
A simple program might have only one goal of each type and achieve these goals in sequence. However, any but the simplest program will have multiple processing goals, and these can “split” and “join” branches that are linked by normal and forked arrows. The goal diagram for the previous Average example in Section 4.2 is presented in Figure 4-8. After reading a sequence of values from the user by the Input Goal, the process goals are split into Sum Goal and Count Goal to process the sum and the count from input numbers respectively. Both Sum Goal and Count Goal are joined into Average Goal to process the average (= sum/count = \( \frac{\text{a}}{\text{b}} \)). Finally, the average result is sent to the Output Goal to display.

Furthermore, the representation of having multiple goals is not only for processing goals as Figure 4-8, but also potentially for both input goals and output goals. For example, a payroll program (see Figure 4-9) needs to display the total wages and the count for a group of people after receiving the pay rates and working hours for each person (the formula for wage is wage = rates * hours = \( \text{c} \ast \text{d} \)). Therefore, the Input Goal is decomposed into two Input Goals to input two sequences of values of both rates and hours. The Process Goal is divided into a goal to process wages first and then to produce the sum and count of these wages. Finally, the Output Goal is also split into two Output Goals to display both the sum of wages and the number count of people.
4.4 Visual Notation of Plans

A plan is a named code segment that accomplishes a programming goal. A group of plans can be used to build up a Plan Library. Recall that goals are classified as being related to input, processing, or output. Similarly, we classify a plan as being an input plan, processing plan, or output plan. Like goals, plans are in a network, where data “flows” between the plans. The design of plan in this Chapter only shows the interface. The associated code segment for the implementation of each plan will be discussed in Chapter 5. Since a plan is a named code segment that accomplishes a programming goal, the interface of a plan is visualised as a box with double lines on both sides like a sub-program icon in flowchart notation (see Table 4-2) (consistent with past practice, [Principle 7]).

A port is used by each plan for either receiving dataflow from another plan or sending dataflow to all of the directly linked plans. The port receiving dataflow is called an in port while the port for sending dataflow is called an out port. Each port is identified and named by the combination of the name of the plan and the function of the port. For example, the port name Input:out represents the out port of an Input Plan, which means the dataflow generated by this Input Plan will be sent out through this named port. Similarly, Sum:in and Sum:out stand for the in port and out port of Sum Plan, which means the Sum Plan will accept a dataflow from its in port (Sum:in), and after processing these data (computing the total), a new dataflow will be sent out from its out port (Sum:out). For the case of multiple in ports, a number or text is appended to the port function sign “in” for the identification of a port. For example, two in ports from the Dividing Plan can be named either as (Dividing:in1) and (Dividing:in2) or as (Dividing:in.dividend) and (Dividing:in.divisor).

By convention, we show the in ports on the left side of a plan box and the out port on the right side. An Input plan has exactly one port, an output port (shown on its right side) and each Output Plan has exactly one port, an input port (shown on its left side). A process plan has both in and out ports. It can have one or many in ports (shown on its left side), but only one out port (shown on its right side).

We distinguish between ports with functions of in and out as well as that are associated with a dataflow that only involves a single value, and those that are the end points of a dataflow with a sequence of values. In the above example, the data-flow link from the port Input:out to port
Sum:in can contain a sequence of values. However, the dataflow from the port Sum:out to port Output:in will only contain a single value. We use different combination of symbols (specifically using arrow and rectangle) to represent ports having a function of in or out as well as containing a single value or a sequence of values (see the symbols for ports in Table 4-2). A rectangle or rectangles are used to represents single or multiple values on the port respectively. When an incoming arrow is on the left of the rectangle(s), the combined symbol represents an in port function. Similarly, when an outgoing arrow is on the right of the rectangle(s) the combined symbol represents an out port function. Finally, we reuse the symbols of both basic dataflow and fork shape dataflow to depict the dataflow between plans in terms of connection and direction.

### Table 4-2 The Visual Notation of Plan

<table>
<thead>
<tr>
<th>Concepts and Definitions (Visual Semantics)</th>
<th>Symbols (Visual Vocabulary)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Plan:</strong> a named code segment that accomplishes a programming goal. Note that the code segment is not shown in the plan diagram.</td>
<td>![Symbol for Plan]</td>
</tr>
<tr>
<td><strong>Port:</strong> a port is attached to a plan to send (out) or receive (in) dataflow. A port name is the combination of the name of the plan and the function of the port (in or out).</td>
<td>![Symbol for Port]</td>
</tr>
<tr>
<td>To receive or send a singular value.</td>
<td>![Symbol for Port]</td>
</tr>
<tr>
<td>To receive or send a sequence of values.</td>
<td>![Symbol for Port]</td>
</tr>
<tr>
<td><strong>Basic Dataflow:</strong> relationship flow of data values between two plans. The direction of dataflow is from one source plan to one destination plan.</td>
<td>![Symbol for Basic Dataflow]</td>
</tr>
<tr>
<td><strong>Fork Shape Dataflow:</strong> a dataflow that has one source plan and many destination plans. Every destination plan linked at the end of the fork shape icon receives the same dataflow from one common source plan.</td>
<td>![Symbol for Fork Shape Dataflow]</td>
</tr>
</tbody>
</table>

### Rules (Visual Grammar)

1) An Input Plan has only one out port on the right side of plan symbol; an Output Plan has only one in port on the left side of plan symbol; and a Process Plan has one or many in ports on the left side of plan symbol, but has only one out port on the right side of plan symbol. Data therefore flows from left to right.

2) Each in port needs to be linked from either a processing or an input plan’s out port.

3) To link two plans, a plan has a basic data-flow symbol from the out port to another plan’s in port; to link many plans, a plan has a fork shape data-flow symbol from the out port to other plans’ in ports.

4) A dataflow is uniquely identified by the combination of source port name and destination port name for this dataflow. For example, a dataflow from the out port of plan1 to the in port of plan2 can be identified by (plan1:out, plan2:in).
Considering Rumbaugh’s principles, the visual notation of plan in Table 4-2 maps the plan concepts to a single symbol (clear mapping, [principle 1]). There is no significant overloading of symbols despite using the one symbol to represent different plan types (in, out, processing) (no overloading of symbols, [principle 2]). This is because there is no need to distinguish the plan types after the plan is mapped from an identified goal. In other words, plans can be regarded as being a single concept (rather than three concepts) and represented by using only one symbol. Both plan and goal diagrams have the same notation of dataflow, and by using the same symbol in both places we are being consistent (uniform mapping, [principle 3]). We are compliant with the principle 7 (consistent with past practice) because we use the plan symbol from existing notations. For other elements, we also reuse the existing notation of dataflow. However, we have to design the port symbols for plan because there is no existing notation for it. Although, in fact, there are existing notations for ports (e.g. in UML), these are not likely to be familiar to novice programmers.

Having a limited size of port embedded in a plan, it may be hard to read the name of a port when printing (looks good when printing, [principle 5]), faxing and copying (fax and copy well, [principle 6]). It is arguable that it may result in the distinction being too subtle (distinctions are not too subtle, [principle 9]) by having port symbols for both single value and a sequence of values. However, we believe that different port symbols capture a useful distinction (no overloading of symbols, [Principle 2]) Furthermore, the port name can be abbreviated by the combination of plan name plus the position on the left (in) or right (out) side of the plan (suppressible details, [principle 12]). However, in the more common case of plan network, the port name (in or out) can be simply omitted and identified by its position of left or right (common cases appear simple, [principle 11]). Moreover, there is only one plan symbol plus four port symbols and two data-flow direction line symbols in the plan notations. From that small number of seven symbols, there are simply no further issues resulting from the application of the rest of the principles such as principle 4 (easy to draw by hand), 8 (self consistent), and 10 (users can remember it).

We now give an example of a plan diagram corresponding to the goal diagram in Figure 4-7. Figure 4-10 shows an Input Plan that generates an output with a sequence of values, which become inputs to the Sum Plan. The Sum Plan inputs the dataflow and generates a single output value, which is also the input dataflow to the Output Plan. Finally, the Output Plan inputs the single value of the dataflow and displays it.
4.5 Evaluation of the Design of Visual Notation

We had looked at Rumbaugh’s principles when designing our visual notation of goals and plans (see right column in Table 4-3). Now, we consider Moody and van Hillegersberg’s (2009) criteria for the evaluation of our visual notation (see left column in Table 4-3). Initially, Moody (2009) had proposed a set of nine principles (see middle column in Table 4-3), not only for cognitive effectiveness in visual notation design, but also for evaluating, comparing, and improving existing visual notations. These principles were based on the theory of visual notations and empirical studies from different fields.

Furthermore, Moody and van Hillegersberg (2009) focused on five principles and applied these evidence-based principles to evaluate the visual syntax of UML 2.0. They emphasised evidence-based design of visual notations. In other words, they argued that the design of visual notations must show evidence of cognitive effectiveness when it is evaluated by the principles.

We observe that some of Moody’s principles can be combined into the new Moody and van Hillegersberg’s principles. We believe that the emphasis of using different visual variables (text plus graphics and dialects) in Moody’s principle 7 (dual coding) and principle 9 (cognitive fit) is similar to having the full range and capacities of visual variables in principle 6 (visual expressiveness). Considering the ability to represent details at different levels, although there is no correspondence in Moody and van Hillegersberg’s principles, the proposal to include hierarchy by Moody’s (2009) principle 4 (complexity management) relates to Rumbaugh’s principle 12 (suppressible details). However, there is no correspondence to Moody’s principle 5 from other two columns. Furthermore, there could be a conflict between Moody’s principle 8 and Rumbaugh’s principle 7, e.g. consistency with UML based on the past practice could lead to complexity. The comparison of visual notation principles by Moody and van Hillegersberg’s
We evaluate the effectiveness of our goal and plan visual notation by following the same criteria of Moody and van Hillegersberg’s (2009) evaluation of UML, which focuses on five major principles rather than the original nine principles. We consider each of Moody and van Hillegersberg’s principles, and for each principle, we explain it and then assess our notation against the principle. Following Moody and van Hillegersberg, the assessment is conducted by

<table>
<thead>
<tr>
<th>Moody and van Hillegersberg’s Principles</th>
<th>Moody’s Principles</th>
<th>Rumbaugh’s Principles</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Semiotic clarity (one-to-one correspondence between symbols and concepts)</td>
<td>1. Semiotic clarity (one-to-one correspondence between semantic constructs and graphic symbols)</td>
<td>Corresponds to: [principle 1] (clear mapping), [principle 2] (no overloading), [principle 3] (uniform mapping), and [principle 8] (self consistent).</td>
</tr>
<tr>
<td>2. Perceptual discriminability (symbols can be differentiated from each other)</td>
<td>2. Perceptual discriminability (be clearly distinguishable to each other)</td>
<td>Corresponds to: [principle 9] (distinctions are not too subtle), and also related to: [principle 4] (easy to draw), [principle 5] (looks good when printed), and [principle 6] (must fax and copy well using monochrome images).</td>
</tr>
<tr>
<td>3. Perceptual immediacy (have natural association with the concepts or relationship they represent)</td>
<td>3. Semantic transparency (appearance suggests meaning)</td>
<td>Corresponds roughly to: [principle 7] (consistent with past practice) [principle 10] (users can remember it), and also related to: [principle 11] (common cases appear simple).</td>
</tr>
<tr>
<td>4. Visual expressiveness (the number of different visual variables being used)</td>
<td>6. Visual expressiveness (use the full range and capacities of visual variables), 7. dual coding (use text to complement graphics), and 9. cognitive fit (use different visual dialects).</td>
<td>No correspondence (although does support: [principle 9] (distinctions are not too subtle), but potentially conflicts with: [principle 4] (easy to draw by hand), [principle 5] (looks good when printed), and [principle 6] (must fax and copy well using monochrome images).</td>
</tr>
<tr>
<td>5. Graphic parsimony (less is more)</td>
<td>8. Graphic economy (the number of symbols should be cognitively manageable)</td>
<td>Generalisation of: [principle 10] (users can remember it).</td>
</tr>
</tbody>
</table>

We evaluate the effectiveness of our goal and plan visual notation by following the same criteria of Moody and van Hillegersberg’s (2009) evaluation of UML, which focuses on five major principles rather than the original nine principles. We consider each of Moody and van Hillegersberg’s principles, and for each principle, we explain it and then assess our notation against the principle. Following Moody and van Hillegersberg, the assessment is conducted by
considering to what extent the notation meets the desired principle. This assessment is necessarily somewhat subjective since the principles are not precisely measurable.

1) Principle of semiotic clarity

This principle requires one-to-one correspondence between semantic constructs and graphical symbols. Otherwise, anomalies can occur in terms of symbol redundancy (multiple symbols for the same semantic construct), symbol overload (one symbol for different semantic constructs), symbol excess (a symbol does not represent any semantic constructs), and symbol deficit (a semantic construct is not represented by any symbol) (Moody & van Hillegersberg, 2009).

Considering our notation, there is no symbol redundancy issue due to the use of three different icons to represent three different types of goals in the visual goal notation because each symbol represent exactly one type of goal. Although the fork shape of dataflow can be represented by several basic arrows, we keep both data-flow symbols to make the common case simple, which agrees with the principle of perceptual immediacy. We argue that the basic data-flow arrow and the fork shape data-flow arrow represent two different types of dataflow. Thus, there is no redundancy issue to have both data-flow symbols. There is no symbol redundancy issue in plan notation, either.

Since we made the assumption that each goal has a single out port, we do not need to have named ports on goal diagrams, but only use the combination of goal and data-flow icons to present the linkage between goals in order to simplify the initial analysis. Thus, there is no symbol deficit issue by hiding the port in the goal visual notation. There is no symbol deficit issue in the plan visual notation, either.

As discussed earlier, having a separate symbol for input, output and processing plans is considered as symbol overload. The overloading of the plan symbol is relatively harmless because each plan is an atomic element from the plan library and also the plan network is directly mapped from the goal diagram. Thus, there is no need to follow the notation of goals in having three symbols for different types of plans. Moreover, there is no symbol overload issue in the goal notation. There is no excess symbol within the total ten symbols for goals and plans. Therefore, there is no redundancy, overload, excess, or deficit issues for these symbols to represent goals and plans.
2) **Principle of Perceptual Discriminability**

It is also required that symbols should be clearly and perceptually different from each other for easy and accurate discrimination (Moody & van Hillegersberg, 2009). In other words, each symbol should have a distinct visual distance to other symbols by having a unique value on at least one of the visual variables (e.g. shape, colour, and position).

Like UML, our visual notation includes two types of elements: nodes (two-dimensional graphical elements, e.g. goal, plan, and port) and links (one-dimensional graphical elements, e.g. dataflow). In order to enhance the discrimination, we also distinguish symbols within each type of element. For the node element, in order to create a visual distance between goals and plans, we have goals represented by the combination of triangle and rectangle(s), and plans by only rectangle with double lines on both horizontal sides which is similar to the shape of procedure used in flowcharts. We also have four different combinations of plan port symbols stand for either single or multiple values on the port as well as for either in or out function. Specifically, one rectangle or multiple rectangles are used to represents single or multiple values on the port, respectively. The arrow position on the left or right of the rectangle(s) is applied to represent the incoming dataflow to the in port or the outgoing dataflow from the out port, respectively.

For the link element, since dataflow links between goals and between plans, both single arrow and combination of arrows (fork shape) are used to present the linkages in terms of one-to-one and one-to-many. Both data-flow symbols are clearly perceptually distinguishable to represent the linkage to a single or multiple destinations of dataflow. No other visual variables (such as brightness, colour, or texture) are necessary for the dataflow.

3) **Principle of Perceptual Immediacy**

The graphical representation should have natural association with the concepts or relationship they represent. In other words, the appearance of visual representations should suggest their meaning (Moody & van Hillegersberg 2009).

The appearance of goal notation is a combination of rectangle and triangle(s). The triangle indicates the direction of dataflow. For example, according to a left-to-right reading direction, the Input Goal has a triangle on the right hand side indicating that there is a dataflow starting
from here towards the next goal. The appearance of a Process Goal indicates that it will receive dataflow from its left side and then it will send a new dataflow from its right side to the next goal. Similarly, an Output Goal’s appearance suggests it will receive a dataflow from another goal on its left side and it is also the end of this data-flow stream. Thus, the appearance of the Input Goal symbol suggests the beginning of the dataflow; the Process Goal symbol shows the intermediate procedure of dataflow; and the Output Goal indicates the final destination of a dataflow.

Additionally, the shapes of start (input) and end (output) goals can be joined together to form a rectangle through the matched triangles. And the shapes of Process Goals can not only be fitted in between Input and Output goals, but also be chained together with other Process Goals. Eventually, a goal diagram is constructed as an entire rectangle by three shapes of goal symbol from input (open), to process, to output (closed) shape. This indicates that the objective of program goal analysis for novices is to simply use these three different shapes of goals to work towards a rectangle shape in the form of a complete goal diagram. Moreover, the line with arrow illustrates the linkage and the direction of dataflow, and specifically the fork shape data-flow symbol represents clearly that one source dataflow has several copies of streams to different destinations.

On the other hand, the plan symbol uses the traditional procedure (sub-program) symbol with double lines on both left and right hand sides of a rectangle, which implies that a plan is also like a procedure and has more code details in it. Since three icons of goals and two data-flow symbols have the same direction from left to right, we assign the left side port(s) for dataflow into a plan (in port) and set the right side port for dataflow out of the plan (out port). Moreover, the distinguishing of in and out ports is based on a textual label, which is resolved by using a convention (left and right) to use orientation for both symbols of rectangle and arrow as additional cues to help the distinction. Finally, the symbol for ports uses a singular rectangle or stacked rectangles to represent either a singular value or a sequence of values on the port.

4) Principle of Visual Expressiveness

Ideally, a visual notation should have the full range and capacities of visual variables. There are eight visual variables commonly used to enhance visual expressiveness in order to improve perceptual discriminability (Moody 2009). They are categorised into “planar variables” and
“retinal variables” (Moody & van Hillegersberg, 2009). The planar variables include both horizontal and vertical positions in the two spatial dimensions such as (x, y) representing an actor and where an activity is performed. The retinal variables include shape, size, colour, brightness, orientation, and texture. To represent different concepts, it is common to use various basic shapes such as circle, rectangle, triangle or combination of them. However, the same shape can also be used to present different concepts by using a different size (small or big), colour (red, yellow, blue, or mixture of them), brightness (low, medium, high), textural patterns, and orientation angles.

In the evaluation of UML by evidence-based principles, Moody and van Hillegersberg (2009) considered for each diagram which of the eight available visual variables were used. Similarly, Table 4-4 shows visual expressiveness in our goal and plan diagrams. Besides different shapes between goal and plan, we use different shapes for the three types of goals. Different shapes are also used to distinguish ports with single or multiple values. We have used orientation visual variables together to present the data-flow directions between goals and between plans. For example, the direction of dataflow presents the linkage of plans from out port to in port. Planar variables are considered as the most powerful visual variables (Moody & van Hillegersberg, 2009). However, we use them in the form of a left-to-right layout avoiding the noise of complexity to novice programmers. We did not use size, colour and other retinal variables because we consider Rumbaugh’s Principle 4, 5 and 6, but we do use conventional orientation from left to right and focus on ensuring that symbols can be easily drawn by hand and reproduced in black and white. Two data-flow symbols, basic and fork shape, have employed the orientation visual variables to represent the direction of dataflow as well as the linkage of goals or plans.

<table>
<thead>
<tr>
<th>Diagram Type</th>
<th>(X, Y)</th>
<th>Size</th>
<th>Brightness</th>
<th>Colour</th>
<th>Texture</th>
<th>Shape</th>
<th>Orientation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal Diagram</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Plan Network</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

5) **Principle of Graphic Parsimony**

It has been suggested that the size of a visual vocabulary (the number of symbols) leads to graphical complexity and produces barriers to learning and using visual notation (Moody & van Hillegersberg 2009). We have only applied several symbols and visual variables in our visual
notation. There are only five symbols in the goal notation and seven symbols for the plan notation including two reused data-flow symbols in the visual notation for both goals and plans.

In the evaluation of UML, Moody and van Hillegersberg (2009) counted for each diagram type the number of distinct symbols that could be used in that diagram type. Similarly, Table 4-5 summarises the graphic complexity of diagrams according to the number of different graphical symbols used in each diagram. A goal diagram has a total of five symbols: three for goals and two for dataflow (see Table 4-1). A plan network has a total of seven symbols: one for plan; four for ports; and two for dataflow (see Table 4-2). Comparing to other notation such as UML, the graphic complexity is much lower. In other words, we have a limited number of symbols in our visual notation, which reduces the graphic complexity of diagrams.

<table>
<thead>
<tr>
<th>Diagram</th>
<th>Graphic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal Diagram</td>
<td>5</td>
</tr>
<tr>
<td>Plan Network</td>
<td>7</td>
</tr>
</tbody>
</table>

In summary, we have evaluated our visual notations for goal and plans against the Moody and van Hillegersberg’s principles and now briefly summarise the key findings. Although several symbols are used to represent different goals, dataflow and port, there is no redundancy in the symbols, because each symbol represents only one concept for the purpose of having clearly distinguishable symbols. Furthermore, the combination of goal and dataflow can be used indicate a unique port. Therefore, there is no symbol deficit issue from hiding the port at the goal level. On the other hand, there is no symbol overloading issue from using only one plan symbol because each plan is based on a goal identified previously. In other words, our notations meet the principles of both semiotic clarity and perceptual discriminability. Moreover, we use different visual variables such as shape and orientation for having diverse appearance of goals, plans, dataflows, and ports particularly to indicate their meaning in order to meet the principles of perceptual immediacy and visual expressiveness. Finally, we only use a total of ten symbols (including two repeated symbols for dataflow) in both goal and plan diagrams. The limited number of symbols meets the principle of graphic parsimony.
4.6 Summary

The analysis of plan combination leads us to design our visual notation of goals and plans using a dataflow paradigm. Whereas control flow deals well with programming code details at a low level, dataflow is a good starting point for programming design at a high level. We conclude to design programs using goals and plans in dataflow paradigm and to develop program code details in control flow paradigm. Since there are already existing notations for control flow based programming, it is unnecessary for us to develop the notation of goals and plans in control flow paradigm.

In the data-flow paradigm, our visual notations for goals and plans were designed and evaluated based on the theory and principles of visual notation. The development of the visual notation of goals enables a program to be described as a group of related goals, namely a goal diagram. Then, relevant plans that conform to the goal diagram can be identified from the existing plan library to achieve these goals, where the dataflow between plans represents their linkage. All the linkages and plans build up a plan network as a solution of the program at the design level. In other words, after analysis of the goals shown in a goal diagram, a set of plans can be selected from a plan library to construct a plan network.

The visual notation of goals and plans is also novice-oriented. Only a few symbols or icons (in total ten) are used in our visual design notation for goals and plans. The port icons are suppressed from the goal diagram and then introduced in the plan network. Students do not have to learn a lot of icons at the beginning. Instead, they learn to use these limited numbers of icons gradually.

Through the evaluation by the principles of visual notations in software engineering, we believe that our visual notation of goals and plans meets mostly the established criteria. Having defined and evaluated a graphical notation of goals and plans in dataflow paradigm, we now turn to implement the visual notation of plans using a visual programming language.
Chapter 5 A Data-flow Framework for Using Plans

In Chapter 4, we presented our design notation for goals and plans. Specifically, we adopt a data-flow based representation so that the interleaved plans are executable before being merged. We modelled goals and plans as consuming and producing dataflow i.e. sequences of data. A dataflow links two goals or two plans. In order to make plans to be executable without merging, buffers are needed to carry data for every dataflow. Specifically, a buffer is needed for each in port (but not for each out port) of a plan. Our purpose now is to develop a framework in BYOB to allow data-flow plan networks to be executed.

The data buffer is a temporary place to carry sequences of data between plans. Recall in Chapter 4, dataflow from one plan can be sent to multiple plans (e.g. a sequence of data from Input Plan can be sent to both Sum Plan and Count Plan). In other words, dataflow from an out-port of a plan can be sent to one or many in ports of other plans. Therefore, we choose to keep values of dataflow received on the in port of a plan in a data buffer. Consequently, a set of buffers are used for every dataflow between every two plans in a plan network, specifically carrying values of dataflow on every in port of plans in the plan network (see Figure 5-1 and 5-2).

On the top of Figure 5-1 (i. *Plan network diagram*), the plan network represents that a dataflow is produced from Plan1 and sent from the out port of Plan1 (*plan1:out*) to two in ports of Plan 2 and Plan3 (*plan2:in* and *plan3:in*) respectively. The results from both Plan 2 and Plan3 are sent from their out ports (*plan2:out* and *plan3:out*) to two in ports on Plan4 (*plan4:in1* and *plan4:in2*), respectively. Finally, the results from Plan4 are sent from its out port (*plan4:out*) to the in port of the next plan. For example, in order to process the average of a sequence of data, Plan1 inputs these data and produces a dataflow to both Plan2 and Plan3. Plan2 consumes the dataflow from Plan1 and produces a new dataflow to Plan4 containing the sum of data whereas Plan3 consumes the same dataflow from Plan1 and also produces a new dataflow containing the count to Plan4. After consuming both dataflows of sum and count, Plan4 produces a further dataflow with the result of the average to next plan.

The bottom of Figure 5-1 (ii. *The linkage by plan ports and dataflow*) describes the linkage of the plan network by plan ports and dataflow between ports. Each plan is represented having two types of plan-ports (in and out) in two columns. An arrow is used to represent dataflow
from the in port of a plan to the out port of another plan. For example, the first arrow in the table represents a dataflow from the in port of Plan1 to the out port of Plan2. However, it is more complicated when a plan has multiple in ports. For example, since Plan4 has two in ports, it needs extra information e.g. port names (in1 and in2) to differentiate the two in ports in Plan4 as destinations for the two dataflows from Plan2 and Plan3. All the in ports are bold in the table indicating the dataflow will be carried in data buffers.

On the top of Figure 5-2 (i. The abstract model of plan linkage by UML), the plan linkage is represented in an abstract data model by UML. It means the entire program plan linkage (Linkages) consists of many linkages (Linkage) of pairs of adjunct port names (outport_name and inport_name). A Linkage is associated with one buffer list (BufferList) to keep the dataflow from out port of one plan to in port of another plan. In other words, we record a linkage from out port to in port. However, the buffer is associated with the in port. For the fork shape dataflow (one source versus many destinations), many pairs of records are needed in the buffer for the port names to be associated with the linkages. The Linkage using pairs of port names only represents the dataflow relationship between plans. It has no indication of what values each dataflow has. Therefore, we need BufferList to hold values of all the dataflow in a program.
Figure 5-2 Data structure and example of physical model in BYOB for plan linkage
The bottom of Figure 5-2 illustrates the implementation of buffers corresponding to the example in Figure 5-1 based on the above abstract model of plans and dataflows. Firstly, on the left side of the bottom part in Figure 5-2 (ii. List of plan linkages in BYOB), the Linkage in the UML diagram is implemented by a list in BYOB. Groups of two plan port names (see contents in dash boxes in the figure) are listed to describe every data-flow linkage from source to destination in the above abstract model, i.e. from out port to in port between two plans, e.g. [plan1:out] and [plan2:in]; [plan1:out] and [plan3:in]; [plan2:out] and [plan4:in1]; [plan3:out] and [plan4:in2] etc. Then, on the right side of the bottom part in Figure 5-2 (iii. Buffer contents by a list of lists in BYOB), the BufferList in the UML is implemented by a list of lists in BYOB. A group of lists are set up as buffers for data-flow items on every in port according to the in port name. For example, a list that is identified by the in port name of Plan2 [plan2:in] contains all the data items of dataflow on the in port of Plan2 such as item1, item2, etc. Each in port has a corresponding buffer. There are two buffers for Plan4 because it has two in ports (plan4:in1 and plan4:in2).

In Chapter 3, we have also discussed the use of a visual programming language (BYOB) for the research in this thesis. BYOB has a limited number of complex data structures. We have to keep different data types, e.g. number and text, in the same list instead of separating data-flow values and plan information. Therefore, we put the contents of a buffer and the buffer’s identifier in the same list in BYOB. The first item of each list is the name of the in port, which is used for identification. And then dataflow is saved from the second item of each list. A collection of list buffers are also nested in one list for a program. In other words, buffers for a program are a list of lists. Meanwhile, another list is also used to keep the records of plan linkage by alternating out and in port names.

In Section 5.1, we introduce the implementation of a data-flow framework by using data buffers for the dataflow between plans as well as a description of the plan network linkage in terms of plan port names. In Section 5.2, we then describe the implementation of library blocks to support receiving dataflow from buffers into a plan and storing dataflow from a plan into buffers. In Section 5.3, we introduce the development of plan blocks within the framework. Finally, in Section 5.4, we illustrate how our unmerged plan blocks mapping from a plan network diagram can be executed in the data-flow framework.
5.1 Initialising Linkages and Data-flow Buffers

We aim to develop a programming environment in which plans are executable before merging. In the data-flow paradigm, each plan consumes and/or produces sequences of data, i.e. dataflow. We define the following constraints on the relationship between plans:

1) Two plans are linked by the dataflow from the out port of one plan to the in port of another plan;
2) The data from the out port of one plan can flow to the in port of one or multiple plans; and
3) Each in port receives data from exactly one out port.

That is to say, there is a one-to-many relationship between out ports and in ports of plans. Therefore, we only need to record dataflow on the in ports rather than out ports for the linkages between two plans. We use buffers, i.e. lists, to temporarily keep dataflow on every in port. We then use plan ports (in ports and out ports) as interface of linkage between plans to receive and send dataflow.

We need a list of identifiers, e.g. names, for the plan in ports in order to generate relevant buffers. Since two plans are linked by using a pair of plan ports (an out port of one plan and an in port of another plan), each linkage of two plans has two items, one for the out port name and another for the in port name. Therefore, two items are added into a list of linkages by using both out port and in port names to describe a dataflow between two plans. We then create a buffer list corresponding to each of the in ports in the list.

For the example of summing a sequence of input values in Figure 4-9, data-flow buffers and plan linkage for the plan network diagram are implemented by using lists in BYOB as shown in Figure 5-3. Both lists are developed as part of an independent component in the format of a “Sprite” in BYOB. The first list of “Sprite 1 links” (see the left list in Figure 5-3) contains pairs represented by consecutive out port and in port names that are used for the plan linkage. For

---

19 In order to have the infrastructure of implementing buffers be importable into new projects, we have wrapped it up in a BYOB feature called a “Sprite”. In BYOB, a Sprite is an object that performs actions according to user’s instructions, such as moving, jumping or computing. After the component has been imported as a new “Sprite” into a new project, extra text of “Sprite 1” has been added to the name of buffer and link lists (e.g. “Sprite 1 buffers” and “Sprite 1 links”).
example, the first item “values1:out” is an out port (which happens to be for Input Plan) that is linked to the second item “sum:in” (an in port of Sum Plan). The two port names represent the plan linkage by a dataflow from Input Plan to Sum Plan.

![Diagram of data-flow buffers and plan links](image)

**Figure 5-3 Example of data-flow buffers and plan links**

Therefore, all the even numbered items in “links” are the in port names. The list containing nested lists, “Sprite 1 buffers”, can be created according to the names of the even numbered items in the list of “Sprite 1 links” (see the bottom part of Figure 5-3). Each nested list in the list “Sprite 1 buffers” is a buffer that corresponds to an in port (i.e. item 2, 4, 6, etc. in the linkage list), which contains the name of the associated in port in the first item of the buffer. For example, the first nested list in the list of “Sprite 1 buffers” has the name of “sum:in” as its first item. Similarly, for the second dataflow from Sum Plan to Output Plan, the second nested list in the list of “Sprite 1 buffers” has the name “output1:in” as the first item. This representation is redundant (since the in port name is stored in both links and buffers). This redundancy is used for error checking (see Section 5.4).
For creation and manipulation of the described data structure, we developed three plan linkage blocks in BYOB, “Begin Links”, “Link [out-port] to [in-port]”, and “End Links” (see Figure 5-4). The first block, “Begin Links”, initialises both “links” and “buffers” lists. This block removes all the information from the two lists, links (string/text) and buffers (a mixture of string/text and numbers), as shown on the top left of Figure 5-4. The second block, Link [out-port] to [in-port], adds two items into list “links”, one for the out port name, and another for the in port name. The third block, End Links, sets up an individual list for each in port from the even numbered items of “links” and also adds the in port name to the first item of each new list to identify individual buffer for the port (add (item (i + 2) of links) to (list)). And finally, the block, delete (all) of (links), will remove all the port names from the plan linkage list, “links”. Another block, delete (all) of (buffers), will remove all the nested buffer lists for in-ports from the data-flow buffer list, “buffers”.

20 The block, “delete (all) of (links)”, will remove all the port names from the plan linkage list, “links”. Another block, “delete (all) of (buffers)”, will remove all the nested buffer lists for in-ports from the data-flow buffer list, “buffers”.
21 In the BYOB Block Editor, a block, “add (A) to (B)”, adds an item (A) to the end of a list (B), making the list longer. The item can be either a variable or a list to be nested in the list.
22 In the BYOB Block Editor, the block, script variables (i), declares a local variable, e.g. i, which is only used inside the block. The block, set (A) to (B), assigns a value (B) to a variable (A), i.e. A := B. For example, “set i to 0” means “i := 0”. Since each program has a different number of lists in the buffers, the block, “delete (all) of (buffers)”, must remove all the nested lists in the buffer list. Furthermore, the block, set (list) to ([list]), sets up an empty list to variable list for initialisation of each new nested buffer list. The block, change i by (2), means (i := i + 2).
it puts every new list for an individual in port (with in port name in its first item) into the list “buffers” (\texttt{add (list) to (buffers)}).

\section*{5.2 Library Blocks for Implementation of Plans}

In the previous section, we have introduced the implementation of plan linkage by using buffers to keep dataflow that is received by each plan. Specifically, we developed three types of plan linkage blocks to keep track of linkages in a “\textit{links}” list and maintain the dataflow on individual in port into a nested “\textit{buffer}” list. We now focus on sending and getting data to/from the buffers. Therefore, a plan can be implemented (see Section 5.3) by receiving dataflow from its in port and/or sending dataflow to its out port. We also consider providing feedback if the sending and receiving is from an incorrect data-flow buffer by taking advantage of existing records from the list “\textit{links}”.

\subsection*{5.2.1 Sending a value from plan out port to buffers}

In Section 5.1, we discussed how two plans are linked by a dataflow from the out port of one plan to in port of another. We only need to use one buffer to keep the dataflow received in the in port. When we write a plan, we use an infrastructure block (called “\textit{SEND DATA}”, and defined below) (see top of Figure 5-5) that sends a data item from the plan’s out port. The internal implementation of the “\textit{SEND DATA}” block is responsible for using the data structures to deliver the data item to the appropriate in ports on a linked plan.

Inside the Block Editor in Figure 5-5, the sending data block receives a plan out port name id (e.g. \texttt{values1:out}) from its parameter. The default value of [out-port] (“\textit{Copy the default Out-Port name here from its original plan block}”) suggests to use the out port name where the sending data block is held by a plan block. Firstly, it searches for the associated in port names of destination from the Linkage (\textit{links}) according to the out port name (\texttt{if out-port name = id, set (in-port name) to (item (index of links + 2) of links)}). Secondly, it moves to BufferList (\textit{buffers}) getting the first item of individual buffer list (\texttt{set (list name) to (item 1 of item (index of buffers + 1) of buffers)}). It repeats searching through all the lists until the end of the buffers. Thirdly, if the first item from current buffer list matches the in port name identified in the Linkage by the first step (\texttt{in-port name = list name}), the value on the parameter of SEND DATA block is then added to this buffer list (\texttt{add (value) to (item index of buffers + 1 of buffers)}. 

121
Finally, for the “fork” shape dataflow, one out port has many associated in ports. There will be more than matching in port for the given out port, and each matching in port will have its own buffer in the BufferList. Therefore, it repeats the above three steps and may end up sending
the item to more than one buffer list until there is no more associated in port names in the Linkage (links) according to the out port name.

For example, in Figure 5-3, if the SEND DATA block in the Input Plan needs to send an item (e.g. 5) from the plan out port (values1:out) to Sum Plan, first of all, it finds the in port name (sum:in) from the next item of values1:out in the list of “Sprite1 buffers”. It then tries to locate the buffer list which has its first item as sum:in in the list of lists Sprite 1 buffers. Finally, it adds the value 5 to the located buffer list after item sum:in.

We set up a mechanism for checking the validation of out port name of the plan. If the out port name in the “SEND DATA (value) [out-port]” block does not match any out port name in the list of “links” (“out-port name = id” is false, i.e. “valid = false”), an error message will be given. This mechanism can catch human errors when they were made in the programming process. For example, when using plan blocks, the feedback can indicate the inconsistency of port names between the one in the plan block and that in the linkage block ([Link [out-port] to [in-port]]), which is filled by the user. When replacing a parameter of local variable in the plan block by an argument of plan out port name in the expanded plan code details, the feedback can also indicate the inconsistency of the out port names between the new replacement and that in the “Link” block. More details about the application of this feedback mechanism are discussed in Section 5.4.

5.2.2 Getting dataflow from buffers to a plan

We now consider how a plan can get a data item from an in port that is kept in a list in the buffers. Before getting data from an in port, we also need to check whether there is data contained on the in port’s buffer. Therefore, in order to get dataflow from buffers to a plan, we need to create two blocks. One is for testing if there is data on a plan’s in port. Another is for getting a data item from the plan’s in port.

1. Testing if there is data on a plan’s in port

In previous discussion, dataflow on a plan’s in port is kept in a list as a buffer, which is identified by the in port name. All the lists are nested in a list called “buffers”. In order to check whether there is data contained in a buffer list for the relevant in port, we developed a data testing block, NO MORE DATA [in-port], to provide a Boolean result of false when there is data
on the plan in port (see the top of Figure 5-6). In this data testing block, the default value of [in-port] (“Copy the default In-Port name here from its original plan block”) suggests to use the in port name from its host plan block.

Figure 5-6 Implementation of testing if there is data in a plan in port
Recall from Section 5.1, that every buffer in the BufferList (e.g. Sprite 1 buffers) has as its first item an in port name as its unique identifier. Therefore, if the length of a buffer list is equal to one, there is no data in the list. Otherwise, there is at least one datum in the buffer list. Inside the Block Editor in Figure 5-6, if the length of a buffer list is equal to one (length of item (index of buffers + 1) of buffers = 1), there is no data in the list, but only the name of plan in port, and the Boolean value of the testing block, “NO MORE DATA [in-port]”, is true. If the length of a buffer list is greater than one, there is at least one datum in the buffer list and the Boolean value of the testing block is false. For example, in order to test if there is any data on Sum Plan’s in port (sum:in), the block is represented as NO MORE DATA [sum:in]. That is to say, the code searches for the buffer by an in port name as id, e.g. sum:in. The program searched for the same in port name from every buffer list until it found a list having an in port name sum:in (if (name of list = id)). It then starts to test if the length of the list is equal to one. Finally, a Boolean value is sent out by the “report” block.

As with sending a value, we also set up a mechanism for checking the validation of the in port name of the plan. If the in port name in the “NO MORE DATA [in-port]” block does not match any in port name in the list of “buffers” (“name of list = id” is false, i.e. “valid = false”), an error message will be given. This mechanism can also be used to catch human errors when using plans as well as when replacing the parameter by port names in the expanded plan code details (see Section 5.4).

2. Getting a datum from a plan in port

Similar to testing dataflow on a plan in port, the getting data-flow block is developed to retrieve a datum from a buffer list that is used to keep dataflow for the plan in port. The block, GET DATA [in-port], retrieves a data item from a plan’s in port, more exactly to remove a data item from a buffer list for the in port (see Figure 5-7). In this block, the default value of [in-port] (Copy the default In-Port name from its original plan block) indicates to use the in port name from its host plan block.

Since the first item in a buffer list is the plan in port name, the dataflow actually start from the second item. After testing that there is at least one datum existing in the buffer list by using the testing data block, NO MORE DATA? [in-port], a datum can then be obtained from the

23 There is no parameter needed for the report underneath the Block Editor because a “report” block inside the editor sends a Boolean type of value to it.
second item on the list to a block local variable “value”. This item is then removed from the list, and the value returned.

Figure 5-7 Implementation of getting a datum from a plan in port
Inside the Block Editor in Figure 5-7, the parameter in port in the block \( \text{GET DATA } \text{[in-port]} \) is used as id to search for a buffer list from BufferList (e.g. Sprite 1 buffers). When the buffer list for the in port is located (if name of list = id), a datum value can be then retrieved from the second item in the list because the first item is the in port name (set value to item (2) of (index of buffers + 1) of buffers). Next, this datum is also deleted from the buffer list (delete (2) of (index of buffers + 1) of buffers). If there are more existing data afterwards on the list, the rest of items will be promoted to fill the removed vacancy. For example, assume a buffer list for the in port on Sum Plan contains items of \( \text{sum:in}, 1, 2, \) and \( 3 \). The block \( \text{GET DATA } \text{[sum:in]} \) will search for a list that contains “\( \text{sum:in} \)” on its first item according to the parameter of sum:in. After being founded (name of list = id, i.e. \( \text{sum:in} = \text{sum:in} \)), the second item (i.e. \( 1 \)) is retrieved for report and removed from the list. The third item (2) becomes the new second item for next getting data process and so on. The list becomes \( \text{sum:in}, 2, \) and \( 3 \), and the value 1 is returned (report value\(^{24} \)

Once again, we also applied the mechanism for checking the validation of the in port name of the plan. If the in port name from the “\( \text{GET DATA } \text{[in-port]} \)” block does not match any in port name in the list of “buffers” (“name of list = id” is false, i.e. “\( \text{valid = false} \)”), an error message will be given.

\[ \text{5.3 Develop Plan Blocks in the Data-flow Framework} \]

Previously, we have discussed building three plan linkage blocks (i.e. \( \text{Begin Links, Link } \text{[out-port] to [in-port]} \), \( \text{End Links} \)) in Section 5.1 to support linkage between plans by using plan ports and data-flow buffers. In Section 5.2, we discussed developing three data-flow blocks to support sending and receiving dataflow within a plan by using a plan port (in port or out port) to connect to data-flow buffers. We assemble the plan linkage blocks and the data-flow blocks together as a framework. Using this data-flow framework, a plan block can be built by using these three data-flow blocks (\( \text{SEND DATA, NO MORE DATA?}, \) and \( \text{GET DATA} \)) to deal with dataflow on plan ports. Furthermore, a set of plan blocks can be linked by three types of linkage blocks and they are executable without merging because the dataflow within them is implemented by a list of buffers (see Section 5.4).

---

\(^{24}\) There is a parameter “value” for the report underneath the Block Editor because a variable “value” is used to return a value of string or number (rather than a Boolean value) for this block.
In the data-flow framework, a plan block can be implemented in three different types: Input Plan, Process Plan, and Output Plan. We have identified six different plan patterns for building plan blocks, one for Input Plan, four for Process Plan, and one for Output Plan. In the daft-flow paradigm, the complexity of plan pattern is determined by its generating and consuming dataflow. We notice that the plan patterns for Input Plan and Output Plan are less prone to change than those for Process Plans. The basic Input Plan and Output Plan only deal with one dataflow, either generating or consuming. However, a Process Plan deals with at least two dataflows, both consuming and generating. Although we define that each Process Plan only generates one dataflow, it may consumes many dataflows, e.g. a Divide Plan needs two dataflows for dividend and divisor. Moreover, a new dataflow can be generated by the Process Plan either at the time when consuming individual data of a dataflow, or after consuming the entire dataflow. In this section, we focus on how to build individual plan blocks using these data-flow blocks that we have developed. We now discuss how to implement Input Plan and Output Plan first and then Process Plans.

5.3.1 Building Input Plan and Output Plan Blocks

The Input Plan block produces a dataflow to its plan out port from input data. The Output Plan consumes a dataflow on its plan in port and then displays the results. Although both Input and Output Plan blocks have dataflow in different direction (out and in), they have only one port (either out port or in port) for each of them. Notably, the multiple usages of Input Plan and Output Plan blocks can be differentiated by different plan port names. Therefore, the users do not need to define their own Input and Output Plan blocks. This is why we consider the Input and Output Plan blocks to be part of the infrastructure.

1. Building an Input Plan block
An Input Plan block receives input data from keyboard and stops when a sentinel data (e.g. -1 or 99999) is entered. It produces a dataflow to its out port. A program may have multiple Input Plans, and so we use a number to differentiate dataflow from different Input Plans (e.g. values1, values2). Therefore, the interface of Input Plan block includes an out port name (e.g. values1:out) and a sentinel value (e.g. -1) (see the top figure in Figure 5-8). If the Input Plan block is used a second time by the same program, its out port can be renamed as (e.g.) “values2:out”.

128
Inside the Block Editor in Figure 5-8, both parameters (out-port and sentinel) are local variables. They get argument values from default values of the plan port name (out-port = values1:out) and sentinel (sentinel = -1). Using a loop in the plan, every input value except the sentinel (-1) is sent to the plan out port (SEND DATA (Number) [out-port]). Except for the sending data block, the rest of the Input Plan details are actually the traditional Input pattern in control flow paradigm. When using this plan block, novices only need to know that the dataflow has been sent to the plan’s out port rather than to consider the process of keeping the dataflow in a buffer. For example, when Input Plan accepts data of 1, 2, 3 and -1 from keyboard, it sends numbers of 1, 2, and 3 to its out port values1:out by using the block, SEND DATA (Number) [out-port (= values1:out)]. As discussed in Section 5.2.1, data 1, 2, and 3 on the out port values1:out are actually added to the relevant in port’s buffer list.

2. Building an Output Plan block

An Output Plan block displays a result or several results after receiving a dataflow from its plan in port. Similarly, considering a program may have multiple Output Plans, we use a number to identify values from different Output Plan blocks (e.g. output1, output2). Therefore, the interface of the Output Plan block includes an in port name e.g. output1:in (see the top block in Figure 5-9). If the Output Plan block is used a second time in the same program, its in port name is named (e.g.) “output2:in”.

129
Inside *Block Editor* in Figure 5-9, the parameter (in-port) is a local variable. It gets an argument value from the default value of plan in port name (*in-port = output1:out*). Using a loop in the plan (*repeat until NO MORE DATA? [in-port]*) every data value is obtained from the plan in port (*set (Result) to GET DATA [in-port]*) and then displayed (*say ... for 2 secs*). When using this plan, novices only need to understand that the displayed dataflow is obtained from its plan in port rather than to consider the process of getting the dataflow from a list of buffer. For example, if *Output Plan* has results of 1, 2, and 3 on its in port *output1:in*, it displays 1, 2, and 3 sequentially.

### 5.3.2 Building Process Plan Blocks

A *Process Plan* can consume one or more dataflows and produces a new dataflow. The new generated dataflow can have only a single datum or a sequence of data. Subsequently, a *Process Plan* block may have one in port and one out port. It may also have multiple in ports and one out port. Therefore, we need to develop four types of process plan blocks: 1) consuming one dataflow with any number of values from one in port and producing a single datum to its out port; 2) consuming one dataflow from one in port and producing a sequence of data to its out port; 3) consuming data from multiple in ports and producing a single datum.
to its out port; and 4) consuming data from multiple in ports and producing a sequence of data to its out port.

Since a Process Plan block has functions of both consuming and producing dataflow, plus that of processing the data, it needs to have five basic components: 1) initialising variables, e.g. \( \text{sum} = 0 \) or \( \text{count} = 0 \); 2) repeating to test if there is more data on the in port; 3) getting dataflow from the plan in port; 4) processing the dataflow; and 5) sending dataflow to the plan out port. We now discuss process plan blocks that consume one dataflow first and then discuss those that consume multiple dataflows.

1. Building a Process Plan that consumes one dataflow and generates a singular datum

When a Process Plan block only has one in port and one out port, its plan interface will be “Plan Name [(plan name):in] [(plan name):out]”. That is to say that a Process Plan block has its in port first and then its out port. The default port name is the combination of the plan name and the function of the port (in or out). For example, the interface of the Sum Plan which sums a sequence of values, is “Sum Plan [sum:in] [sum:out]”.

When the Process Plan block only generates a singular datum value, the pattern of five components for the process plan block is as shown in Table 5-1. Within the loop, every single datum from the in port is processed. Then a single datum is sent to the out port. Therefore, the SEND DATA block is located outside of the loop.

<table>
<thead>
<tr>
<th>Interface of Plan Block</th>
<th>Plan Name [in-port = plan:in], [out-port = plan:out]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>Initialise variable</td>
</tr>
<tr>
<td>2)</td>
<td>repeat until &lt; NO MORE DATA? [in-port] &gt;</td>
</tr>
<tr>
<td>3)</td>
<td>set (variable) to ( GET DATA [in-port])</td>
</tr>
<tr>
<td>4)</td>
<td>Process variable according to the logic</td>
</tr>
<tr>
<td>5)</td>
<td>SEND DATA (variable) [out-port]</td>
</tr>
</tbody>
</table>

An example of this pattern is the implementation of a Sum Plan block to compute the sum of a sequence of numbers (see Figure 5-10). On the interface of the Sum Plan block, both plan port names are set by the default values (in-port=\text{sum:in} and out-port=\text{sum:out}). In the contents of the Sum Plan block, the five plan components are: 1) initialising variables (set (Sum) to 0); 2) using a loop to find out if there is more data on the plan’s in port (repeat until NO MORE DATA
13) getting the dataflow (set (Number) to GET DATA (in-port)); 4) processing the data (set Sum to Sum + Number); and 5) sending the final result to the plan’s out port (SEND DATA (Sum) (out-port)).

![Block Editor](image)

**Figure 5-10 Example of Process Plan block consuming a dataflow, but generating a single datum**

The *Sum Plan* details are similar to the traditional *Sum Plan* pattern in control flow paradigm that an expert would use. When using this plan block, novices only need to know that the dataflow has been sent to its plan in port and the result of sum can be retrieved from its out port. For example, when data of *sum*:in, 1, 2, and 3 are on the buffer list of the *Sum Plan*’s in port, each data item (e.g. 1, and then 2, and then 3) will be removed from the list and be accumulated (e.g. 0 + 1 = 1, 1+2 = 3, 3+3 = 6) until there is no more data. Finally, the result of *Sum* (e.g. 6) is sent to the plan’s out port *sum*:out.
Similarly, a Count Plan block can also be built by the same pattern in Table 5-1 (see the first part in Figure 5-11) to count the number of items in a sequence of data. The Count Plan has two ports, count:in and count:out. The plan details show the pattern of how the data is repeatedly obtained from the in port (count:in) and then counted (set (Count) to (Count) +1). At the end, the processed result is sent to the out port (count:out).
A “Max” plan is also an example of this pattern. It is built to find the maximum number in a sequence of numbers \((\text{maximum} = -99999, \text{if number} > \text{maximum}, \text{then maximum} = \text{number})\) (see the second part in Figure 5-11).

Moreover, a Search Plan block can also be built by following this pattern to find if a particular target value is in a sequence of numbers (see Figure 5-12). The result of either “found!” or “not found” will be the single datum on its out port.

![Figure 5-12 Another example of the same pattern for building Search Plan block](image)

2. Building a Process Plan that consumes one dataflow and generates a sequence of data

When a Process Plan block consumes data from a single in port and produces a sequence of data to its out port, the pattern of five components for the process plan block is similar to the previous one, but the last step of sending data to out port \((\text{SEND DATA (variable) (out-port)})\) must be put inside the loop body (see Table 5-2).

For example, to produce a sequence of even numbers from a sequence of input values, the Even Number Plan block is created as in Figure 5-13. In Figure 5-13, the plan block gets every data from its in port \((\text{even:in})\) by “set (Number) to GET DATA [in-port]”. If the data is an even data \((\text{Number mod 2} = 0)\), it will be then sent to the plan out port by “SEND DATA Number [out-port]”. For example, when the in port has data of 1, 2, 3, 4, 5, and 6, it will send 2, 4, and 6.
Table 5-2 The Pattern of Process Plan: consuming dataflow from its in port, and generating a new dataflow to its out port

<table>
<thead>
<tr>
<th>Interface of Plan Block</th>
<th>Plan Name [in-port = plan:in], [out-port = plan:out]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>Initialise variable</td>
</tr>
<tr>
<td>2)</td>
<td>repeat until &lt; NO MORE DATA? [in-port] &gt;</td>
</tr>
<tr>
<td>3)</td>
<td>set (Number) to (GET DATA [in-port])</td>
</tr>
<tr>
<td>4)</td>
<td>Process (Number) to (variable) according to the logic</td>
</tr>
<tr>
<td>5)</td>
<td>SEND DATA (variable) [out-port]</td>
</tr>
</tbody>
</table>

Figure 5-13 Example of Process Plan block consuming a dataflow, and also generating a dataflow

3. Building a Process Plan that consumes multiple dataflow and generates a single datum

We have discussed how to build up a plan block to consume only one dataflow from one linked plan block. However, some Process Plans need to consume multiple dataflows from different plans. In other words, they must have multiple in ports to receive dataflow from multiple linked plan blocks. The pattern of five components for consuming multiple dataflows and producing a single datum (see Table 5-3) is similar to the first pattern in Table 5-1. In this pattern, the loop stops when any in port to the plan is empty. In other words, if the process plan gets no data from any of its in ports, the loop will stop. Consequently, if there is no data on one of the in ports, the process plan will abandon the rest of data on the other in ports. Because the pattern gets data from every in port equally, the retrieved data can be processed
validly. However, the amount of data retrieved from each in port depends on the shortest number of input data items on an individual in port.

For example, a Sum of Multiplying Plan block (see Figure 5-14) has two in ports to receive two sequences of items (e.g. 1, 2, 3 and 2, 5, 4), item1 and item2. It multiplies every two items, (item1 * item2), and sums the multiplying results of every two items (e.g. 1x2 + 2x5 + 3x4 = 2 + 10 + 12 = 24). Finally, the result of the Sum (e.g. 24) is sent to its plan out port. If data on the second sequence of items is different in the length of items (e.g. 2, 5, and 2, 5, 4, 8), the result will take the shorter length of items from two sequences (e.g. 1x2 + 2x5 = 2 + 10 = 12, and 1x2 + 2x5 + 3x4 = 2 + 10 + 12 = 24 ). Comparing to the Sum Plan block in the first pattern, it merges the function of multiplying two items with the function of Sum Plan. We use this example only for demonstration of the new pattern in Table 5-3. Alternatively, the process of sum of two multiplying items can also be implemented by using both Multiplying Plan and Sum Plan together. The implementation is discussed in the second example in Chapter 6.

Table 5-3 The Pattern of Process Plan: consuming multiple dataflow from its in ports, but generating only one single datum to its out port

<table>
<thead>
<tr>
<th>Interface of Plan Block</th>
<th>Plan Name [in-port1 = plan:in.item1] [in-port2 = plan:in.item2] [in-port3=... ] [out-port = plan:out]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>Initialise variables</td>
</tr>
<tr>
<td>2)</td>
<td>repeat until &lt; NO MORE DATA? [in-port1] &gt; or &lt; NO MORE DATA? [in-port2] &gt; or &lt;...&gt;</td>
</tr>
<tr>
<td>3)</td>
<td>set (variable1) to ( GET DATA [in-port1]) set (variable2) to ( GET DATA [in-port2]) set ....</td>
</tr>
<tr>
<td>4)</td>
<td>Process variables according to the algorithm variable = f(variable1, variable2, ...)</td>
</tr>
<tr>
<td>5)</td>
<td>SEND DATA (variable) [out-port]</td>
</tr>
</tbody>
</table>
4. Building a Process Plan that consumes multiple dataflow and generates a sequence of data

It is similar to the second pattern in Table 5-2 when building a Process Plan that consumes multiple dataflow and produces a sequence of data. In the pattern of the five components for the process plan block (see Table 5-4), the last step of sending data to out port (SEND DATA (variable) [out-port]) must also be put inside the loop body.

Table 5-4 The Pattern of Process Plan: consuming multiple dataflow from its in ports, and generating a new dataflow to its out port

<table>
<thead>
<tr>
<th>Interface of Plan Block</th>
<th>Plan Name [in-port1 = plan:in.item1] [in-port2 = plan:in.item2] [in-port3=...] [out-port = plan:out]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1)</td>
<td>Initialise variables</td>
</tr>
<tr>
<td>2)</td>
<td>repeat until &lt; NO MORE DATA? [in-port1] &gt; or &lt; NO MORE DATA? [in-port2] &gt;</td>
</tr>
</tbody>
</table>
| 3)                      | set (variable1) to (GET DATA [in-port1])  
set (variable2) to (GET DATA [in-port2])  
set ......                                                                    |
| 4)                      | Process variables according to the algorithm variable = f(variable1, variable2, ...)                    |
| 5)                      | SEND DATA (variable) [out-port]                                                                         |

For example, a Multiplying Plan block (see Figure 5-15) has two in ports to receive two sequences of items (e.g. 1, 2, 3 and 2, 5, 4), item1 and item2. It multiplies every two items (“set (Result) to Number1 * Number2”) (e.g. 1x2, 2x5, 3x4), and directly sends the results to an out
port ("SEND DATA Result [out-port]") before getting the next two items for multiplying (i.e. the resulting output is the sequence of values 2, 10, and 12).

Similarly, a Dividing Plan block can be built to receive two sequences of items (dividend and divisor) (see Figure 5-16). It generates a sequence of results for the pairs of input data. A warning message (Divisor must not be zero 0!) will be generated if a divisor is zero.

Using three data-flow blocks (testing, getting and sending data blocks), a Process Plan can be built by following one of the four patterns (from Table 5-1 to Table 5-4). A group of plan blocks can be provided as a plan library for novices to implement their design in plan network diagrams and work toward the final program code. Therefore, we can provide to novices a data-flow framework with a plan library together with three plan linkage and three data-flow blocks for building new plan blocks. In other words, novices can use existing plan blocks or build their new plan blocks to develop programs in our data-flow framework.

Figure 5-15 Example of Process Plan block consuming multiple dataflows, and generating a dataflow
5.4 Example of How Plans Blocks Work in the Data-flow Framework

Previously, we have built up our data-flow framework for developing programs by using goals and plans in a visual programming environment. This framework includes 1) plan linkage blocks (see Section 5.1), 2) data-flow blocks (sending, testing, and getting) (see Section 5.2), and 3) a small library of plans blocks (see Section 5.3). We now discuss how this data-flow framework supports the implementation of programs from a design in the form of a plan network diagram.

We start from the example of summing a sequence of input values in Section 5.1. The question can be described as follows:

Write a program that will read in integers and output their total value. Stop reading when the value -1 is input.

According to the plan network diagram (see top part in Figure 5-17), firstly, we need to use three plan blocks from the plan library in the data-flow framework (Input Plan, Sum Plan, and Output Plan). (If a plan block does not exist in the framework, according to the process plan...
patterns in Section 5.3, a new plan block can be built by using three data-flow blocks together with other control flow blocks.) Secondly, we need to link these plan blocks by the plan linkage blocks (Begin Links, Link [out-port] to [in-port], End Links) and fill in the port names in the linkage blocks (Link [out-port] to [in-port]) based on the dataflow connected port names in the diagram. We need two Link blocks for the two dataflows between plans in the plan network diagram. The parameters of both Link blocks are the port names on both ends of the dataflow arrows from left to right. For example, the parameters of the first Links block are [values1:out] and [sum:in]. And the parameters of the second Link block are [sum:out] and [output1:in].

In the bottom part of Figure 5-17, the implementation using plan blocks and their linkages is shown on the left side. The linkage blocks are put before the plan blocks. All the port names in the linkage blocks are filled from left to right following the dataflow direction in the plan network diagram. The same port name in both linkage block and plan block must be consistent (e.g. value1:out and value:out are inconsistent if both were in linkage block and plan block respectively).

![Figure 5-17 Example of implementation plan diagram by using plan blocks and plan linkage](image)
Both information of linkage and buffers for dataflow in two lists (Sprite 1 links and Sprite 1 buffers) plus two variables (Number and Sum) are shown on the right side. The process that novices use to map from plan network diagram to plan blocks is discussed in Section 6.2.3.

Figures 5-18 and 5-19 show snapshots during the program’s execution. In Figure 5-18, after the sequence of test data (e.g. 1, 2, 3 and -1) has been entered, the buffer list “sum:in” keeps the dataflow (e.g. 1, 2, and 3, but not the sentinel -1) from Input Plan block on the in port of Sum Plan. The two variables, Number and Sum, show the last value in the dataflow when the variables are used to transfer dataflow.

After a sentinel value (-1) is entered, the Input Plan completes executing, and dataflow on the in port of Sum Plan is processed by the plan (1 + 2+ 3 = 6) and removed from the buffer list (sum:in) (see the top list of left Sprite 1 buffers in Figure 5-19). The result (6) is sent from out port of Sum Plan to the in port of Output Plan (output1:in) in the buffer list under item “output1:in” (see the bottom list of left Sprite 1 buffers in Figure 5-19). Finally, the result (6) is removed from the buffer list identified by the first item “output1:in” and then displayed by the Output Plan block. Therefore, the implementation of the plan network diagram by using plan blocks and their linkage provided by the data-flow framework is executable without merging the details of plan blocks.

Figure 5-18 Example of dataflow in buffer list for the in port of plan
In Section 5-1, we have set up a mechanism to provide feedback when a user made a mistake in typing port names. If a user has typos in an out port, for example, spelling “values1:out” from *Input Plan* as “values:out”, an error message from sending data-flow block (cf. Section 5.2.1) will be displayed as shown in Figure 5-20. It reminds the user to check the consistency of spelling for out port “values1:out” by comparing out port names in both *Link* and *Plan* blocks. Similarly, if a user has typos in an in port name, for example, spelling “sum:in” from *Sum Plan* as “sum”, two types of messages from both testing and getting data-flow blocks (cf. Section 5.2) will be displayed as in Figure 5-21. Both messages remind the user to check the consistency of spelling for in port “sum:in” by comparing in port names in both *Link* and *Plan* blocks.
Previously, we have seen plans blocks are executable after being linked in the data-flow framework. We now turn to explore how different plan code details work together in the data-flow framework. The local variables (e.g. “in-port” and “out-port” in a plan block) receive argument values from the plan’s interface (e.g. port name or constant value). Through expanding the plan’s details, the interface has been removed. Therefore, in order to maintain the argument value in the plan details we have to manually replace these local variables with the corresponding argument values from the plan’s interface. We expand all the plan blocks and replace their plan port parameters (in port and out port) and other local variables (e.g. sentinel) by arguments of plan names (e.g. `sum:in` and `sum:out`) and data values (e.g. -1 or 9999) (see Figure 5-22). For example, for the expanded details from *Input Plan* block, the local variable “*out-port*” is replaced by the plan’s out port name `output1:out` whereas three local variables named as “sentinel” are substituted by value -1. Similarly, for the expanded details from *Sum Plan* block, the local variable “*in-port*” is replaced by the plan’s in port name “`sum:in`” whereas the local variable “*out-port*” is substituted by the plan’s out port name “`sum:out`”. Finally, for the expanded details from the *Output Plan* block, the local variable “*in-port*” is replaced by the plan’s in port name “`output1:in`” (the process details of expanding are
discussed in Section 6.2.4). Since a plan block is made by its plan code details, results from plan code details are the same as those from the plan block (See Figure 5-19).

Figure 5-22 Example of plan details working in the Data-flow Framework

5.5 Summary

To support using plans in the data-flow paradigm, we developed a data-flow framework by using a visual programming language—BYOB. Firstly, we used a list of buffers to keep dataflow on the in port of each plan as well as a list to record the linkages between plans. We developed three blocks for specifying the links between plans. Secondly, we developed another three blocks for use inside a plan which send, test, and retrieve dataflow on the plan port. Thirdly, we introduced how to use these data-flow blocks together with traditional control flow blocks to develop three types of plan blocks: input, process, and output plan blocks. Specifically, we introduced the implementation of process plan blocks in four patterns. Finally, we illustrated how plan blocks and their linkages were used to implement a plan network diagram, and how the (unmerged) plans can be executed due to the buffers provided by our data-flow framework. In the next chapter, we present a programming process for using goals and plans within this data-flow framework to develop programs.
Chapter 6 The Programming Process for Programming with Goals and Plans

In Chapter 4, we presented visual design notations to represent goals and plans using the data-flow paradigm. Based on general principles for designing a visual notation, we developed and evaluated our visual design notations for goals and plans. We proposed to start program design with visual notations using the data-flow paradigm and then shift to program code using the control flow paradigm. Further in Chapter 5, we developed a data-flow framework for the implementation of the visual plan notation by using a visual programming language — BYOB. We also described how to build up plans in the plan library using the data-flow framework so that novices can not only retrieve plans from the provided library, but can also create and add their own plans into the library.

We now turn to providing support to novices with a detailed process for using our visual notation in programming. We first briefly explain why we need a process for novice programming (when starting from goals and plans). We then present a process for programming, which includes feedback in each phase.

6.1 Why is a Programming Process Needed?

Recall from Chapter 1 the claim that “novice programmers know the syntax and semantics of individual statements, but they do not know how to combine these features into valid programs” (Winslow 1996, page 17). In other words, novices lack problem-solving strategies and plans (de Raadt, 2008; Winslow, 1996). Therefore, we need not only to introduce goals and plans, but also to provide strategies and some form of guidance for programming.

Caspersen and Kölling (2009) advocated providing novices with a detailed step-by-step process for programming. They argued that “If we do not explicitly teach the programming process, we end up with two groups of students: those who cannot cope with the challenge of development and those who can discover their own implicit process” (Caspersen & Kölling, 2009, p3). This indicates that without a process we might also end up with two groups of students: those who are successful and those who are unsuccessful. This is undesirable, because the group that failed may have been able to do better if they were given a process, rather than being left to
attempt to discover one on their own. Furthermore, the process that the “successful” students discovered may not be a good one. Therefore, to avoid repeating a similar situation to that observed by Caspersen and Kölling, we need to explicitly provide a step-by-step process as an appropriate form of guidance.

Caspersen and Kölling were not the first to advocate providing novices with a process. Much earlier, Gantenbein (1989) proposed a process model for teaching programming that was based on the classical software life cycle model and inspired by a process for writing English essays. This model includes five steps: 1) defining the problem; 2) selecting an approach; 3) designing a solution; 4) implementing the solution; and 5) testing the implementation. He believed that by identifying the steps of the programming process, students can understand what is required of them at each point in the process. However, Gantenbein did not define a detailed process for programming and did not apply structures that experts use such as schemas or plans.

Pattis (1990) also proposed that the programming process needs to be broken down into a series of well-defined steps, and suggested that it is important to provide feedback from each step. Providing feedback at each step was considered to be critical in giving students confidence. In fact, feedback is at the heart of test-driven development, and Janzen and Saiedian (2008) proposed to improve teaching by using a “test-driven learning” (TDL) approach. The TDL approach can be classified as either test-first or test-last: “Test-first refers to writing automated unit tests immediately before new functional units are written, and test-last means the tests are written immediately after new functional units are written” (Janzien & Saiedian, 2008, p533). In other words, the tests can be written either before or after the program is developed. They conducted an experimental evaluation of their approach. In their experiment, there were two years of students (CS1 and CS2). In each year of students, there were two groups (Group A and Group B). Each group was asked to complete two projects (Project 1 and Project 2)—see Table 6-1. Group A students from each year were asked to complete the first project with a test-first approach and the second project with a test-last approach. Conversely, Group B students were asked to complete the first project with a test-last approach and the second project with a test-first approach. Although students in the Group As switched from the test-first to test-last approach, they consistently showed better performance in terms of testing results for projects, productivity (volume of code versus time spent), and project grades on both projects compared with those in the Group Bs who started
with a test-last approach. Through a post-experiment survey, Janzen and Saiedian also found that after being exposed to both approaches, students preferred the test-first approach. Overall, their results indicate that the test-first approach can increase students’ performance. However, neither Pattis nor Jansen and Saiedian provided a detailed process that could be taught to novices.

Table 6-1 Janzen and Saiedian’s Evaluation of the TDL Approach

<table>
<thead>
<tr>
<th>Students</th>
<th>Group As</th>
<th>Group Bs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Project 1</td>
<td>Project 2</td>
</tr>
<tr>
<td>CS1</td>
<td>Test-first</td>
<td>Test-last</td>
</tr>
<tr>
<td>CS2</td>
<td>Test-first</td>
<td>Test-last</td>
</tr>
</tbody>
</table>

Moreover, a number of detailed processes for teaching novices have been proposed, for example, *Programming by Numbers* (Glaser et al., 2000) and *TeachScheme* (Felleisen et al., 2004). As discussed in Chapter 2, both approaches provide a clear process for creating the smallest components of functions, using stepwise refinement. However, they are in the progress compelled by data, i.e. data-driven and more suited to functional programming languages than to mainstream procedural languages. Another process that has been proposed is STREAM (Caspersen & Kölling, 2009), which aims to teach novices a process for object-oriented programming that follows an incremental development approach. The process includes six steps: Stubs, Tests, Representation, Evaluation, Attributes, and Methods (STREAM). The key aspect of this process is incremental development from solving a simpler part of the problem to a gradually larger part of the problem to eventually the entire problem. It is also referred to as stepwise improvement rather than conventional stepwise refinement. The stepwise refinement is from abstract to concrete based on the refinement calculus. A preliminary evaluation was done by observing and taking notes of students’ processes of programming during the examination. The results indicated that students can follow the process of writing program code, but ignore the importance of testing even though a test-first approach has been suggested. Since this process focuses on object-oriented programming, it is not adequate for our purpose of using goals and plans. In conclude, some of the above approaches emphasise the test-first approach, but do not provide a detailed process. Others accentuate details of the process, but only support providing feedback at the end of process, rather than at the end of each step or phase in the process. Finally, and for our purposes, perhaps most importantly, none of the proposed processes discussed above included the use of goals and plans (or a visual programming language).
In summary:

- It is important to provide a detailed process of programming.
- Test-first (or test-early) is a good practice within the process.
- Although some processes have been proposed, none of them support goals and plans (or a visual language).

### 6.2 A Programming Process Using Goals and Plans

We present a detailed process that uses a “test-early” approach and supports goals, plans and a visual programming language. We divide the process into six phases. Each phase includes several steps and is designed to provide feedback to engage students with the process. Our programming process is summarised in Figure 6-1, and consists of six phases: (1) devising test cases; (2) analysing goals and plans; (3) mapping the plan network to BYOB using plan blocks; (4) expanding plan blocks; (5) merging the expanded plan details; and (6) simplifying the merged details. Each phase includes three or four steps.

In the description that follows, we illustrate the programming process using three examples. The first example demonstrates a simple case with single input, single process, and single output goals. The second example illustrates a more complex case with multiple process goals. The third example shows the most complicated case with multiple input, process, and output goals.

#### 6.2.1 Phase 1. Devising Test Cases

The first phase, devising test cases, consists of three steps: 1) identify the output and input and specify inputs of test cases; 2) decide how to compute the output from input; 3) calculate outputs from specified example inputs of test cases.

**1. Identify the output and input and specify inputs of test cases**

First of all, the output of a program can be identified from the requirements of the question. This is also the final goal being achieved by the program. Next, the input of a program can be identified from the conditions and constraints provided in the question. Moreover, examples of input test cases need to be specified. For example:
1. Devise test cases:
   1) Identify the output and input and specify inputs of test cases;
   2) Discover how to compute the output from input;
   3) Calculate outputs from specified inputs.

2. Analyse goals and plans:
   1) Draw goal diagram, identifying goals and joining them by data-flow links;
   2) Map to plan network, identifying ports for each dataflow;
   3) Desk-check plan network.

Are the results the same as those from previous phase?

3. Encode plan network using BYOB plan blocks:
   1) Apply plan blocks from library or build new ones;
   2) Link plan blocks using ports for each dataflow;
   3) Test.

Are the results the same as those from previous phase?

4. Expand plan blocks and change parameters:
   1) Replace plan blocks by plan code details found inside each plan block;
   2) Replace every parameter of each plan with its plan port;
   3) If the same plan is used multiple times, the same variables in the different occurrences of the plan are renamed differently, e.g. `<name1>&<name2>`;
   4) Test.

Are the results the same as those from previous phase?

5. Merge plan code details:
   1) Collect all the initialization blocks.
   2) Combine loops that share the same dataflow.
   3) Remove loop control if driven by a dataflow including only a single value;
   4) Test.

Are the results the same as those from previous phase?

6. Simplify the Merged Details:
   1) If two variables share a dataflow to/from the same port, rename the second variable to be the same as the first one;
   2) Remove all the scaffolding blocks;
   3) Test.

Are the results the same as those from previous phase?

Figure 6-1 Programming process by using goals and plans in our visual programming environment
**[Example 1]** Write a program that will read in integers and output their total value. Stop reading when the value -1 is input.

In this example, the output is the “total value” while the inputs are a sequence of integers. The integer “-1” is a sentinel indicating the last input, but is not processed. Inputs of test cases are specified at the beginning. An example of specified inputs for one test case is 1, 2, 3, and -1. Another test case is 3, 4, 5, 6, and -1. The following two examples are more complicated input-process-output cases.

**[Example 2]** Write a program that will read in integers and output their average. Stop reading when the sentinel value (-1) is input.

This example was originally used by Soloway (1986) to analyse goals and plans. It is also used for our evaluation in Chapter 7. Here, it is used to demonstrate how to solve the problem with multiple process goals. The output is the average while the inputs are integers. The test inputs given for the previous example can also be used in this example. Another test case is 2, 3, 7, 8, and -1.

**[Example 3]** Write a program that will read in pay rates until a sentinel value (-1) is input, and then read in working hours until a sentinel value (-1) is input, and then calculate wages (the formula for wage is wage = rates * hours). The program outputs the total wages and the number of employees.

With this example, we demonstrate how to solve a problem with multiple input and output goals together with multiple process goals. Two outputs are identified as “total wages” and “count of people” while two inputs are recognised as “pay rates” and “working hours”. The specified test case inputs for rates and hours are 15, 20, 30, -1, and 20, 40, 20, -1, respectively. Another test case inputs for rates and hours are 20, 30, 40, 50, -1 and 40, 40, 20, 20, -1, respectively.

2. **Discover how to compute the output from input**

It is critical to discover the relationship between input and output. At this stage, the goal is not to develop a general method or algorithm, but merely to understand how to manually process the input of the previously specified test cases to produce outputs. We argue that this manual
processing, specifically using simple test cases, promotes novices’ understanding of the conversion from input to output.

For the above Example 1, the computation process is to calculate the sum of all the input numbers except the sentinel in order to get the output. For example, when the input data is 1, 2, 3, and -1, through the computation process of calculating a sum \((1+ 2 + 3 = 6)\), the output result of total is 6. For Example 2, the computation process is more complicated. Firstly, it needs to compute sum and count from the input data and then compute the average. For example, when the input is the same data as in Example 1, the output will be \((1+2+3)/(1+1+1) = 6/3 = 2\). For Example 3, where the input has two sequences of data, the computation process is even more complicated to output the sum of wages and the count. For the sum of wages, firstly, it needs to get wages for each person from pay rates and hours, and then to accumulate these wages. For example, when the inputs of rates and hours are 15, 20, 30, -1, and 20, 40, 20, -1, the outputs of sum and count are \((15*20 + 20*40 + 30*20 = 1700)\) and \((1+1+1 = 3)\), respectively.

3. Calculate outputs from specified inputs

For each test case, the output answer must be calculated in order to create a ground truth for comparison in the following phases in the process. The three tables below (Table 6-2, Table 6-3, and Table 6-4) give the output answers of tests cases for the above three examples.

**Table 6-2 Test Cases for Example 1**

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1)</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data:</td>
<td>1, 2, 3, -1</td>
<td>3, 4, 5, 6, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td>6</td>
<td>18</td>
</tr>
</tbody>
</table>

**Table 6-3 Test Cases for Example 2**

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1)</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data</td>
<td>1, 2, 3, -1</td>
<td>2, 3, 7, 8, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

**Table 6-4 Test Cases for Example 3**

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1)</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data</td>
<td>rates</td>
<td>15, 20, 30, -1</td>
</tr>
<tr>
<td></td>
<td>hours</td>
<td>20, 40, 20, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td>sum</td>
<td>1700</td>
</tr>
<tr>
<td></td>
<td>count</td>
<td>3</td>
</tr>
</tbody>
</table>
6.2.2 Phase 2. Analysing Goals and Plans

1. Draw a Goal Diagram

In the second phase of analysis, a goal diagram is completed to represent every intermediate stage or goal that needs to be achieved to solve this problem. The diagram must include three types of goal icons (Input, Process and Output) to represent the different stages of processing. The linkage between goals is represented by the flow of data between them. The dataflow also constrains the order of goals (i.e. starting from the source goal to destination goal). In general, the development of the goal diagram is an incremental refinement process, in which the processing goals are hierarchically decomposed.

As noted in Chapter 4, by convention data flows from left to right, so all goal diagrams begin with an Input goal on the left, and finish with an output goal on the right, with processing goals in between. In fact, the number of input and output goals are determined by the input and output specified in the previous step: each distinct input (e.g. rates, hours) has a distinct Input goal, and each distinct output (e.g. sum, count) has a distinct output goal. In general, the development of the goal diagram is an incremental refinement process, in which the processing goals are hierarchically decomposed. Students start developing goal diagram from an Input-Process-Output top level goal pattern. They decompose each goal into sub-goals hierarchically. For example, from Input Goal, when students find the program needs to input two sequences of different data, such as hours and pay rates for computing wages, they will need to have two goals decomposed from the Input Goal, naming Input1 and Input2.

For Example 1, based on the initial phase and the input-process-output pattern, three goals (input, sum, and output) are identified and the order or sequence to achieve these goals is represented and linked by a dataflow (see Figure 6-2).

![Figure 6-2 Three basic goals for Example 1](image)

For Example 2, although this program includes one input, and one output, it has multiple processing goals, which may need hierarchical refinement. Initially, a “compute average” goal
is introduced. However, to achieve this goal, it is necessary to calculate the sum and count first and then divide the results from the sum and count to get the average. Hence, the “compute average” goal can be decomposed into three goals: sum, count, and divide, where both sum and count receive the same dataflow from the input goal (in parallel), and send their results to the divide goal (also in parallel). Finally, the result of the divide goal is sent to the output goal (see Figure 6-3). After a dataflow is sent by the Input Goal, a fork shape dataflow is used to represent a duplicate copy of the dataflow from Input Goal to Sum Goal as well as to Count Goal. Next, these process two goals (Sum Goal and Count Goal) are linked to two in ports on the Divide Plan. Finally, the result of the Divide Goal are sent to the Output Goal.

![Figure 6-3 A goal diagram for Example 2](image)

For Example 3, according to the test cases in Phase 1, two input goals to input two sequences of values for rates and hours are identified. Similarly, from test cases, two goals for outputs of sum and count are also identified. Subsequently, two process goals for computing sum and count are also needed before they can be output. Also, according to the manual process of wages in Phase 1 (the formula for each wage is wage = rates * hours), a process goal is also identified.

Based on the manual process in Phase 1, the five identified goals are organised in a goal diagram for Example 3 in Figure 6-4 (repeated from Figure 4-9). Firstly, two input goals (Input1 Goal and Input2 Goal) are linked to two in ports on the Multiply Goal that computes the wages individually by multiplication. Then, the results from the Multiply Goal are sent to two process goals (Sum Goal and Count Goal) through a “fork” shape dataflow. Next, these two process goals produce the sum and count of these wages and send results to two output goals (Output1 and Output2), respectively. Finally, two output goals display both the sum of wages and the count of payments independently.
Once goals are refined to a level where they are sufficiently fine-grained, they can be mapped to plans in a one-to-one manner (i.e. each goal becomes a plan), resulting in a plan network diagram. For teaching novice programming, a sufficiently fine-grained decomposition of goals means that the decomposed goals correspond to BYOB plan blocks in a provided plan library or that they can be implemented simply. We acknowledge that it may be challenging for novice programmers to recognise when to stop the refinement of goals. As we have discussed in the above step, the manual process with test cases in Phase 1 is the foundation for supporting novices in identifying goals and understanding the process. In the teaching process (see Chapter 7), a sub-goal labelling approach is also used to support identifying goals. We propose that every plan block that we provided in our plan library is atomic, which has the basic process function. Although we introduce how to build up students’ own plan blocks, our plan library contains most of the plans that are needed in this research. Other complex processes can be built up by merging these atomic plan blocks. Therefore, students stop their goal decomposition when they can find relevant plan block in the plan library, which is matching the goal that students used in their manual process or labelled in the sub-goal labelling. Alternatively, they may find similar plan block and modify it for relevant application, e.g. modifying maximum value plan to minimum value plan, and multiplying plan to dividing plan. Otherwise, they may have to develop a new plan block by using the data-flow blocks to link control flow code segment under the plan block patterns discussed in Chapter 5.

The goal diagram is mapped to a plan network by replacing the notation of goals with that of plans (see Figure 6-5). Importantly, the port name (a combination of the plan name and of the word “in” or “out”) of each plan is uniquely identified in each plan for the purpose of linking between plans by dataflow. As we have discussed in Chapter 4, we use a convention where in ports are on the left of a plan and out ports are on the right. We use a single box to represent a port which can send or receive a dataflow with only one datum. We also use a multiple layer

---

Figure 6-4 A goal diagram for Example 3 with multiple input-process-output goals
box to represent a port which can send or receive a dataflow with a sequence of data. So, novices need to work out which dataflow are multiple values and which are single values. This is done based on their manual process with test cases in Phase 1. For example, according to the manual process in Table 6-2, the dataflow from Input Plan to Sum Plan is a sequence of values. After being processed by Sum Plan, the dataflow become a single value from Sum Plan to Output Plan. Importantly, every port name in each plan block must be uniquely named in order to link plans using dataflow. Otherwise, dataflow may be linked to unrelated plans.

![Figure 6-5 A plan network diagram for Example 1](image)

For Example 2, the goal diagram is mapped to a plan network by replacing the notation of goals with that of plans (see Figure 6-6). As discussed in Chapter 4, each processing plan normally has two ports, one in and one out. In cases where a plan has multiple incoming dataflow, its graphical representation shows multiple in ports with different names. Specifically, in order to distinguish two in ports on one plan, we add different extension name to each in port name. For example, the two in ports of the Dividing Plan are identified as Dividing:in.dividend and Dividing:in.divisor.

![Figure 6-6 A plan network for Example 2](image)

Similarly, for Example 3, the Process Wage Plan (which can be implemented by the Multiply Plan in the plan library) receives two dataflow from two Input Plans. It also sends the same dataflow to two plans, Sum Plan and Count Plan, in the fork shape of dataflow. Finally, two different results, Sum and Count are sent to two Output Plans.
3. Desk-check the Plan Network

A desk-check is applied to verify dataflow passing between every two linked plan ports. The final results of dataflow from output plans are compared with the output results from test cases in the first phase in order to confirm that the plan network diagram is accurate. A desk-check table is used to test whether or not this phase of analysis is correct. We illustrate this using Example 1. Table 6-5 shows the desk-check table for Example 1. The table consists of two parts. The first part contains the first three rows of the table: Test Cases, Testing Data, and Output Answers. Its contents are based on the test cases specified in the first phase.

The second part of the table comprises the rest of the rows, and is based on the plan network. It records the dataflow through the plan network. Each row represents one port within the plan network. The cell within the “in” row of a plan is filled with a copy of the data from the relevant “out” row, i.e. the “out” port that is linked to that “in” port. For example, the “in” row for the Sum Plan is simply a copy of the “out” row of the Input Plan, since the Input plan’s out port is linked to the Sum plan’s in port. The cells within the “out” row of a plan are filled by generating output from its input, for example, the “out” row for the Sum Plan is the sum of its inputs.

Table 6-5 Desk-check of dataflow in plan ports for Example 1

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1) (For Example: )</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data</td>
<td>1, 2, 3, -1</td>
<td>3, 4, 5, 6, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td>6</td>
<td>18</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plans:</th>
<th>Ports:</th>
<th>Dataflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>out</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td>1, 2, 3</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>6</td>
</tr>
<tr>
<td>Output</td>
<td>in</td>
<td>6</td>
</tr>
</tbody>
</table>

156
The analysis phase of the process has been successfully completed when the results in the last row (Output Plan) are the same as those in the row of Output Answers (the third row) for each test case. The desk-check in this phase is part of the evaluation and testing schedule for the whole programming process. As students proceed through the whole process in Figure 6-5, they maintain a summary of completed tests (see Table 6-6). In this summary, the first three columns are filled according to the results from Table 6-5. The remaining columns correspond to later phases in the process. After each phase, the results from testing are recorded and compared to results from the earlier phases in the process. Comparing the results phase-by-phase determines whether the programming process can continue to the next phase.

Table 6-6 Summary of Test Schedule for Example 1

<table>
<thead>
<tr>
<th>Test Case(1)</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Cases</td>
<td>1, 2, 3, -1</td>
</tr>
<tr>
<td>Testing Data</td>
<td>6</td>
</tr>
<tr>
<td>Output Answer (Phase1)</td>
<td>6</td>
</tr>
<tr>
<td>Results from Analysis (Phase2)</td>
<td>6</td>
</tr>
<tr>
<td>Results from Design by plan blocks (Phase3)</td>
<td>6</td>
</tr>
<tr>
<td>Results from expand plan details (Phase4)</td>
<td>6</td>
</tr>
<tr>
<td>Results from merged details (Phase5)</td>
<td>6</td>
</tr>
<tr>
<td>Results from final program (Phase6)</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 6-7 illustrates a desk-check table that is used to test whether or not the second phase of analysis is correct for Example 2 by checking the records of dataflow through every port. For example, after entering the test data for Test Case (1), a dataflow (1, 2, and 3) is sent from the out port of the Input Plan to the in ports of both Sum Plan and Count Plan. Consequently, two dataflow, one from the out port of Sum Plan (6) and another from the out port of Count Plan (3), are sent to the two in ports of Dividing Plan, respectively. Finally, a dataflow with the average (6/3 = 2) is sent from the out port of the Dividing Plan to the in port of the Output Plan for displaying. The analysis phase of the process has been successfully completed when the results in the last row of Output Plan are the same as those in the row of Output Answers in the table.
Table 6-7 Desk-check table of dataflow for computing an average (Example 2)

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case (1) (For Example:)</th>
<th>Test Case (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data</td>
<td>1, 2, 3, -1</td>
<td>2, 3, 7, 8, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plans:</th>
<th>Ports:</th>
<th>Dataflow</th>
<th>Dataflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>out</td>
<td>1, 2, 3</td>
<td>2, 3, 7, 8</td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td>1, 2, 3</td>
<td>2, 3, 7, 8</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>Count</td>
<td>in</td>
<td>1, 2, 3</td>
<td>2, 3, 7, 8</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Dividing</td>
<td>in.dividend</td>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>in.divisor</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Output</td>
<td>in</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 6-8 demonstrates how a desk-check table is used to test whether or not the second phase of analysis is correct for Example 3. The analysis phase of the process has been successfully completed after the value in the in port of the first Output Plan are the same as the output answers of sum; and the value in the in port of the second Output Plan are the same as the output answers of count for each test case.

Table 6-8 Desk-check table of dataflow for computing wages (Example 3)

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case (1) (For Example:)</th>
<th>Test Case (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data</td>
<td>rates 15, 20, 30, -1</td>
<td>20, 30, 40, 50, -1</td>
</tr>
<tr>
<td></td>
<td>hours 20, 40, 20, -1</td>
<td>40, 40, 20, 20, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td>sum 1700</td>
<td>3800</td>
</tr>
<tr>
<td></td>
<td>count 3</td>
<td>4</td>
</tr>
<tr>
<td>Plans:</td>
<td>Ports:</td>
<td>Dataflow</td>
</tr>
<tr>
<td>Input (1)</td>
<td>out(1)</td>
<td>15, 20, 30</td>
</tr>
<tr>
<td>Input (2)</td>
<td>out(2)</td>
<td>20, 40, 20</td>
</tr>
<tr>
<td>Multiply</td>
<td>in.item1</td>
<td>15, 20, 30</td>
</tr>
<tr>
<td></td>
<td>in.item2</td>
<td>20, 40, 20</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>300, 800, 600</td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td>300, 800, 600</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>1700</td>
</tr>
<tr>
<td>Output (1)</td>
<td>in</td>
<td>1700</td>
</tr>
<tr>
<td>Count</td>
<td>in</td>
<td>300, 800, 600</td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>3</td>
</tr>
<tr>
<td>Output (2)</td>
<td>in</td>
<td>3</td>
</tr>
</tbody>
</table>

During the phase of goals and plans analysis, dataflow from input to process to output goals has been identified and represented in diagrams using the visual notations. Plans have been
linked by a dataflow from the out port of a plan to an in port of another plan. The goal diagram is the foundation for the plan diagram. The former is used for initial analysis and is focused on “what” to achieve; the latter is used for implementation and is concerned with “how” to achieve the solution by using existing components through specified linkage. The plan diagram not only provides guidance for choosing plan blocks from a provided library for the implementation of the diagram in later phases, but also illustrates both the plan order and linkages to organise these selected plans. After confirmation by desk-checking, the plan network diagram can be implemented using plan blocks in the next phase.

6.2.3 Phase 3. Encoding the Plan Network Using BYOB Plan Blocks

Following the confirmation of the correctness of the goal and plan analysis by desk-checking, the diagram of the above plan network can be mapped to an executable program by using BYOB plan blocks. In this third phase, our data-flow framework (see Chapter 5) is used to implement the plan network diagram. The data-flow framework includes a plan library with a number of plan blocks as well as linkage blocks to connect the applied plan blocks, which are implemented in the BYOB visual programming language. Plan blocks can be selected from our provided plan library. Otherwise, students need to build their own plan blocks. In the second step of this phase, the links between the plan blocks are implemented using the data-flow framework. Each “Link” block of the data-flow framework declares that a link exists between two plan blocks and represents a dataflow from one plan block to another.

The process for mapping a plan diagram to BYOB blocks is fairly straightforward and mechanical. Each plan icon is replaced by a plan block in BYOB, and every dataflow is mapped to a block of the data-flow framework (Link [out-port] to [next plan in-port]) to link an out port to an in port. The details of encoding the plan network diagrams for the above three examples are described as follows.

Note that these linked plan blocks are executable. That is to say, students do not need to wait to get test feedback until these plan blocks are merged in phases 5 and 6. In our data-flow framework, the infrastructure (a set of “Link” blocks) implements dataflow using buffers (see Chapter 5) to carry dataflow from one plan block to another so that the unmerged plan blocks are executable. With the feedback at this phase, students can ensure that they are on the right track towards a correct solution. This positive feedback encourages them to persist to the next
phase of the process.

1. Apply plan blocks from library (or build new plan blocks)

Plan blocks can be mapped according to a plan network diagram by dragging-and-dropping them from our plan library developed in BYOB. However, dragging the needed plan block and dropping it to the right place between other dropped plan blocks is critical. After goal decomposition into undivided sub-goals, the relevant plans should be found in the plan library. Otherwise, a new plan block can be built into the library. Regarding the order of dropped plan blocks, they are arranged in a suitable sequence one after another according to a “left to right” heuristic in the diagram. Specifically, plan blocks are put in order by following the direction of dataflow in the diagram. For example, a plan that receives dataflow from another plan is placed after the plan that sends dataflow. If there are two branches of plan blocks that send dataflows to the same destination plan block respectively, these two branches can be put in any order. On the other hand, if there are two branches of plans that receive dataflow from the same plan block respectively, these two branches can be put in any order, but after the plan block that sends the dataflow.

For Example 1, according to the plan network diagram in Figure 6-5, the three identified plan blocks Input Plan, Sum Plan, and Output Plan can be directly applied by dragging-and-dropping from our plan library developed in BYOB. Following the direction of dataflow in the diagram, three plan blocks are also arranged in a suitable sequence according to a “left to right” heuristic in the diagram to ensure that a plan occurs after the plans that provide it with data (see Figure 6-8).

![Figure 6-8 Implementation of plan network diagram by using plan blocks for Example 1](image)

For Example 2, from the plan network diagram in Figure 6-6, only four identified plan blocks (i.e. all except Dividing Plan) can be used from our plan library developed in BYOB. If a plan
block cannot be found in the provided plan library, students have to build a new plan block (see Chapter 5). For this example, students need to build a new Dividing Plan block by referring to the provided Multiplying Plan block which has a similar pattern of two in ports to receive two items.

After building a new plan block into the plan library, students can drag-and-drop these five identified plan blocks and put them sequentially based on the diagram. According to the direction of dataflow in the diagram that is from left to right, the plan blocks are put into a sequence. For example, the left most Input Plan is the first one mapped from the diagram into an Input Plan block on the top and the right most Output Plan is the last one mapped into an Output Plan block on the bottom of the program. For the parallel plans, there is no constraint on their relative order. For example, because Sum Plan and Count Plan are in parallel in Figure 6-6, they can be put in either order (Sum Plan first or Count Plan first) after Input Plan and before Dividing Plan (see Figure 6-9).

Figure 6-9 Implementation of plan network diagram by using plan blocks for Example 2

Example 3 is complicated by the use of two input and two output plan blocks. According to the plan network diagram in Figure 6-7, seven identified plan blocks can be directly dragged and dropped from the provided plan library. They are put sequentially based on the plan network diagram (see Figure 6-10). Since two Input Plans are used in the same program, the out port of the second Input Plan must be distinct from that of the first Input Plan in order to distinguish the dataflow from different plans. By default, the out port of the first Input Plan is named “value1:out”. Therefore, the out port of the second Input Plan can be renamed to
“value2:out”. Similarly, the in port of the second Output Plan can be renamed as “output2:in” to distinguish it from the default name of the in port in the first Output Plan (“output1:in”).

Again, similarly to the previous example, for the parallel plans there is no constraint on the order between them. For example, Figure 6-10 shows two parallel Input Plans for entering data of pay rates and working hours. Because these plans are in parallel, they can be put in any order before the Multiplying Plan. Also, both Sum Plan and Count Plan are in parallel in Figure 6-10 and therefore, the corresponding plan blocks can be put in either order. Similarly, when two Output Plans are in parallel, their plan blocks can also be put in either order.

2. Link the plan blocks using plan ports for the implementation of dataflow

In order to be executable, plan blocks need to be linked using plan ports so that a dataflow can be sent from the out port of one plan block to the in port of another plan block. According to the plan network diagram, the two parameters of each “Link [current plan out-port] to [next plan in-port]” block are filled with port names from the two plan blocks to be linked. The left parameter of the “Link” block is filled with the out port of a plan block and the right parameter of the “Link” block is filled with the in port of the next linked plan block. Moreover, there is no order requirement to the “Link” blocks. That is to say, the link blocks can be put in any order.
For Example 1, according to Figure 6-5, the first arrow from out port “Input:out” to in port “Sum:in” links Input Plan to Sum Plan. Accordingly, based on the plan blocks in Figure 6-8, the first “Link” block is filled by the default out port name “values1:out” from the Input Plan block and the default in port name “Sum:in” from the Sum Plan block (see Figure 6-11). These port names can be typed in or filled by copying-and-pasting from the port of the plan block. A similar way can be then applied to implement the second data-flow arrow from out port “Sum:out” to in port “Output:in” for the linkage from Sum Plan to Output Plan in Figure 6-11. The second “Link” block is filled by the default out port name “Sum:out” from the Sum Plan block and the default in port name “Output1:in” from the Output Plan block based on Figure 6-8. Both “Link” blocks of the data-flow framework are declared between blocks “Begin Links” and “End Links”. Finally, the combination of plan linkages and plan blocks is shown in Figure 6-12.

![Diagram](image)

Figure 6-11 Linkage of plan blocks with port names for Example 1

![Diagram](image)

Figure 6-12 The program by using plan blocks for Example 1

Example 2 (see Figure 6-13) shows how “fork” shape dataflow (i.e. a dataflow from one out port moves to many in ports) are handled: the first two “Link” blocks represent that the two
dataflow in a fork shape arrow carry the same data from the out port of Input Plan to in ports of both Sum Plan and Count Plan simultaneously. Figure 6-14 shows the encoding of Example 3.

![Diagram showing dataflow and plan blocks]

**Figure 6-13 Using scaffolding to link plan blocks for Example 2**

### 3. Test

The plan network diagram can be mapped into BYOB using plan linkages and plan blocks. That is to say that the implementation for each of the above three examples is executable and testable in our data-flow framework. Testing provides feedback to confirm whether the design by using plan blocks is accurate. For example, the testing results are filled into column Phase 3 in Table 6-6. Otherwise, it is necessary to check whether these plan blocks, their order, and their linkages are matching to the original plan network diagram. It is also important to check that the port names in “Link” blocks are consistent with the port names in plan blocks, e.g. port name “values1:out” in “Link” block must be the same as that in the Input Plan block.

When using parallel dataflow in a plan diagram, it is important to check that all the port names linked by a “fork” shape data-flow arrow match those in the relevant Link blocks. For example, in Example 2, it is essential to ensure that the port “values1:out” is linked to both “sum:in” and “count:in” ports by using two “Link” blocks (i.e. Link [values1:out] to [sum:in], and Link [values1:out] to [count:in]).
Finally, when using a plan block more than once in a program, students need to check if the port name of the same plan block has been changed in the second copy. For example, in Example 3, students need to check if the port name of the second Input Plan block is renamed to “values2:out” as well as second copy of Output Plan block to “output2.in”.

![Diagram of plan blocks](image)

**Figure 6-14 Using scaffolding to link plan blocks for Example 3**

### 6.2.4 Phase 4. Expanding Plan Blocks and Changing Parameters

In order to merge these plan blocks, the details of each plan block must be firstly expanded to replace its original plan block. In other words, expanding plan blocks means replacing each plan block with the defined details within it. In other words, each procedure call (plan block) is replaced by its body with appropriate substitution of parameters. The expanded details are also executable due to the “Link” blocks. The expansion process consists of the following steps:

1. **Replace plan blocks by plan code details found inside each plan block**

   Each plan block can be expanded by editing the plan block and duplicating the blocks inside a “shell” (or interface) of the plan block. After this step, all the plan blocks are replaced by the details that composed these plan blocks (see changes from top to left bottom in Figure 6-15).
2. Replace every parameter of each plan by its argument value in the plan block

Recall Chapter 5, every plan block is built up by at least one of the data-flow blocks (“GET DATA”, “SEND DATA”, and “NO MORE DATA”). Each data-flow block contains one local parameter, either “in-port” or “out-port”, receiving or sending data from or to its host plan block. All the plan blocks (through their interface “shell”) are linked by Link blocks using port names. In other words, the local parameter is associated with Link blocks through the “shell” of plan block. However, after removing the “shell” of each plan block, these local parameters lose their connection to the Link blocks. Therefore, they must be replaced by the real port names in order to correspond to those in the Link blocks. Moreover, after expanding, local parameter “sentinel” from Input Plan must be changed to its argument value (-1) according to the original Input Plan block. Therefore, in Example 1, the details of replacing local parameters after expanding are as follows (see Figure 6-15):

- For the details duplicated from “Input Plan”, replace all instances of a variable with the values given, i.e. replace three “sentinel” by value -1; and replace one “out-port” by value of text “values1:out” (through typing in or copying-and-pasting from the Input Plan block).
- For the details duplicated from “Sum Plan”, also replace all instances of a variable with the values given, i.e. replace two “in-port” by text “sum:in” (through typing in or copying-and-pasting from the in port (left) of “Sum Plan” plan block); and replace one “out-port” by text “sum:out”;
- For the details duplicated from “Output Plan”, once again, replace all instances of a variable with the values given, i.e. replace two “in-port” by text “Output1:in”.

For Examples 2 and 3, the process of expanding the details of plan blocks and replacing variable are similar.

3. If the same plan is used twice, the same variables from both the same plans are renamed differently, such as [name1] and [name2]

When a program uses the same plan block more than once, the variable that receives or sends data in the plan details must be renamed in different copies of the plan details. It is similar to the renaming of port name, but it applies to variables that are not parameters. Example 3 is the only case for renaming variables in these three examples. Because two Input Plan blocks are used in Figure 6-10, both of them include the same variable Number that must be differentiated. Therefore, the variable from the first Input Plan is renamed as Number1 and the
variable from the second Input Plan is renamed as to Number2. Similarly, because both Output Plan blocks being used include the same variable Result, the variable from the first Output Plan is renamed to Result1 and the variable from the second Output Plan is renamed to Result2. Figure 6-16 shows the process of expanding plan blocks, replacing parameters and renaming variables for Example 3.

4. Test
The results of expanded plan details from three examples are also executable and testable. Therefore, the table of test schedule is used to record the intermediate level results. For example, Table 6-6 is used again to fill the testing results in the column of Phase 4 to ensure the results from current phase are consistent with those from previous phases. The feedback results confirm that there are no errors from the process of expanding and replacing in this phase.
Figure 6-15 Example 1 of expanding plan blocks and replacing original local parameters
Figure 6-16 Expanding plan blocks and replacing original local parameters for Example 3
6.2.5 Phase 5. Merging Plan Code Details

The merging of expanded plan details aims to combine the contents from different plans into one program in which only a single datum is sent and received between plans so that dataflow (and the associated use of buffers) can be eliminated. In other words, traditional variables, rather than buffers, are used to communicate data between plans. Since our plan blocks are all built up by following the same pattern (iterating, reading items from their input port, and dealing with items one at a time), they can be merged by following three steps.

1. Collect all the initialisation block

   The first step of merging plan details is to collect all the blocks that initialise variables by setting and inputting initial values, and put them immediately after the “End Links” block. These initialisation blocks can be identified as being those that are not inside a loop. For Example 1, the initialising blocks are a setting value block (“set [Sum] to 0”) and two blocks for inputting (“ask” and “set [Number] to answer”) outside the loop (not the two blocks for inputting inside the loop). For both Example 2 and Example 3, the setting blocks are similar to Example 1 such as “set Sum to 0” and “set Count to 0”. Specifically, for Example 3, the inputting blocks are four blocks outside two input loops (“ask” and “set [Number1] to answer” as well as (“ask” and “set [Number2] to answer”).

2. Combine loops which share the same dataflow

   When the first loop is used to generate the dataflow and the other loops receive this dataflow, the bodies of the other loops can be moved within the first loop. Specifically, consider the case where output port port_A is linked to input port port_B, and we have the following two loops:

   Repeat until <condition>
   <part 1 of the first loop body>
   SEND DATA (value, port_A)
   <part 2 of the first loop body>
   End repeat

   Repeat until NO MORE DATA(port_B)?
   Set Var to GET DATA (port_B)
   <the second loop body>
   End repeat

   The first loop sends a sequence of values to port_A. Because port_A is linked to port_B, the

25 The SEND DATA blocks are an exception to this principle: they are outside a loop, but are not initialisation, and so are not moved.
second loop will receive these data values from \textit{port}\_\textit{B}. In other words, both loops will execute the same number of times because each data sent in the first loop results in a corresponding single execution of the second loop. Therefore, the second loop can be eliminated by moving its body inside the first loop, where the data is sent out from the loop body of the first plan and received by the loop body from the second plan:

\begin{verbatim}
Repeat until <condition>
  <part1 of the first loop body>
  SEND DATA (value, port\_A)
  Set Var to GET DATA (port\_B)
  <the second loop body>
  <part2 of the first loop body>
End repeat
\end{verbatim}

The processing of dataflow between unmerged loop bodies is analogous to batch processing. On the other hand, after merging, each datum sent to \textit{port}\_\textit{A} from the first loop body is \textit{immediately} received by the second loop body. The processing of dataflow between merged loop bodies is therefore analogous to real-time processing.

We have discussed that in the case of multiple loops that receive the same dataflow from the first loop (i.e. a “fork” shape dataflow), the bodies of the other loops can be moved within the first loop. Similarly, in the case there are many loops sending and receiving dataflow in sequence. That is to say, the first loop is used to generate the dataflow and the second loop receives this dataflow, and then the second loop also generates another dataflow and the third loop receives this new dataflow, etc. Therefore, the above merging is repeated until there is one loop. In other words, initially, the first loop and the second loop are merged within the first loop condition. Then the resulting merged loop is merged with the third loop still within the first loop condition, and so on. Finally, in the case that multiple loops send dataflow respectively to the same loop, we assume that the loop receives dataflow equally from each loop. Therefore, we can also merge these loops together within one of the sending loop. All the loop bodies that send dataflow can be placed in any order, but before loop body that receives dataflow.

\begin{quote}
For \textit{Example 1}, since the loop of the Input Plan generates a dataflow to be shared by the loop of \textit{Sum Plan} (see the left side of Figure 6-17 for the unmerged plan details), the two loops are merged under the loop control from the first loop, “\textit{Repeat Until Number = -1}” (see the middle of Figure 6-17 for the merged version). The body of the loop of the Sum Plan (see the middle of
Figure 6-17 is put after “SEND DATA (Number) [values1:out]”, to directly receive the dataflow, and before the rest of blocks (inputting next value of Number) inside the first loop of Input Plan. The blocks outside of each loop body such as “SEND DATA (Sum) [sum:out]” are still kept outside of the merged loop.

For Example 2, when the same dataflow is shared by the loops from different plans, these loops can be merged together by the loop control of the data-flow sender. The merged loop bodies are arranged in order from sending to getting the dataflow. Since both Sum and Count Plans receive the same dataflow from the out port of the Input Plan, there is no constraint on their order. Both loop bodies from Sum and Count Plans can be merged together under the loop of Input Plan. They are put after sending dataflow in the Input Plan loop. The programs before and after merging of three plan details are shown on both sides in Figure 6-18.
Figure 6-18 Merging plan details of Example 2
For Example 3, when loop bodies from different plans are sending or receiving the same amount of data through dataflow, they can be merged together under the first loop control. Similarly to the previous example, both Sum Plan and Count Plan loop bodies receive the same amount of data of wages from the Multiplying Plan loop body. These three loop bodies can be then merged together under the loop control of data-flow sender, Multiplying Plan loop.

Before merging, the program requires to input all rates and then all hours (e.g. 15, 20, 30, -1, as well as 20, 40, 20, -1). Even though both Input Plans may receive different number of data by mistake (e.g. 15, 20, 30, 10, 10, -1, as well as 20, 40, 20, -1), the Multiply Plan still generates a dataflow (e.g. 300, 800, 600) based on the common number of both dataflow (e.g. 3, the minimum number). This is because the data-flow control blocks (NO MORE DATA [multiply:in.item1] and NO MORE DATA [multiply:in.item2]) stops extra unpaired data from being processed as part of the new dataflow (i.e. only three rates data will be got from one in port because three hours data can be got from the another in port). Moreover, the Multiplying Plan loop body receives equal valid amount of data from both two Input Plan loop bodies (hours and rates). These three loop bodies can be then merged under the loop control from one of the Input Plans (e.g. the first Input Plan) because both Input Plans send equal amount of data.

However, after merging, both Input Plans are interleaved. That is to say the merged program interleaves the requests of rates and hours (e.g. 15, 20, 20, 40, 30, 20, -1, -1). Because of having a common valid amount of dataflow (e.g. three data items for rates and three for hours), the merged program only checks for the sentinel (-1) for one of the input streams. Therefore, five loop bodies can be merged together under the loop control of the Input Plan loop body (see Figure 6-19).
Figure 6-19 Merging plan details of Example 3
In summary, the loop bodies of two Input Plans send the same valid amount of data (hours and rates) to the loop body of the Multiplying Plan, and the loop body of the Multiplying Plan also sends exactly the same amount result of wages to both Sum Plan and Count Plan loop bodies. Therefore, these five loop bodies can be merged and controlled by the condition from the loop body of the first Input Plan. The loop bodies from five merging plan blocks are put based on their original order before the merging, specifically following the order from sending to receiving dataflow. The initial input blocks of the second Input Plan are regarded as set initial value to the variable and put it at the beginning of the program. The second input block from the second Input Plan is also merged after the first input block from the first Input Plan. These input blocks are put at the end of the merged body.

3. Remove the loop control if it is driven by a dataflow including only a single value

This is a special case of the previous step. If a plan only sends one datum to the next plan without a loop control, the loop of the next plan can be removed when the ports of two plans are linked. In other words, for two linked plans, there is no need to have a loop control for the second plan if it is driven by only one datum. For example:

```plaintext
SEND DATA (value, port_A)
Repeat until NO MORE DATA(port_B)?
    Set Var to GET DATA (port_B)
    <the second loop body>
End repeat
```

Since the input dataflow from `port_B` only has single value because there is only one value sent to the linked port `port_A`, the loop can be removed as follows:

```plaintext
SEND DATA (value, port_A)
Set Var to GET DATA (port_B)
<the second loop body>
```

For Example 1, since from the Sum Plan (“SEND DATA (Sum) [sum:out]”), only one datum is sent to the Output Plan, the loop control “repeat until (NO MORE DATA? [output1:in])” of the Output Plan can be removed (see Figure 6-20).
Figure 6-20 Removing loop control from Output Plan for Example 1

For Example 2 in Figure 6-18, after merging the details of Input Plan, Sum Plan and Count Plan, the “SEND DATA (Sum) [sum:out]” block sends only a singular datum to its out port, which is linked to one of the in ports of Dividing Plan (dividend). Meanwhile, the “SEND DATA (Count) [count:out]” block also sends a singular datum to its out port, which is linked to another in port of the Dividing Plan (divisor). In other words, the Dividing Plan only deals with a single value of dataflow on both of its in ports. Therefore, the loop control for Dividing Plan can be removed. Consequently, the Dividing Plan only sends a singular datum to its out port by “SEND DATA (Result) [dividing:out]” and therefore, the loop control from Output Plan can also be removed.

For Example 3 in Figure 6-19, after merging five plan blocks, the “SEND DATA (Sum) [sum:out]” block sends only a singular datum to its out port, which is linked to the in port of the first Output Plan. Meanwhile, the “SEND DATA (Count) [count:out]” block also sends a singular datum to its out port, which is linked to the in port of the second Output Plan. In other words, the two Output Plan blocks only deal with a single value. Therefore, the loop controls in both Output Plans can be removed.
The merged program is also executable, which is used to provide feedback from this phase of the process. The test schedule table is updated with the testing results in the column of Phase 5 in Table 6-6. The feedback encourages students to complete the process.

6.2.6 Phase 6. Simplifying the Merged Details

We can also see, after merging, that a single datum is sent and received between plans without considering the test whether or not there is any more data in a port. In other words, a sequence of data between plans has been replaced by a single value. If the variables sending and receiving values through linked plan ports and these variables can be communicated directly by the using same variable name, then the plan ports can be removed. That is to say, a single datum can be sent and directly received between plans by traditional variables. Therefore, the data-flow framework with buffers which were associated with the dataflow becomes redundant and can be removed.

1. If two variables share a dataflow to or from the same port, rename the second variable to the first one

Before removing the data-flow framework, we need to combine variables that deal with the same data but have different variable names. When a variable has its value sent to an output port, and subsequently another variable receives the same value from a linked input port, the second variable should be consistently renamed to match the first one. When we have code of the form:

```
LINK p1 p2
...
SEND DATA (v1, p1)
v2 := GET DATA (p2)
<code referring to v2>
```

Then the variable v2 receives its value (via the SEND and GET) from v1, and can be renamed to v1:

```
LINK p1 p2
...
SEND DATA (v1, p1)
v1 := GET DATA (p2)
<code referring to v1>
```

For Example 1, because of blocks “SEND DATA (Sum, [sum:out])” (see box 1 in Figure 6-21) and “set (Result) GET DATA [output1:in]” (see box 2 in Figure 6-21), where both ports [sum:out] and [output1:in] (see box 3 in Figure 6-21) are linked, the variable Result will have the same
value as variable Sum. Therefore, replace the variable Result with Sum (see the bottom part of right side in Figure 6-21). The same process is applied to Example 2 (see Figure 6-22) and to Example 3 (see Figure 6-23).

Figure 6-21 Example of renaming linked variable for Example 1
Figure 6-22 Example of renaming linked variable for Example 2
Figure 6.23 Example of renaming linked variable for Example 3

2. Remove the data-flow framework

As per previous discussion, merging expanded plan details combines the details from different plans into one program in which data can be sent and received freely without considering the test whether or not there is any more data in a port. Furthermore, the initial renaming of linked variables makes every single datum pass only through variables rather than through both variables and ports. In other words, the traditional variables and control flow are gradually replacing the plan ports and dataflow during the process. Our data-flow framework with buffers which was associated with the dataflow can now be removed, since it is no longer used in the merged program. Specifically, each SEND-GET pair is of the form (SEND DATA...
(VARIABLE) [OUT-PORT], set (variable) to (GET DATA [IN-PORT])) which, since the ports are linked, has no effect on the value of the variable, and so can be deleted.

We can now simply remove all the blocks of the data-flow framework to achieve the final program in BYOB. The data-flow framework is created in grey colour, which makes it easy to see which blocks should be removed. The three final programs can be seen in Figure 6-24, 6-25, and 6-26.

Figure 6-24 The final simplified plan blocks of Example 1
Figure 6-25 The final simplified plan blocks of Example 2
3. Test

The testing of final program ensures the completion of the programming process. The final program is obviously executable and testable. The test schedule table is updated with the testing results in the column of Phase 6 in Table 6-6. The feedback supports the process of simplifying in this final phase. It also confirms the entire process.
6.3 Summary

A detailed step-by-step programming process for novices to use goals and plans in programming has potential in educational practice. We decided not only to introduce goals and plans to novices, but also to provide them with strategies and some form of guidance to combine these plans into a valid program.

We developed a detailed process that uses a “test-early” approach and supports the use of goals, plans and a visual programming language. The programming process includes six phases: (1) devising test cases; (2) analysing goals and plans; (3) encoding the plan network using plan blocks; (4) expanding plan blocks; (5) merging plan blocks; and 6) simplifying the results. The intermediate level products in each phase of the process except the first phase are executable and testable, which provides immediate feedback to novices and engages them in the process.

In particular, the first phase of predicting expected results from test data in different test cases provides novices opportunities to simulate the programming process manually. By the mental prediction and manual process with test cases, novices are confident to proceed through the process.

Through the second phase of analysing, all the goals and the order to achieve these goals (i.e. dataflow) for the solution are identified and represented in a goal diagram. According to the goal diagram, a plan network diagram can be directly mapped by using the symbol of plans. Furthermore, plan ports are identified on each plan in order to link plans by the dataflow. Moreover, a desk-check is applied to find out the dataflow through every port of each plan. If the dataflow on the in port of output plans are the same as the predicted outputs, this phase has been completed and continue to next phase.

In the third phase, the plan network is encoded by using plan blocks implemented in BYOB. The data-flow framework is used to link these plan blocks by plan ports. Through the linkage, the dataflows are transferred from one plan block to another. Therefore, the linked plan blocks are executable and testable. Novices can receive feedback at the plan level rather than wait till the final program code.
The fourth phase of expanding is the transition from the plan level to the programming details. The details of each plan block can be directly copied from the definition of plan details. However, the local variables (i.e. parameters) extracted from the plan blocks must be replaced by the relevant plan ports (i.e. arguments). Once again, these linked plan details are still executable and testable.

The fifth phase of merging is to combine the details from different plans into one program so that only one datum is transferred between plans. In other world, control flow and variables are used to transfer data between plans. This phase is the transition from data-flow paradigm to control flow paradigm. Again, the merged program is executable and testable.

Finally, the last phase of simplifying is to remove the blocks for the function of dataflows. Through the above merging, singular datum can be represented by variables and manipulated by control flow. Renaming of the port linked variables makes the datum independent from plan ports. The blocks for the function of dataflows become redundant. Therefore, the data-flow framework for both plan block linkage and dataflow can be removed. The final program contains only conventional variables and control flow.
Chapter 7 Teaching Method

In Chapter 6, we introduced our programming process which began with program design using a visual notation for goals and plans in the data-flow paradigm, and then shifted to implementation from plans by merging them, resulting in program code in a visual programming language — BYOB. We now turn to discussion of our teaching method, focusing on how we can support novices to learn the programming process of using goals and plans in the visual programming environment.

We observe that programming is a mental activity carried out by humans, and we therefore need to consider human cognitive factors. Therefore, we propose that our teaching method is developed by taking advantage of cognitive load theory (CLT) in general, and more specifically, considering the theory applied to the design of instructional material for teaching programming.

In Section 7.1, we start by reviewing CLT. We then proceed to explore the implications of CLT for teaching practice in general in Section 7.2. In Section 7.3, we focus on what we can learn from existing studies regarding instructional design for learning programming. We then set up our teaching strategies for improving cognitive load when using goals and plans in Section 7.4. Finally, in Section 7.5, we demonstrate how to apply our teaching strategies in our programming process using a visual programming environment (specifically, BYOB).

7.1 Cognitive Load Theory

Cognitive Load Theory (CLT) (Paas et al., 2003; Van Merriënboer, Kirschner, & Kester, 2003; Van Merriënboer & Sweller, 2005) was developed for instructional design in the 1980s based on the experimental observation that the human brain can only handle a limited amount of information at the same time. This amount was described as “seven plus or minus two” (or five to nine) items, chunks, or “elements” of information that can be held in a human’s working memory (Miller, 1956). More specifically, CLT is based on the hypothesis that the human cognitive architectural model consists of two different types of memory: working memory (or short-term memory) and long-term memory (Paas, Renkl, & Sweller, 2004; Van Merriënboer & Sweller, 2005) (see Figure 7-1). Working memory is the place where all the conscious cognitive processing occurs, such as thinking, reasoning and judging. However, it has very limited
capacity. Consequently, information can be easily lost from working memory due to overload. Conversely, long-term memory has practically unlimited space to keep large amounts of information, including diverse plans or schemas for solving various problems. Working memory and long-term memory are, respectively, analogous to the primary storage (RAM) and the secondary storage (hard disk) in a computer.

![Figure 7-1 Working memory and long-term memory](image)

Recently, CLT has been used to analyse cognitive load when learning programming (Mason & Cooper, 2012; Caspersen & Bennedsen, 2007; Mead et al. 2006). In this section, we firstly review general implications of CLT to learning, and then consider how it applies specifically to the learning of programming.

In instructional design (Sweller, 1994), the individual parts of material that must be learned are called elements. A schema is a cognitive construct for organising elements of information into a basic unit of knowledge. It is stored in long-term memory (see Figure 7-2). Schema construction is the process of organising this knowledge into a unit (i.e. a schema) and moving it from working memory to long-term memory (Sweller, Van Merriënboer, & Paas, 1998). In the other direction, schemas can also be retrieved from long-term memory to working memory (Paas et al., 2003). We use the terminology “schema retrieval” to refer to the retrieving of existing schemas from long-term memory to working memory.
Schema construction organises multiple elements as a single element and stores it in long-term memory, which, therefore, frees working memory capacity. Automation also applies to schema construction, allowing schemas to be processed unconsciously which results in a reduction of working memory load. Although automation of schema construction requires extensive practice, it can free working memory resources for other activities (Van Merriënboer & Sweller, 2005). Therefore, both schema construction and schema automation can reduce the load on working memory (Paas et al., 2004). In summary, schema construction is the process of schemas being formed by organising information and moved into long-term memory, i.e. construction is equivalent to organising plus storing.

Early CLT research (Paas et al., 2004) mainly concerned instructional design, with the aim of reducing unnecessary cognitive load on the working memory. Therefore, in CLT, cognitive load is connected with working memory and is measured in three categories: intrinsic cognitive load, extraneous cognitive load, and germane cognitive load, which we now proceed to define (Paas et al., 2003; Paas et al., 2004; van Merriënboer & Sweller, 2005) (see Figure 7-3).
Before further discussion of these three categories of cognitive load, we turn to analyse the notation of an element. Sweller (1994) defined that an element is any information that needs to be learned. If every element of a learning material can be understood individually, this material is called low-element interactivity material, for example, the words of a foreign language can be learned independently. However, if all the elements of the learning material cannot be understood until their interactions are processed in working memory simultaneously, this material is called high-element interactivity material. For example, when learning to speak in a foreign language, the elements include words (including different meanings and pronunciations), the grammar of sentences, and foreign cultures and habits. Subsequently, intrinsic cognitive load is the load from a learning material that must be processed simultaneously in the working memory such as words, grammar and cultures. A key issue that affects intrinsic cognitive load is the coupling (“interactivity”) of the concepts to be learned. All these elements must be processed in the working memory at the same time, which affects the intrinsic cognitive load. Intrinsic cognitive load is about the nature of the material that is being learned and is mainly determined by its element interactivity.

Conversely, extraneous or ineffective cognitive load (van Merriënboer & Sweller, 2005; Paas et al., 2004; Paas, et al., 2003) is the load that is unnecessary because it does not contribute to schema construction or schema automation. It is artificial and can be caused by inappropriate teaching methods, which can be reduced by instructors (Sweller, 1994). “If element interactivity can be reduced without altering what is learned, the load is extraneous” (Sweller, 2010, p125). For example, if the goal of learning is the listening comprehension and speaking of a foreign language, then spelling individual words when learning pronunciation is extraneous cognitive load. However, if the goal of learning includes reading and writing in a foreign language, then remembering the spelling of individual words is essential and hence intrinsic cognitive load.

Finally, germane or effective cognitive load (van Merriënboer & Sweller, 2005; Paas et al., 2004; Paas et al., 2003) is the load that is directly relevant to the processes of schema construction and schema automation. “It refers to the working memory resources that the learner devotes to dealing with the intrinsic cognitive load associated with the information” (Sweller, 2010, p126). That is to say, germane cognitive load refers to the working memory resources that deal with the contribution of teaching instructions to the learning. For example,
the load of instructions on mental resource that is imposed by processing information towards constructing and automating schemas.

### 7.2 Cognitive Load Theory in General Teaching Practice

CLT (Paas et al., 2003; Van Merriënboer et al., 2003) posits that novices are likely to suffer a heavy cognitive load in complex learning and that reducing cognitive overload can help learning. For example, the lower the extraneous cognitive load, the more possibility there is of increasing germane cognitive load. Two approaches that have been proposed to reduce cognitive load in teaching practice are the use of worked examples to reduce extraneous cognitive load and learning in stages: progressing from subsets or simplified material to the full material.

The use of worked examples, one of the best known techniques, was advocated to reduce the extraneous cognitive load to a low level and to hence allow for an increase in germane cognitive load at the same time (Paas et al., 2003). It has been found that teaching with worked examples is highly efficient for schema construction (Sweller et al., 1998). Worked examples include a problem state (input or conditions), a goal state (output or questions), and the solution steps (the process or the relationship between input and output), which enable learners to develop a generalised solution, i.e. construct a schema, and reduce extraneous cognitive load.

Although early CLT regarded intrinsic cognitive load as a constant that could not be altered (Sweller, 1994), a review (van Merriënboer & Sweller, 2005) reported on the reduction of intrinsic cognitive load for novices by progressive methods such as sequential learning, simple-to-complex, and part-to-whole task ordering when dealing with complex materials. Simple-to-complex sequencing is progressing from simplified versions of the situation to more complex versions, e.g. when learning mathematical calculations, starting with adding two single digit items and then progressing to adding items with multiple digits. Part-to-whole sequencing is progressing from partial tasks to the whole task, e.g. analysing data from an experiment (a part task) as opposed to conceiving, designing, carrying out, and analysing the experiment (a whole task).
These methods artificially reduce the element interactivity by isolating the interactions among the elements, which is based on the assumption that it is unnecessary for learners to understand complex information at an early stage. The full complexity can then be presented later. For example, the movement of objects is initially described by the result of a simple mathematic calculation in primary school education. It is then represented by an algebra equation in high school education. Finally, it is illustrated by a complex calculus equation in university education.

However, Sweller et al. (1998) raised the concern that although heavy use of worked examples can provide stereotyped solution patterns, excessive use of worked examples could restrain creativity. Moreover, one study found that using worked examples is not enough to achieve effective learning (Renkl, 2002). The study showed that although following worked examples has significant advantages that contribute to reducing cognitive load, the benefit from the worked examples mainly depends on how well learners explain the solutions of the examples to themselves (called self-explanation). That is to say when students are given worked examples to study and are prompted to explain each step of the worked examples in their own words, they obtain higher learning gains than without such prompting. Further study (Van Merriënboer & Sweller, 2005) suggested that adding self-explanation to worked examples is associated with a higher germane cognitive load:

“Students who used predominantly cognitive and metacognitive elaboration strategies invested more mental effort than students who used a passive strategy and also did best on the subsequent tests, indicating that effective example elaboration is associated with a higher germane cognitive load. A common instructional problem is that many students use a passive elaboration strategy ... [Students] do not spontaneously provide fruitful self-explanations when they study worked examples” (van Merriënboer & Sweller, 2005, p163).

Therefore, learners’ self-explanation activities were advocated to be used together with worked examples. In order to support self-explanation, the provision of instructional explanations was also proposed as a supplement (Renkl, 2002).

The previous paragraph argued that worked examples do not always work, and that certain factors are correlated with success in using worked examples. Hence, the effectiveness of using worked examples was enhanced by encouraging self-explanation. However, not all
students are able to give adequate self-explanation of the worked examples because of their lack of domain knowledge, especially, both knowledge of principles about objects, events and purposes of procedures, and strategic knowledge of selecting steps in a procedure (Van Gog, Paas, & Van Merriënboer, 2008).

Worked examples show how to achieve a goal from certain conditions, but do not show the strategy of problem solving. Given the issues that can arise in using worked examples, Van Gog, Paas, and Van Merriënboer (2004) proposed that the effectiveness of worked examples could be improved by adding process-oriented information. They suggested that “students have to understand a procedure to be capable of solving problems from novel categories” and “They need to know the domain principles and know why the solution steps are taken and why they are performed in this particular order” (van Gog et al. 2004, p85). Furthermore, “Taking a process approach to the use of worked examples in instruction might have beneficial effects on understanding and far transfer performance” (van Gog et al. 2004, p86). Far transfer is the ability to apply knowledge to novel scenarios while near transfer is the ability to apply knowledge to a similar scenario (Gray et al., 2007). Therefore, they advocated adding process-oriented information to worked examples, e.g. a step-by-step hands-on lab example, for the purpose not only of showing the solution steps, but also for explaining the reasons behind these steps. Following the process, novices mimic experts’ problem-solving behaviour. Since the process-oriented information includes experts’ “why” (principles or rules) and “how” (processes) information for solving a problem, it is clearly relevant to novices’ performance and prompts their continuing commitment in their task. Van Gog et al. (2004, 2008) suggested that the use of process-oriented worked examples can foster novices’ learning by reducing extraneous cognitive load and increasing germane cognitive load.

When using a process, there is a concern about increasing extraneous cognitive load because novices have to focus some of their attention on the process (van Merriënboer & Sweller, 2005). They suggested that teachers should motivate students to dedicate sufficient cognitive effort to the process as well as to enhance elaboration and self-explanations in practice. A further study (van Gog et al., 2008) confirmed that using process-oriented worked examples can, initially, promote novice learning. However, the process can become redundant and then be removed gradually (i.e. be faded) after novices have learned the process and are able to self-explain the procedure. In other words, the process is provided at the beginning in order to support working on certain tasks. It is removed eventually in later tasks.
In summary, CLT is based on the assumption that there is a limited capacity in working memory for the conscious processing of information elements. Elements are organised as schemas: units of knowledge stored in long-term memory with unlimited capacity. The process of creating and storing a schema into long-term memory is called schema construction. However, the existing schemas can be retrieved from long-term memory to working memory ("schema retrieval") and applied to solve problems. Furthermore, schema automation is the process of making schema operations more automatic, which frees working memory resources for other activities. Organising elements in schemas (schema construction) and automation of the process (schema automation) are two critical mechanisms of CLT to reduce the cognitive load in working memory. The cognitive load in working memory is classified into three categories: intrinsic cognitive load, extraneous cognitive load, and germane cognitive load. Table 7-1 lists four of the significant approaches to instructional design for effective novice learning by reducing the cognitive load in working memory.

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Effect on Cognitive Load</th>
<th>Studies of CLT in Teaching</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Worked examples</td>
<td>Reduce extraneous cognitive load.</td>
<td>Sweller et al. (1998)</td>
</tr>
<tr>
<td>a) simple-to-complex,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b) part-to-whole</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Self-explanations</td>
<td>Increase germane cognitive load.</td>
<td>van Merriënboer and Sweller 2005</td>
</tr>
<tr>
<td>scaffold fading</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### 7.3 Cognitive Load Theory in Teaching Programming

In the previous section, we have discussed strategies and considerations based on CLT for the design of instructional material in general. We now explore various works over the years that have applied these ideas to improve teaching and learning of programming.

The study of programming skills is considered to be complex learning, where a number of information elements need to be processed simultaneously. Indeed, novices need not only to consider the syntax and semantics of a computer programming language, but also to think about the method of combining the statements of a computer language into a valid program. Both programming knowledge (the facts, such as knowledge of planning methods, knowledge
of the computer language, and knowledge of debugging tools) and programming strategies (the methods of dealing with the facts, such as strategies for problem solving, strategies for implementing, and strategies for testing) are essential to building up a framework for novice programming (Robins et al. 2003). The knowledge and its relevant strategies at each stage must be processed in working memory at the same time. The number of elements processed simultaneously in working memory depends on the complexity of element interactivity and expertise of the learner. Robins (2010) argued that every concept in programming closely depends on many others. He proposed a hypothesis of learning edge momentum (LEM), in which the successful learning of a new concept makes it easier for further study of its related concepts, whereas unsuccessful learning makes it harder. The LEM hypothesis effectively conjectures that learning programming has very high element interactivity. He modelled the LEM effect to explain the bimodal distribution of CS1 grades, which has both a high failure rate and a high rate of high grades.

Recent work revealed that cognitive overload is still a problem for novice programmers. Mason and Cooper (2012) applied CLT to investigate the mental effort (the resources that are spent on cognitive load and the focus of attention in order to complete a task) of poorly performing students (the bottom 10%) in order to find out the reason for their learning failure. They suggested that the failure of students at the low end of performance is due to cognitive overload. They proposed that computer programming has an intrinsically difficult nature due to the inherent complexities, interrelations, and subtleties of the task. They also proposed that the way information is presented to learners is extraneous to the conceptual understanding needed for learning. In other words, computer language constructs presented to students are extraneous to understanding the concepts of programming. Furthermore, they argued that when the physical location of instructional text and a text-referenced diagram are separated, visual search is required to mentally integrate the meaning between textual and graphical components. Due to the split-attention effect caused by the isolated treatment of programming concepts and computer languages syntax, extraneous cognitive load is needed for a search and mental integration.

In a survey, Mason and Cooper associated the mental effort of understanding a problem statement with intrinsic load; the mental effort of using the environment and syntax with extraneous load; and the mental efforts of reinforcing previous programming concepts or learning from the problem with germane load. The survey results showed that the poorly
performing students had made significantly higher mental effort (even at extreme levels) than the average students. Therefore, they argued that poorly performing students deployed many cognitive resources on intrinsic cognitive load such as learning concepts and understanding problem statements.

Moreover, they proposed that:

“The choice of computer programming language and/or environment may have a direct impact upon the ease of programming, learning and applying underlying programming concepts. The language itself is extraneous to understanding these core programming concepts” (Mason & Cooper, 2012, p188).

They argued that students spend large amounts of their cognitive resources on extraneous cognitive load such as learning the syntax of a computer language and coping with the programming environment. Therefore, with such a high level of these three cognitive loads, they concluded that these students will have inadequate cognitive resources for the germane cognitive load devoted to the processing of schema construction and automation.

Finally, they claimed that since poorly performing students have made extreme levels of mental effort, these students have insufficient cognitive resources for the processes of learning. They suggested considering CLT for the improvement of instructional design of programming courses by focusing on programming concepts rather than programming language syntax. They also suggested using both worked examples and “fading” worked examples when teaching programming.

Fading worked examples (Gray et al., 2007) are a sequence of partially worked examples with decreasing solution steps, which are used to reduce cognitive load by isolating effort within a few limited steps each time. Gray et al. developed a set of fading worked example (FWE) sequences which cover the programming process including design, implementation, and semantics (such as writing assertions, execution and verification). However, they developed the FWE sequences only in the dimension of programming language structure (in terms of selection, iteration, and subroutine calls) instead of considering on the aspect of programming plans, schema, or patterns. They argued that FWEs promote developing near-transfer skills and showed that less time and effort is needed compared with using worked examples without fading. They claimed that extensive practice with FWEs helps develop far transfer skills. However, their development of partially worked examples had not been evaluated.
On the other hand, Soloway (1986) advocated providing “canned” solutions in the form of programming language templates to achieve common goals. Consequently, schema construction\(^{26}\) has been considered relevant to teaching novice programming by using existing experts’ programming plans. Furthermore, Rist (1989) proposed creating simple procedural program plans, storing them as plan schemas, and retrieving them for problem solving in a process of top-down “stepwise refinement”. Soloway (1986) also proposed using combinations of plans to achieve goals. Moreover, Van Merriënboer and Paas (1990) claimed that “Automation leads to task-specific procedures that may directly control programming behaviour” (p286).

A schema in programming is a general programming plan. It can be created through a process of mindful abstraction from worked examples. This can be helpful if worked examples are available. For example, a count plan is abstracted from several programs with the function of summing and counting numbers. The automation of using schemas requires extensive practice on worked examples. Van Merriënboer and Paas (1990) proposed a teaching approach of using task-specific procedures to free up processing resources which include observing the running programs, studying programs, modifying existing programs, and generating new programs. They also emphasised that extensive practice using worked examples is beneficial to learning programming.

In the discussion of CLT in Section 7.2, it was suggested that self-explanation should be used with worked examples to enhance their effectiveness because it prompts the transfer from learning to schema construction. In the 1980s, Mayer (1981) discussed the process of meaningful learning of technical information in novice programming. He found that using a concrete model of a computer first had better results than having a model last (i.e. this is an instance of isolating and simplifying elements). Interestingly, he also found that the model elaboration experimental subject group, in which learners were encouraged to explain the information in their own words (i.e. use self-explanation), performed better than the control group on problems requiring creative transfer to a new situation. Therefore, he advocated providing a simple concrete computer model and emphasised encouraging learners to explain the information in their own words in order to reduce cognitive load.

\(^{26}\) Some research has used the term “schema acquisition” instead of “schema construction”. However, we could find no precise definitions of these terms and the difference between them (if any) is unclear.
Caspersen and Bennedsen (2007) applied CLT to the instructional design of a programming course for the purpose of minimising extraneous cognitive load and maximising germane cognitive load. Specifically, they proposed three principles for the use of worked examples: 1) highlight sub-goals by labelling or by visually separating steps, 2) include multiple examples, and 3) encourage self-explanation. Moreover, they proposed to apply a technique of cognitive apprenticeship.

“The theory of cognitive apprenticeship holds that masters of a skill often fail to take into account the implicit process involved in carrying out complex skills when they are teaching novices. To combat these tendencies, cognitive apprenticeship is designed, among other things, to bring these tacit processes into the open, where students can observe, enact, and practice them with help from the teacher” (Caspersen & Bennedsen, 2007, p 112).

They also considered the four aspects of traditional apprenticeships including modelling, scaffolding, fading, and coaching. Modelling refers using models provided by experts. Scaffolding is an approach that supports temporarily achieving a solution. Supporting material is used as a scaffold and can be eventually removed. Fading means that the master gradually transfers more and more responsibilities to the apprentice.

“In order to translate the model of traditional apprenticeship to cognitive apprenticeship, teachers need to: identify the processes of the task and make them visible to students; situate abstract tasks in authentic contexts, so that students understand the relevance of the work; and vary the diversity of situations and articulate the common aspects so that students can transfer what they learn” (Collins, Brown, & Holum, 1991, p8, quoted by Caspersen & Bennedsen, 2007).

Using a model-based approach, Caspersen and Bennedsen developed a programming process for teaching object-oriented programming. They proposed programming as a modelling process, comprising three sub-processes: 1) abstraction in a real (or imaginary) world (called the referent system) by identifying relevant concepts and phenomena, 2) abstraction in the programming physical model (called the model system) by representing concepts and phenomena, and 3) modelling from the referent system to the model system. They adopted the incremental approach for teaching novices beginning with simple tasks and then moving to progressively more complex tasks together with worked examples. They advocated the
principle of “consume before produce” for the purpose of teaching code writing. The incremental steps included: 1) using provided methods in O-O classes, 2) modifying the provided methods, 3) extending these methods, 4) creating new methods, 5) creating new classes, and 6) creating a new model. They also explicitly taught patterns to reinforce schema construction. The pattern-based instruction (Mulder, 2005) was based on the solutions to basic recurring algorithmic problems, which are called algorithmic patterns. In order to solve a problem, inferences are made from a familiar pattern to an unfamiliar situation, which is also called analogical reasoning.

Caspersen and Bennedsen integrated multiple theories and techniques such as CLT, cognitive apprenticeship, scaffolding, faded guidance, worked examples, an incremental approach, a model-based approach and a pattern-based approach into programming education. However, they did not conduct any formal evaluation of their instructional design. Although they attempted to use process-oriented worked examples, they did not provide an effective way to motivate and engage novices in the process, e.g. it lacked immediate feedback from each phase of the process to increase germane cognitive load for schema construction, and did not use a visual programming environment to reduce the intrinsic cognitive load of syntax.

More recently, Margulieux et al. (2012) also explored techniques to reduce cognitive load in learning programming. In order to decrease intrinsic cognitive load through reducing the amount of information needed for solving a problem, they proposed to isolate computational thinking from syntax by using a drag-and-drop programming language such as Android App Inventor or Scratch. They also proposed three techniques to reduce extraneous cognitive load including: 1) worked examples, 2) sub-goal labels, and 3) scaffolding. Their notion of sub-goal labelling was based on the work of Catrambone (1994) who proposed that “a sub-goal groups a set of steps under a meaningful task or purpose” (p606). He argued that “a label for a group of steps in examples helped participants form subgoals as assessed by measures such as problem-solving performance and talk aloud protocols” (Catrambone, 1998, p355). In order to improve students’ programming, Margulieux et al. (2012) backed up Catrambone’s ideas by using sub-goal labelled instructional material. They utilised text labels as a cue to combine several steps into a meaningful group, which represents a sub-goal. These labels can be seen as a concise reference to a long series of steps in the instructional material. An example of sub-goals used in instructional material is to “create components; set properties; handle events from My Blocks; ...” (p73). Each sub-goal is a summary of several steps. For example, under the
sub-goal “handle events from My Blocks”, there are two steps: “1. Click on ‘My Block’…”; and “2. Click on ‘clap’ and drag out…” (p73). “Subgoal labels group steps of a worked example into a meaningful unit and help students identify the structural information from incidental information” (p72). Hence, students needed to focus on fewer steps in terms of sub-goals rather than considering all the individual steps. This is similar to the process of schema construction by organising elements into schemas to free up working memory from extraneous cognitive load. Through identifying the structure of worked example by using sub-goal labels, students are also encouraged to make self-explanations of the worked examples. Furthermore, Margulieux et al. advocated using scaffolding as an intermediate step from worked example to solving problem as well as using worked examples to reduce extraneous load.

Margulieux et al. conducted two experiments to evaluate whether or not participants had attempted to satisfy all the necessary sub-goals and complete them correctly. In the first experiment, there were 40 participants divided into two groups of 20. The experimental group with sub-goal labels in their instructional material was able to identify sub-goals, complete correct sub-goals effectively, define variables, and retain learned material better than the control group. In the second experiment, there were only 12 participants, which was not enough for the statistical tests when they were divided into two groups. Thus, the analysis was based on the effect size measured in terms of the attempted and correct sub-goals within the two groups. An effect size similar to that formed between groups from the first experiment was obtained from the second experiment. Margulieux et al. argued that if there had been the same number of students in both experiments, they would have observed the same statistically significant difference in performance between groups because of the same effect size within the two groups. In the second experiment, they concluded that participants in the experimental group performed better than those in the control group in attempting and completing correct sub-goals, describing their strategies and goals, and in effectively using computer language statements or commands. The results suggested that using subgoal-labelled instructional materials helps novices to learn programming by reducing extraneous cognitive load. The performance from explicitly using labels for sub-goals in the instructional material indicates that it is a promising method.

Following the use of labelled sub-goals, Kim, Miller, and Gajos (2013) proposed a method of labelling sub-goals to support learning. They added short quizzes to video tutorials including
sub-goal labelling questions in order to encourage learners to explicitly summarise the details they have learned. In subsequent work, Kim, Nguyen, et al. (2014) proposed to improve the learning experience of watching a how-to video by adding step-by-step annotations. They believed that step-by-step instructions encourage learners to sequentially process and improve task performance. They developed an interactive video player to display step descriptions and intermediate results. They found that learners performed better when using the new player compared to a traditional video player in the domain of cooking, makeup, and Photoshop usage. Furthermore, Kim, Guo, et al. (2014) applied a keyword summary to support learners watching online educational videos. They proposed that interaction data has potential to help both teaching and learning. They added data-driven and visualisation features to functions such as navigation trace, transcript search, keyword summary, and summary of high learning activity. Specifically, the learners could add their personal bookmarks or labels for future reference. There were six participants who took the on-campus version watching video without using interaction data, while another six participants used the data-driven interaction techniques. Kim, Guo, et al. found no significant differences in task performance amongst these 12 participants. However, the participants believed that the interaction data such as bookmarks is useful support in completing their tasks. Although it was not proposed in a teaching programming context, we see the sub-goal labelling method as promising.

Finally, recall from Chapter 2 the concern by Guzdial (2009) that teaching novices programming by having them to program with minimal guidance was ineffective. In a subsequent debate (Guzdial & Robertson, 2010), Robertson proposed to separate guided instruction from the practice of programming. She argued that it was:

“important that students know how to search and discover information for themselves. They require skills in self-directed learning. In the context of programming, for example, we may wish them to know how to look up documentation. We would also generally expect them to be able to search for information sources in the first stage of carrying out a research project” (Guzdial & Robertson, 2010, p11).

Although Robertson advocated unguided learning in the initial information collection, she was in favour of using BlueJ with small examples to “wire in” (or combine) small segments of students’ own code in their further program development.
According to CLT, novices’ working memory is limited and is easily overloaded by information. In complex learning, e.g. programming, novices need scaffolding to support their learning, e.g. an explicit process to follow. Also, recall from Chapter 2 the concern by Gilmore (1990):

“It seems that possessing knowledge is not the only problem that novice programmers have. In a number of cases they show that they have the knowledge, but also that they do not know how to adequately use it” (Gilmore, 1990, p232).

Robertson’s (Guzdial & Robertson, 2010) approach might work in subjects that require a single command, e.g. searching for relevant information from the Internet or an online library catalogue. Although she proposed using BlueJ as a visual programming environment and segments of code that are similar to plans, she still did not indicate a detailed process of combining these segments of collected information into a program. As a consequence to this: without explicit programming process, students could end up with two groups: those who are successful and those who are unsuccessful as proposed by Caspersen and Kölling (2009) (see Chapter 6).

7.4 Teaching Method for Improving Cognitive Load When Using Goals and Plans

Before we describe our programming process for using goals and plans, we need to introduce our teaching method based on CLT and its implication in teaching programming. Through the study of CLT and its practices in programming, we conclude that working memory, which has only limited capacity, can be easy overloaded by high-element interactivity in complex learning such as computer programming. Therefore, providing a plan library can directly support novices to reduce the extraneous cognitive load of organising the elements for a plan and to increase germane cognitive load when using goals and plans in program development.

Furthermore, it is hard for novices to find their own way of programming without explicit step-by-step guidance. Therefore, support in the form of a detailed process is needed for the transition towards schema construction and automation, i.e. increasing their germane cognitive load. In other words, using a process provided by experts, novices can diminish their extraneous cognitive load by organising elements using schemas and also increasing their germane cognitive load.
Moreover, progressive methods have been shown to be an effective way to reduce element interactivity by isolating the interactions among the elements to several places within simple environments. In order to decrease the intrinsic cognitive load, we chose this method by using the visual programming language—BYOB, so that students can focus on the programming skills first before their programming knowledge by learning syntax in other computer languages.

We also introduced visual notations for goals and plans in a simple to complex order. Before using the visual notations, we provide the details of data processing in an example, and require students to complete a similar manual process exercise. We ask students to name each sub-process rather than provide these names based on the method of sub-goal labelling, as the students have not yet analysed the goal structure (e.g. name the processing details of adding 1 + 2 + 3 = 6 as “sum”). Using the scaffolding of naming each part of the manual process, the notation of goals and plans is applied in the analysis of the solution. After using the visual notations developed in Chapter 4, we provide a plan library in a dataflow framework developed in Chapter 5. Students can directly use the existing plans or schemas that experts have rather than consider how to build all the plans. With extensive practice with existing plans, students will become familiar with commonly used plans. The process developed in Chapter 6 allows novices to focus on how to combine these plans to solve a problem in a top-down manner using stepwise refinement. In other words, when we use our visual notation of goals and plans to design a program, we also need to provide detailed guidance for novices regarding how to progress from the design level to the code level and from the data-flow paradigm to the conventional control flow paradigm.

The support provided by teaching a detailed process is a scaffold that is based on a particular way that masters demonstrate the complex skills step-by-step when teaching apprentices. However, introducing a process for programming will increase extraneous cognitive load for novices to learn and to follow the process. Therefore, worked examples are needed together with the process in order to motivate novices for effective learning (schema construction and automation). Especially, with goals and plans at different levels and using different paradigms, we need to provide feedback from intermediate products in each phase of the process in order to enhance the engagement. This feedback provides a confirmation of success in organising elements into schemas in one phase and reducing relevant extraneous cognitive load.
We now summarise several approaches in teaching programming and their effects on reducing cognitive load. Before describing details of our teaching method, we outline our strategies which combine our development with current programming teaching practice based on CLT. Our strategy includes all of the features discussed previously (see Table 7-2). For now, we focus on the strategies themselves. The discussion of where they get implemented is in Section 7.5.

1. Our strategy is based on worked examples and fading worked examples in order to reduce novices’ extraneous cognitive load. Furthermore, we start introducing completed worked examples at both the design and code levels and then let students complete exercises with partially worked examples with embedded process information in the program code comments.

2. Our strategy is based on the progressive method to isolate the contents of solving problems in a simple-to-complex and part-to-whole manner in order to reduce novices’ intrinsic cognitive load. We introduce a simple worked example first followed by more complex examples. We apply part of the visual notation of the goal diagram and extend this to use the full of notation in the plan network diagram.

3. Our strategy promotes novices’ self-explanations of “what” to achieve (goals) and “how” to achieve them (plans and dataflows) at the beginning of the programming process, in order to increase novices’ germane cognitive load. Students summarise and label certain process steps with names when predicting output from testing data. We use a visual notation for goals and plans to support students in presenting solutions based on their understanding of goals and plans. We also use a desk-check table for them to evaluate their visual diagrams in order to prompt their self-explanations in different formats.

4. Our strategy is based on process-oriented worked examples in order to increase novices’ germane cognitive load. We not only provide detailed steps of the programming process from using goals and plans to the final program, but also make a set of general rules for merging those plans available for diverse situations. In general, the process is a scaffold for novices. After novices can efficiently apply strategies that experts have for the purpose of organising plans to solve problems, they do not need to keep the process at all. Moreover, this strategy focuses on using processes when teaching worked examples whereas the first strategy in the list is not about the processes of teaching, but about the contents of teaching.
5. Our strategy is based on the test-first and test-driven principles as commonly practiced in software development. Test-first means writing a test first, then running that test after the code is written, i.e. preparing a test before writing code. Our strategy aims to provide feedback from every phase of the process so that cognitive resources used by misconceptions can be freed in each phase and be available for germane cognitive load. In other words, it reduces unexpected cognitive load by avoiding unrelated extraneous cognitive load. Since the worked examples used in each phase of our programming process are executable, the feedback from intermediate levels indicates the success of learning in each phase, and the process can move to the next phase. Therefore, the feedback builds up novices’ confidence step by step and engages them to complete the process.

Table 7-2 Comparison of CLT approaches in teaching programming

<table>
<thead>
<tr>
<th>CLT Approaches</th>
<th>Changes of Cognitive Load</th>
<th>Approaches Applied in Teaching Programming</th>
<th>Our Teaching Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Worked examples and fading worked examples</td>
<td>Reduce extraneous cognitive load.</td>
<td>Gray et al. (2007)</td>
<td>Using executable examples for students to learn the process by reducing from providing full guidance to providing less guidance.</td>
</tr>
<tr>
<td>2. Progressive methods: 1) simple-to-complex, 2) part-to-whole</td>
<td>Reduce intrinsic cognitive load.</td>
<td>Caspersen and Bennedsen (2007)</td>
<td>Combining both methods. While the first one is for different ordering of worked examples, the second is for introducing notation from goals to plans, as well as from design level to code level.</td>
</tr>
<tr>
<td>4. Process-oriented worked examples and scaffold fading</td>
<td>Increase germane cognitive load.</td>
<td>van Merriënboer and Paas (1990), Caspersen and Bennedsen (2007)</td>
<td>Our teaching strategy includes information about both rules and steps. The programming process itself is a scaffold for novices to use only at the beginning.</td>
</tr>
<tr>
<td>5. Provide feedback from each phase of the process</td>
<td>Avoid wasted effort and allow students to focus on the current phase, which may reduce cognitive load.</td>
<td>Proposed in this thesis</td>
<td>The feedback from each phase provides confidence to engage novices to continue with the programming process rather than to be entangled with issues or errors due to work done in a previous phase.</td>
</tr>
</tbody>
</table>
7.5 Teaching with a Programming Process by Using Goals and Plans

Previously (Section 7.4), we have identified five CLT approaches in teaching programming, which are 1) worked examples and fading worked examples; 2) progressive methods: a. simple-to-complex, b. part-to-whole; 3) self-explanations and instructional explanations; 4) process-oriented worked examples and scaffold fading; and 5) provide feedback from each phase of the process. After introducing our programming process, we now briefly describe how we teach the process by using these five teaching approaches.

7.5.1 Starting from Worked Examples and Fading Worked Examples

We teach students using worked examples in every phase in the programming process rather than have one final worked example for each problem. In other words, we provide several intermediate level worked examples for students to learn the current phase and to advance to the next phase.

First of all, we provide an example of the first test case in a table and require students to complete the table for the second test case in order to support them to understand the manual process of the solution. We emphasise the critical role of the first phase of devising test cases in the process. The failure of the first step could mislead the student doing the rest of phases in the process. Therefore, we provide scaffolding to support students from the beginning of the first phase.

Students are firstly required to study the question in order to identify the output and input. Studying the question helps students to discover the requirements and the provided conditions, which provides the first step of guidance in the learning process towards the programming process. For Example 1 from Section 6.2.1, we required students to “talk aloud” by explicitly exploring the question in order to understand the requirement that output is the “total of the values” while the inputs are a sequence of integers. The integer “-1” is explicitly explained as a sentinel representing the end of the input, but is not included in the data to be processed into the sum. For Example 2, students are also required to understand the output is the “average” while the inputs are integers, excluding the last value of “-1”. For Example 3, two outputs are identified as “total wages” and “count of people” while two inputs are
recognised as “pay rates” and “working hours”.

Next, students use a provided test case as scaffolding to understand the processing from input to output. For Example 1, the column of Test Case (1) in Table 7-3 shows the process of computing the sum (1+ 2 + 3 = 6) when the input data is 1, 2, 3, and -1, as well as the output result is 6.

We argue that the understanding the process details is a critical step to start the analysis. We then require students to compute the results of two test cases in order to predict the output results for further phases in the process. After concrete calculation details, the goals can be identified for the analysis. Diagrams of goals and plans can also be built, including the dataflow between them.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1)* (For example: )</th>
<th>Test Case(2)*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data</td>
<td>Input</td>
<td></td>
</tr>
<tr>
<td>Predicted</td>
<td>Process</td>
<td>Process</td>
</tr>
<tr>
<td>Answers</td>
<td>1, 2, 3, -1</td>
<td>3, 4, 5, 6, -1</td>
</tr>
<tr>
<td></td>
<td>1 + 2 + 3 = 6</td>
<td>3 + 4 + 5 + 6 = 18</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>18</td>
</tr>
</tbody>
</table>

*All the italic and underlined numbers are provided to students. The rest of the numbers are the results that students were required to work out.

Furthermore, for each question, we provide students with a sequence worked examples that cover each phase in the programming process. We give worked examples in BYOB plan blocks and require students to expand the details from these plan blocks. Again, we offer worked examples of expanded plan details and require students to merge these details. Once again, we provide merged programs and also ask students to simplify them into a final program. Finally, we provide a final solution.

In these worked examples, we also embed process information to avoid students’ attention being split between workbook and program. For Example 1, Figure 7-4 illustrates how to implement a plan network (see the first part of Figure 7-4) as an executable program using plan blocks (see the second part of Figure 7-4) under the data-flow framework. The embedded process information provides step-by-step guidance for novices to map from the plan network to a plan block based program. Firstly, it provides information to connect the program to the diagram from the previous analysis (see the top comment in the second part of Figure 7-4). Secondly, it provides information on how to organise the plan blocks according to the diagram (see comment 1 in the second part of Figure 7-4). Thirdly, it provides detailed information on
how to link these plans blocks using plan ports in the diagram (see comments 2, 3, and 4 in the second part of Figure 7-4). Fourthly, it provides summarised information to support students in renewing the completed tasks (see comment “In Summary” in the second part of Figure 7-4). Finally, it reminds students to test and confirm the results from the current phase (see the last comment in the second part of Figure 7-4). Note, we use “values1:out” instead of “input:out” for the out port name of Input Plan as more than one Input Plan can be used in the same program. For example, the out port name for the second Input Plan can be named “values2:out”. Similarly, we use “output1:in” instead of “output:in” for the in port name of Output Plan as more than one Output Plan can be used in the same program.

Note, we use "values1:out" instead of "input:out" for the out port name of Input Plan as more than one Input Plan can be used in the same program. For example, the out port name for the second Input Plan can be named “values2:out”. Similarly, we use “output1:in” instead of “output:in” for the in port name of Output Plan as more than one Output Plan can be used in the same program.

Figure 7-4 Worked example with embedded process information for Example 1

In the following phases of the programming process, we provided students with a sequence of fading worked examples for each phase together with process information. That is to say, every worked example for the current phase starts from the solution of the worked example in the previous phase. After students successfully learn a worked example, the scaffolding of the current phase is faded and similar scaffolding for the worked example starts in the next phase.
In Figure 7-5, following the process information in program comments, students start from a partially worked example in Part 1 to complete the expanding of plan details in Part 2. Part 1 is the result of the previous phase (see Figure 7-5). Finally, the solution of Part 2 is also provided in another worked example to start for next phase.

Figure 7-5 Expanding from partially worked example for Example 1

For the merging phase, students start from Part 1 to merge the plan details in Part 2 in Figure 7-6. The solution of Part 2 is also provided separately in the next worked example. Both parts in Figure 7-6 show the status before and after merging. Using the expanded plan details, students focus on how to merge these details from different plans by following the provided process information (see Figure 7-6).

Again, in Figure 7-7 for the phase of simplifying, students start from Part 1 to simplifying the plan details in Part 2. In this worked example, students only focus on how to simplify the merged details by removing the dataflow scaffolding blocks (i.e. linking, testing, getting, and
sending dataflow blocks). In other words, they smoothly transfer the program from the dataflow paradigm to the control flow paradigm. Figure 7-7 illustrates the steps of renaming a variable first and then removing the data-flow blocks for the simplifying phase. The solution of Part 2 is also provided separately as the final program.

Figure 7-6 Merging from partially worked example for Example 1

Figure 7-7 Simplifying from partially worked example for Example 1

7.5.2 Using Progressive Methods

Recall that in CLT approaches, progressive methods can be used to reduce intrinsic cognitive
load. After novices master the simple format and part of information, the cognitive resources for this part will be freed. When moving to the complex format and the whole information, novices will only focus on the new information. In other words, they will use fewer cognitive resources for the new information than for the entire information. Progressive methods are not only used from step to step in each phase, but are also applied from phase to phase in the process. In the initial analysis of goals to achieve, the goal diagram only needs some of the symbols to identify goals and the dataflows between goals. On further consideration of implementation of the dataflows by linking though ports, the symbols of ports are then introduced in terms of single and multiple data plus the text giving unique port names and combining a plan name and the functions of in port and out port. In other words, the symbols are introduced progressively in the further step of drawing a plan network diagram rather than in the initial goal diagram.

All phases of the process are also introduced progressively from simple to complex. The process starts from the phase of predicting output by understanding the entire process, to analysis by visual notation and then the implementation by plan blocks. Students then work from several plan blocks to multiple details expanded from the plan blocks. Furthermore, they are required to follow several rules to merge these details from multiple plans into one program.

Examples are also introduced from simple to complex. A basic Input-Process-Output example is introduced first (see Example 1). Students start learning the programming process from three basic plans. They then begin to work on similar exercises, e.g. counting a sequence of numbers, or displaying maximum or minimum value of a sequence of numbers. Next, we introduce more complex processing such as displaying the sum of even numbers in a sequence (e.g. when entering the numbers 1, 2, 3, 4, 5, 6 and sentinel -1, display result $2 + 4 + 6 = 12$). Students are required to complete a similar exercise to display the sum of odd numbers in a sequence (e.g. when entering the numbers 1, 2, 3, 4, 5, 6 and sentinel -1, display result $1 + 3 + 5 = 9$). Finally, we introduce multiple inputs and outputs together with multiple processes. We start with an example of calculating wages ( = rates x hours), which needs two input plans to input rates and hours respectively. Students are then required to calculate Sales ( = price x amount) using two input plans for price and amount. And then we introduce multiple output plans (see Example 3). We also required students to modify previous example to display sum of wages, count of payment, and maximum wage.
7.5.3 Promoting Self-explanation

Recall that self-explanation is considered as an important approach to increase germane cognitive load. We emphasise using a manual process to help students to understand the purpose or goal of the program, and to learn the process of how to mentally achieve the goal. We require students to predict output from input data and also to label implicitly represented sub-goals in the question. After studying the worked example from Test Case (1), students are required to predict the answer for Test Case (2). Through this manual process, students become familiar with: 1) how many goals they had achieved, and 2) in what the orders these goals were achieved. We argue that the manual process not only assists students to understand and self-explain the processing details on the paper, but also represents students’ mental process on the paper. In this point of view, parts of students’ cognitive resources for the mental process can be released in their manual process by permanently recording their ideas from memory on the paper. The diagram also indicates why the previous phase needs many different processes and what relationships are about these processes. Therefore, the manual process promotes self-explanation and reduces novices’ cognitive load. For Example 1, in column Test Case (2) in Table 7-3, we provide testing data for input (e.g. 3, 4, 5, 6, and -1). Students are required to compute the sum of these numbers excluding the sentinel value (e.g. -1), and then to output the total value as the result (e.g. 18).

Moreover, the prediction of answers in the first phase encourages students to simulate the process in order to identify the goals to achieve. For more complex cases, students are required to name all the sub-goals by labelling, which summarises a group of steps in the process (see Tables 7-4 and 7-5 for Examples 2 and 3 in Section 6.2.1). We use sub-goals in a different sense to the sub-goal labelling researchers because for them a sub-goal relates to a part of the process whereas for us (following Soloway and his colleagues) a sub-goal is a part of the program and design. Now, the summary of the details (i.e. by labelling sub-goals) is an abstraction based on the student’s understanding of the process details. The summary represents chunks of information. Labelling sub-goals in the program design encourages students to understand the step-by-step details of a manual process and then to summarise the details. A set of summaries promotes students’ understanding of the order of the process steps at a higher abstract level. Through the abstraction, students can build up their mental model of understanding and then subsequently represent their mental model in a diagram.
Therefore, labelling the process details by summarising groups of details promotes students’ understanding and self-explanation to the program. It assists students to identify the goals and sub-goals as well as the relationships between these goals. With individual labelled sub-goals, it also supports the process of goal decomposition.

For Example 2, the column for Test Case (1) in Table 7-4 demonstrates three steps to process the input data. Students are firstly required to predict the answer of the average computation for Test Case (2) from the provided Testing Data in Table 7-4. They are then required to give a name for each sub-process in the sub row of the process section under column Naming the Details (e.g. sum). This table is scaffolding to promote goal decomposition by the method of sub-goal labelling (Kim et al., 2013). It assists students to identify goals to achieve and obtain testing results. The test case and results can be used as correct answers for the tests in the following phases.

**Table 7-4 Predict Answers of Average and Name Sub-processes (Sub-goal Labelling)**

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1) (For Example:)</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data (Input)</td>
<td>1, 2, 3, -1</td>
<td>2, 3, 7, 8, -1</td>
</tr>
<tr>
<td>Predicting Answers</td>
<td>Naming the Details</td>
<td>Computing Details</td>
</tr>
<tr>
<td>Process</td>
<td>e.g. sum</td>
<td>Computing Details</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1+2+3 = 6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1+1+1 = 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6/3 = 2</td>
</tr>
<tr>
<td>Output</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

We advocate that self-explanation can take different forms, not only in textual form, but also in graphical and in numerical forms. When predicting outputs in the first phase (in a textual form), labelling sub-goals promotes students’ self-explanation of what goals to achieve. By contrast, when making the diagrams of goals and plans in the second phase, students need to know what goals and plans are used; how they are related to each other; where they are located in the diagrams; when they are used; why they are presented in such a way; etc. We argue that the diagram of goals and plans promotes students’ self-explanation in a graphical form. A goal diagram represents students’ understanding of these questions. Specifically, after mapping the goal diagram into a plan network diagram, it represents that students worked out how to achieve these goals by using plans and linking these plans through ports. An adage says: “A picture is worth a thousand words.” We argue that the diagram represents the mental model of students’ understanding of the program. While novices work in the process of
developing their diagram, they are showing their progress in what they understand and how they interpret their understanding of the solution to the problem.

Furthermore, a desk-check table in the second phase is used to simulate dataflow in each port of the plans. After development of goal/plan diagrams, desk-checking is used to evaluate the diagram. It supports students’ mental processing of their plan network diagrams. It also promotes students’ self-explanation in a numerical format such as what data is in each port; how the computational formulas are applied to the data; where the results are received from and sent to, etc. Through the process of evaluation by testing cases, students re-access the diagram to enhance their understanding in different way by computing. It provides measurable feedback about the diagram to promote their self-explanation in order to further understand and improve the diagram. Similar to predicting results without the diagram in the first phase, the desk-check results through the diagram confirms the design in the second phase.

7.5.4 Using Process-oriented Worked Examples and Scaffold Fading

We have described in the above that we use worked examples in every phase in the process rather than only one final example. For each example, we have provided a sequence of worked examples phase-by-phase. Every worked example is connected from problem to solution for the current phase. We provide detailed steps and rules for how to complete the current phase and move into the next phase. Each worked example is not only passively showing the results of the current phase, but also actively indicating the steps of how to work out a solution by keeping the step-by-step process of current phase in the example as comments.

Scaffolding is used at both the programming code level and the programming strategy level. It eventually fades from both levels. At the programming code level, we firstly introduce the “Link” block in the data-flow framework to link plan blocks by dataflow so that plan blocks can be executable before merging. We then show the data-flow blocks of the data-flow framework, e.g. “set (variable) to GET DATA [port]” and “SEND DATA (variable) [port]”, to receive dataflow from a plan port to a variable and to send data from a variable to a plan port. Finally, after merging the plan details into one program, only a single datum in the dataflow is sent or received from a plan port. The single datum is also send and received from variables by control flow. Therefore, the data-flow framework is redundant and removed eventually.
At the programming strategy level, the fading of the data-flow framework in programming is visible in the final program while the fading of the process as scaffolding is invisible. We consider our process is scaffolding for students to learn programming. Students follow the process step-by-step to learn programming at the beginning. After they have done intensive practice, they find their way of programming with the explanation by themselves; they can then fade this programming process.

### 7.5.5 Providing Feedback in the Process

In order to provide early and frequent feedback (see Table 7-2, #5), we adopted the test-driven learning approach. After each phase except the first phase, students are required to check if the results from the current phase are the same as those from previous phase, i.e. whether they produce the expected answers for the test cases. For the second phase, the checking is done by a manual desk-check against the predicted results from the first phase. For the following four phases, all the checking is completed by comparing the test results from executable programs against those from the previous phase. Therefore, through feedback in each phase, students can find out whether the tasks they have done are correct, which provides confidence to engage novices to continue with the programming process rather than to be entangled with issues or errors due to work done in the current phase. We argued that the collection of feedback can avoid wasted effort and allow focus on the correct phase which may reduce cognitive load. Particularly, collecting feedback at the beginning can advise students by continuing the successful task or altering their solution after an unsuccessful attempt.

### 7.6 Summary and Discussion

A detailed step-by-step programming process for novices to use goals and plans in programming has potential in educational practice. In order to guide the teaching of our programming process, we reviewed the existing research in cognitive load theory (CLT) and its implications for teaching computer programming. We decided not only to introduce goals and plans to novices, but also to provide them with strategies and some form of guidance to combine these plans into a valid program.
According to CLT, a human’s working memory is limited. Through learning procedures, people can acquire schemas based on abstraction from concrete problems and solutions. Schema construction is the process of organising information into a unit (i.e. a schema) and storing it in long-term memory.

It is hard to learn and teach computer programming because of high-element interaction involved in the programming process which increases cognitive load on working memory. Therefore, we make diverse efforts to reduce intrinsic cognitive load and extraneous cognitive load, and to increase germane cognitive load for novices when deciding our teaching strategies, particularly using the process-oriented worked examples in a visual goal and plan environment.

The use of a plan library can directly support novices to reduce the extraneous cognitive load of organising the elements for a plan and to increase germane cognitive load when teaching goals and plans in programming. Furthermore, using a visual programming environment (such as BYOB), in the process of programming does not require students to remember syntax and to learn the use of a compiling tool, which reduces intrinsic cognitive load.

The use of a programming process reduces extraneous cognitive load and increases the germane cognitive load of having to decide "what do I do next?" which benefits effective learning (schema construction). Although teaching with worked examples is commonly used for decreasing extraneous cognitive load, it still creates two different groups of learners who are respectively successful and unsuccessful. Therefore, in the design phase, we use labelling of sub-goals after predicting outputs to promote self-explanations of “what” goals to achieve. We promote students’ self-explanation in different formats such as graphs and numbers. Using diagrams of goals and plans as well as desk checking, students can work out the answers of “how”, “where”, “when” and “why” questions in different formats, which reduces their extraneous cognitive load and also increases their germane cognitive load.

The feedback from intermediate products in each phase of our process motivates and engages novices in the process. We argued that the collection of feedback can avoid wasted effort and allows focus on the correct phase which may reduce cognitive load. Otherwise, issues or errors may continuously take cognitive resources during each phase in the process until the feedback at the end of program testing. We also argue that the separated phases reduce the intrinsic
cognitive load by isolating the entire process into different stages in terms of design, implementation, dataflow and control flow so that novices learn the entire process in several progressive steps from part to whole.

We propose that providing schemas or plans that experts use can reduce the novice’s burden on working memory because novices do not have to learn how to construct all the schemas or plans by organising diverse elements. Furthermore, providing support for using and building these schemas or plans in a step-by-step process can reduce the load in working memory. We foresee that after extensive practice of the process toward semi-automation, novices can gradually foster unconsciously using the existing schemas or plans, or creating new ones.

Although learning the process will increase extraneous cognitive load, it is proposed to be used by novices as scaffolding only at the beginning of learning. After novices can self-explain the worked examples in each phase of the process, the process will be faded. In order to support novices’ effective learning, our teaching strategies of using goals and plans was attempted to reduce cognitive load. We advocated using five techniques: 1) worked examples and fading worked examples, 2) progressive methods, 3) self-explanation by using sub-goal labelling, visual notation, and desk-checking, 4) process-oriented worked examples and scaffold fading, and 5) feedback from each phase of programming process for decreasing the extraneous and intrinsic cognitive load.

By applying these teaching strategies to support our teaching of the programming process introduced in Chapter 6, we also developed a sequence of fading worked examples (FWEs) for every phase of the process. We combined worked examples and FWEs in each phase by providing a completed worked example from the previous phase with process information and requiring students to learn this and complete the current phase. Therefore, our teaching strategies are used in teaching different phases of our programming process.

To summarise, we developed our teaching strategies of using goals and plans in programming according to the cognitive load theory (CLT) and its implications for teaching computer programming. Based on these strategies, we developed our programming process of using goals and plans in a visual programming language. We next conducted a field experiment to evaluate our approach.
Chapter 8 Evaluation

In previous chapters, we have introduced the development of our data-flow framework for using goals and plans in BYOB. We also introduced a programming process for teaching novice programming by using goals and plans with this framework. The purpose of this chapter is to evaluate whether the proposed visual goal-plan method has an effect on students’ learning.

We initially evaluated our approach in 2011 (Hu, Winikoff, & Cranefield, 2012, 2013). The results from our proposed method for teaching novice programming were significantly different to previous results from conventional methods taught by the author of this thesis. However, from a small data sample (eight students), our p-value (.031) is not much less than the threshold value of .05. Therefore, we continued evaluating our method in 2013. Based on our earlier experience, we adjusted the way of delivering our method, particularly in the design of instructional material based on CLT to avoid a split-attention effect, by combining instructional information into worked examples, as well as providing fading worked examples for exercises as described in Chapter 7. Now, we combine our experiences in both 2011 and 2013 to evaluate our approach. This chapter extends the 2011 evaluation by adding data from 2013.

8.1 Research Design

8.1.1 Methodology of Collecting Data

We aim to compare students’ examination results after teaching by using the proposed visual goal-plan method to those obtained after teaching by using a conventional method. For the evaluation, we choose to use a real teaching setting, rather than a short workshop. The author of this thesis taught novice programming at Tairawhiti Polytechnic (renamed as Eastern Institute of Technology Tairawhiti Campus since 2011), Gisborne, New Zealand, from 1997 to 2009. Students were heterogeneous, from different age groups, academic backgrounds and learning cultures. The introductory programming course is taught as a nine week module PP490: Programming Principles for the Diploma of Information Technology, which is a first year course for the Bachelor of Computing Systems. They were taught interactively in a small class (between 8 to 16 students each year) with a mixture of theory and practice in a computer lab. The teaching used a traditional method where the constructs of a programming language (C++
before 2000, and *Visual Basic* from 2000) were introduced (see Table 8-1, right side). Flowcharts and pseudocode were taught as techniques for program design. Desk-checking, testing and debugging were also taught to support novices’ learning.

Although a number of various teaching innovations had been introduced by the author over the years, such as using a sub-set of programming language and visualisation (Hu, 2004), building and applying an object-oriented class library (DLL, Dynamic Link Library) (Hu, 2006), using a simple programming environment (e.g. *Visual Basic*) and gamification (Hu, 2007, 2008), there were no significant changes to the traditional method. Students felt programming is hard and time consuming. Some of them could understand the provided code, but they found it difficult to construct programs.

From 2011, the author taught this course at *EIT Tairawhiti* again, using the proposed new approach described in this thesis. Before the evaluation started, ethical approval was obtained from the University of Otago and was also accepted by EIT. Students were told about the experiment for research into the improvement of teaching programming. They signed consent forms for participants when the first class started (see Appendix C). As before, the course was delivered as a mixture of upfront teaching and exercises in a computer lab. The course outline is given in Table 8-1 (left side).

In 2011 and 201327, using BYOB, the course retained a conventional method for the first half of the course. However, the second half of the course followed the experimental method, teaching the visual goal-plan approach, and the process presented in Chapter 6 (see Table 8-1). The two parts were separated by a mid-term examination at the end of Week 4. The conventional method was taught differently compared to how it used to be (from 2006 to 2009), not only due to differences in programming languages, but also the time frames of progress. After the mid-term examination the new method was introduced into the curriculum following our programming process using the data-flow framework in BYOB. In 2011, we only provided some plans in the plan library. Students needed to modify similar plan blocks or create a new plan block to complete their tasks. For example, we hid the Dividing Plan block, but provided the Multiplying Plan block in the assessment, although it was provided during the teaching time. However, in 2013, all the plan blocks needed for assessment were provided. We

---

27 Due to a range of constraints (such as “leakage” from using the same examination question, and a small group size) we planned not to collect data in 2012.
regard the old and the new teaching methods as independent variables in the experiment, while the examination result for each teaching method is the dependent variable.

Table 8-1 Course Outline by Years*

<table>
<thead>
<tr>
<th>Week</th>
<th>2011 and 2013</th>
<th>2006 – 2009</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Topics</td>
<td>Methods</td>
</tr>
<tr>
<td>1</td>
<td>Introduction</td>
<td>· Computer languages &lt;br&gt; · Sequence: Input, Process, Output &lt;br&gt; · Flowchart &lt;br&gt; · BYOB, desk-checking, and debugging &lt;br&gt; · Pseudocode</td>
</tr>
<tr>
<td></td>
<td>to BYOB</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Making Decision</td>
<td>· What is Selection in &quot;if&quot; statement &lt;br&gt; · Flowchart &lt;br&gt; · Desk-checking &lt;br&gt; · Pseudocode &lt;br&gt; · BYOB</td>
</tr>
<tr>
<td>3</td>
<td>More about Making Decision</td>
<td>· Nesting of Selection &lt;br&gt; · Flowchart &lt;br&gt; · Desk-checking &lt;br&gt; · BYOB and documentation &lt;br&gt; · Pseudocode</td>
</tr>
<tr>
<td>4</td>
<td>Repeating Actions</td>
<td>· What is repetition &lt;br&gt; · Flowchart &lt;br&gt; · Desk-checking &lt;br&gt; · BYOB &lt;br&gt; · Pseudocode &lt;br&gt; · Nesting of repetition</td>
</tr>
<tr>
<td>5</td>
<td>Analysis of Problems</td>
<td>· Analysis of Goals &lt;br&gt; · Design by Plans</td>
</tr>
<tr>
<td>6</td>
<td>Steps of Solving Problems</td>
<td>· Solving Problems by Provided Plans</td>
</tr>
<tr>
<td>7</td>
<td>Put All Together to Solve Problems</td>
<td>· Build Your Own Plans &lt;br&gt; · Solving Problems from Goal to Program</td>
</tr>
<tr>
<td>8</td>
<td>Revision</td>
<td>· Revision</td>
</tr>
<tr>
<td>9</td>
<td>Test</td>
<td>· Written and Practical Test</td>
</tr>
</tbody>
</table>

*In 2011 and 2013, after conventional method was taught in weeks 1-4, the new proposed method was taught in weeks 5-8, which are bold in the table.
As shown in the course outline in Table 8-1, in the first week, we introduced the visual programming language BYOB in the experimental method (see Appendix D). We started from the history of programming and presented the basic steps of input-process-out for problem solving (e.g. compute wages) in a programming flowchart. We then introduced the BYOB programming environment and implemented the flowchart in BYOB. We also introduce desk-checking and debugging to build students’ confidence in their programming results by comparing them with manual results. Finally, we introduced pseudocode to summarise the program.

On the other hand, for the conventional method’s first week we focused on two syntax-free programming notations, flowcharts and pseudocode, for the programming representations rather than directly jumping into an integrated programming environment to start coding.

In the second week of the experimental method we introduced selection using “if” statements. The same programming methods such as flowcharts, desk-checking, debugging, and pseudocode are still applied to the new introduced statement. In the conventional method we started by introducing syntax, semantics, and the programming environment to implement the basic input-process-output in Visual Basic (VB). We also introduced desk-checking and debugging to support learning of VB.

In the third week, for the experimental method, we introduced nesting of “if” statements by using the same programming methods as in the previous two weeks. For the conventional method, because the syntax-free method took an extra week, the “if” statement was introduced a week later than in the experimental method.

In the fourth week, for the experimental method, we introduced the “loop” statement as well as nesting of loops. For the conventional method, it was still behind and worked on nesting of “if” statements. Except for the difference in progress and programming language, there are no other differences for the two methods in the first four weeks.

In the fifth week, for the experimental method, we started introducing the concepts of goals and plans, visual notation of goals and plans, and our data-flow framework. For the conventional method, it then moved to “loop” statements and still applied the same programming methods.
In the sixth week, for the experimental method, we introduced our programming process of using goals and plans to solve problems. For the conventional method, we introduced examples of using mixed selection and repetition to solve problems.

In the seventh week, for the experimental method, we introduced more examples of solving problems by using goals and plans to let students have more practice with the programming process. We also introduced how to build students’ own plan blocks for inclusion in their plan library. For the conventional method, we also introduced more examples of solving problems using mixed selection and repetition. So far, the two teaching methods progress at the same rate, but use different programming methods. Finally, in the eighth and ninth weeks, it is time for both methods to have revision and assessment.

Since we had small groups of students in each year, we considered various experimental designs for choosing samples of the dependent variable grounded by the independent variable (see Table 8-2). If we had a large number of students, we would consider dividing them into two groups to produce unrelated samples (see Design 1 in Table 8-2). One group could be taught by the conventional method and another group could be taught by the proposed method. We could then collect final examination results from both groups by using the same examination questions in order to evaluate how they were affected by different teaching methods. However, we had small groups of students (8 and 14) in 2011 and 2013 respectively. In each year, it was not possible to divide the students into two groups and avoid cross-group communication.

<table>
<thead>
<tr>
<th>Teaching methods in various designs of comparison</th>
<th>Design 1</th>
<th>Design 2</th>
<th>Design 3</th>
<th>Final Design</th>
</tr>
</thead>
</table>

In order to overcome the constraint of having limited participant numbers, we then considered collecting related samples by teaching both methods to the same students (see Design 2 in Table 8-2). For example, we could teach the conventional method to a class before the mid-
term examination and change to the proposed method after the mid-term examination. After collecting sample data from both mid-term and final examinations using similar examination questions, we could directly compare the data to evaluate student performance under different teaching methods (independent variable). The major advantage is that it reduces the background variation of different cohorts of students in different years. However, the serious disadvantage of sampling from the same cohort of students taught by both methods is a profound carry-over effect, which means one method could affect the learning result from another method for the same student. Therefore, we cannot compare results from the same group of participants by teaching the conventional and the proposed methods, even though these results can be statistically significant different.

Since we did not have enough students in each year to use unrelated samples and we were also unable to use related samples in the same year, we had to compare results from the group taught using the new method against those taught using the old method in previous years (see Design 3 in Table 8-2). That is to say we needed to collect unrelated samples from different years. Since the author had taught the same course at the same institute for a few years, we could then compare the final examination results of those students taught by the new method in 2011 and 2013 to those taught by the conventional method from 2006 to 2009 (see Hypotheses 4 in Section 8.1.2).

However, in the above comparison of unrelated samples from different years, there are other differences such as different student cohorts, exam questions, computer languages, and marking standards. Therefore, we defined a number of hypotheses (see Hypotheses 1, 2, and 3 in Section 8.1.2) that correspond to different plausible explanations for variations in student performances due to these confounding factors. Subsequently, our final experimental design integrated the other three possible experimental designs by not only comparing results from 2011 and 2013 to those in 2006-2009, but also comparing results from the final exam in 2011 and 2013 to those in the mid-term in 2011 and 2013 (see Final Design in Table 8-2).

We measured examination performance as an imperfect proxy for assessing learning. Specifically, in order to assess programming ability, we only considered the answers to the programming question collected from the final examinations from 2006 to 2009. For 2011 and 2013, we collected answers from both the final examination and the mid-term test. The examination also had other questions (e.g. filling in the blanks for concepts and segments of
code, and converting between a flowchart and pseudocode) that did not assess the students’ ability to develop a complete program. The students could have done poorly on the programming question, but they still may have done well on the examination overall. At institutes of technology in New Zealand, each student has two chances to pass the examination: test and re-sit (some conditions apply). Where students took this opportunity, we only collected results from their first attempt.

The programming questions used in the examinations were similar and comparable across years, e.g. calculating the sum and (positive or negative) count, or the average of a sequence of numbers (see the programming questions in Appendix A-1). These programming questions were done on a computer, using a programming environment, rather than on paper. We note that these instruments for the evaluation of our experimental method in 2011 and 2013 were based on a question that Soloway (1986) originally used:

Write a program that will read in integers and output their average. Stop reading when the sentinel value (-1) is input.

Similar to other years, for the programming questions in the final examination in 2011 and 2013, we implicitly explained the relationship between the input and the output, by providing an example (e.g. after entering data 1, 2, 3, and -1, the average is 2). Additionally, we explicitly required students to follow the new programming process (as described in Chapter 6). We could then track whether students used the new method rather than the old method. We propose that by following a set of detailed instructions in the 2011 and 2013 final examinations (see Appendix A-1), students actually completed all the steps of our new programming process. From the collection of students’ code files and sample testing results (e.g. a screenshot of testing output) in each phase of the programming process such as expanding, merging, it suggests that students’ programs were developed under the new method rather than by the old method. Although we advocated that the sub-goal labelling approach (see Chapter 7) is a promising method to identify sub-goals in the goal analysis, we did not use it in our teaching practice and evaluation.

All the answers on the programming question were re-marked using a common marking rubric (see the marking schedule in Appendix A-2). The results were independently double checked by one supervisor. The re-marking criteria were:
1. identifying all the variables correctly, for example when calculating an average, the required variables were “number”, “sum” and “count” (weighting 10%);

2. correctly using fragments of key code, for example having code to count the number of values entered. For this criterion, we only assessed the presence of essential code fragments, without requiring them to be combined correctly (weighting 40%);

3. combining code fragments correctly. For this criterion, segments were required to be combined to achieve the required functions although they may still have bugs (weighting 30%); and

4. the final program being tested and bug free. This criterion required that the combination was correct (weighting 20%).

The original marking criteria were based on the completion of program functions and language usage. The function requirements included input (weighting 10%), output (weighting 10%), selection (weighting 10%), repeating (weighting 10%), integration of selections (weighting 20%), and integration of selection and loop (weighting 20%). Language usage covered variables (weighting 10%) and syntax (weighting 10%). Conversely, the new re-marking criteria focused on the basic usage of plan-like fragments (weighting 40%) rather than program functions. The new criteria also paid much more attention to merging of these fragments for the construction of the entire program (weighting 30%) rather than using individual language sentence. The new criteria emphasised testing practice and using debugging tools (weighting 20%) rather than language syntax. Finally, the new criteria also required students to use required variables (weighting 10%).

In summary, the first programming course was delivered by following the outline in Table 8-1. All the examinations were taken in a computer lab. The programming questions were re-marked according to the new criteria and double checked. The data for the conventional method were obtained by re-marking the programming question in the final exam from 2006 to 2009 as well as from the mid-term exam in 2011 and 2013. The data for the proposed new method were obtained from the final exam results in 2011 (initial evaluation) and in 2013 (further evaluation). The distributions of re-marked results and student numbers are shown in Figure 8-1.
8.1.2 Hypotheses of Evaluation

Our analysis aims to determine to what extent the new method for teaching programming made a difference. However, in order to draw conclusions about any difference between the results from the new method in 2011 and 2013 and those from the old method in earlier years, we consider four sources of experimental error and alternative causes of the error in terms of sampling, assignment, conditions and measurement. These four sources present as differences from year to year: different student cohorts, exam questions, computer languages, and marking standards. They are considered as confounding variables. Therefore, in order to eliminate the influence of experimental errors, we develop three additional hypotheses.

Firstly, we test the hypothesis that there is no significant variance within the years 2006-2009. If this hypothesis holds, then it suggests that differences due to variances in the cohort across years, (slightly) different assessment questions and relevant marking schedules are not significant.
Secondly, we hypothesise that there is no significant difference between the examination results in the earlier years and the mid-term examination results in 2011 and 2013. We expect no difference because all followed on from teaching using the traditional teaching method, and this would show that using BYOB on its own makes no significant difference. If true, this suggests that the introduction of BYOB per se was not sufficient to explain any difference. It would also suggest that any difference between the final results from 2011 and 2013 and those from earlier years are not due to differences between the cohorts, exam questions and marking schedule. Since the mid-term examination is earlier than the final examination (2006-2009), we would actually expect worse performance in the mid-term by the students. However, it may be that BYOB did improve things, but that the improvement was “hidden” because the mid-term test was earlier than the final exam in earlier years.

Thirdly, we hypothesise that there is no significant difference between the final exam results in 2011 and 2013. If true, it suggests no significant difference in terms of student cohorts, exam questions and marking schedule when teaching using the experimental method. It would also suggest that any difference between the final results from 2011 and 2013 and those from earlier years are not due to the differences between 2011 and 2013. However, the results may be significantly different between these two years. While students in 2011 had to build or modify a plan block, students in 2013 could use all the plan blocks from the library.

Finally and most importantly, we hypothesise that there is no significant difference between the final examination results in 2011 and 2013 and those in previous years. If the hypothesis does not hold, it suggests that there are significant differences between the new teaching approach (2011 and 2013) and a traditional approach (2006-2009). However, it doesn’t indicate which years cause the significant differences. Therefore, we need further post hoc tests for the identification of these years (see Section 8.3). The details of these four hypotheses are listed below and their relationships are shown in Figure 8-2.

**Hypothesis 1:***

*The null hypothesis (H₀) is that the final examination scores among the years from 2006 to 2009, which used a conventional method, come from populations with identical “locations”. In other words, the mean ranks of these final examination scores are not expected to be significantly different.*
The alternative hypothesis ($H_1$) is that the final examination scores among the years from 2006 to 2009, which used a conventional method, do not come from populations with identical “locations”. In other words, the mean ranks of the final examination scores among the years from 2006 to 2009 are expected to be significantly different.

**Hypothesis 2:**

The null hypothesis ($H_0$) is that the mid-term examination scores from 2011 and from 2013, and the final examination scores among the years from 2006 to 2009, which all used a conventional method, come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the mid-term examination scores from 2011 and from 2013, and the examination scores from 2006 to 2009, which all used a conventional method, do not come from populations with identical “locations”. In other words, the mean ranks of the mid-term examination scores in the year 2011 and 2013, and from the final examination scores among the years from 2006 to 2009 are expected to be significantly different.

**Hypothesis 3:**

The null hypothesis ($H_0$) is that the final examination scores from 2011 and from 2013, which used the experimental method, come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the final examination scores from 2011 and from 2013, which used the experimental method, do not come from populations with identical “locations”. In other words, the mean ranks of final examination scores in the years of 2011 and 2013 are expected to be significantly different.

**Hypothesis 4:**

The null hypothesis ($H_0$) is that the final examination scores when using a conventional method (2006 to 2009) and those when using the experimental method (in 2011 and
come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the final examination scores when using a conventional method (2006 to 2009) and those when using the experimental method (in 2011 and 2013) do not come from populations with identical “locations”. In other words, the mean ranks of final examination scores based on the experimental method and a conventional method are expected to be significantly different.

Figure 8-2 The relationships of four hypotheses

8.1.3 Effect Size

In inferential statistics, we must measure a $p$-value\(^{28}\) for our null hypothesis significant testing (NHST). However, the statistical significance only means that we can be confident that our result is unlikely to be due to random variation in samples. In other words, the $p$-value gives us the likelihood that our result is not due to chance. However, it does not tell us about the strength of the relationship between dependent variable and independent variable.

"Whereas a test of statistical significance is traditionally used to provide evidence (attained $p$-value) that a null hypothesis is wrong, an effect size (ES) measures the degree to which such a null hypothesis is wrong (if it is wrong)” (Grissom & Kim 2012, p5). The effect size reflects how large the effect of an independent variable was. The larger the effect size, the more influence there is on the examination score by the relevant teaching method.

---

\(^{28}\) A $p$-value is “the probability of observing a difference as extreme as your sample (or more so) if the null hypothesis were actually true” (Siegel, 1990, p356). The lower the $p$-value, the more likely an observed difference is caused by a real difference rather than by chance. Normally, a $p$-value less than .05 is considered to be statistically significant against the null hypothesis.
8.1.4 Power Analysis and Predicting Sample Sizes

Emphasis on research design to ensure acceptable levels of statistical power is advocated in software engineering experiments, particularly in the poorly reported discipline of information systems research (Dybå, Kampenes, & Sjøberg, 2006). The power of a statistical test is the probability of rejecting a null hypothesis when the null hypothesis is false. It is inversely related to beta (β) level or the probability of making a Type II error (i.e. power = 1 – β). There are four parameters in a power analysis: alpha level, power, sample size, and effect size. Any one of the parameters can be calculated from the other three parameters. According to Cohen (1992), an accepted level of power should be .80 for a study to be worth conducting. Therefore, we can predict our sample size in order to have enough power for the detection of a significant effect at α = .05 level.

Before we make any assumptions regarding our data distribution, we choose a one way ANOVA test in G*Power to analyse statistical power because of its general tolerance to deviations from the normal distribution (Glass, Peckham & Sanders 1972). The summary of remarked exam results and student numbers are shown in Table 8-3. There are a total of 72 sample points from six years.

Table 8-3 Summary of student results

<table>
<thead>
<tr>
<th>Method</th>
<th>Year</th>
<th>N</th>
<th>Mean</th>
<th>95% Confidence Interval for Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lower Bound</td>
</tr>
<tr>
<td>Conventional</td>
<td>2006</td>
<td>13</td>
<td>33.31</td>
<td>9.49</td>
</tr>
<tr>
<td>Method</td>
<td>2007</td>
<td>16</td>
<td>53.81</td>
<td>30.57</td>
</tr>
<tr>
<td></td>
<td>2008</td>
<td>13</td>
<td>36.77</td>
<td>7.78</td>
</tr>
<tr>
<td></td>
<td>2009</td>
<td>8</td>
<td>39.38</td>
<td>1.89</td>
</tr>
<tr>
<td></td>
<td>Mid-term 2011</td>
<td>7</td>
<td>52.43</td>
<td>21.00</td>
</tr>
<tr>
<td></td>
<td>Mid-term 2013</td>
<td>14</td>
<td>31.14</td>
<td>15.45</td>
</tr>
<tr>
<td>Experiment</td>
<td>2011</td>
<td>8</td>
<td>84.75</td>
<td>61.55</td>
</tr>
<tr>
<td>Method</td>
<td>2013</td>
<td>14</td>
<td>91.00</td>
<td>82.88</td>
</tr>
</tbody>
</table>

29 Beta (β) level is the probability of making the wrong decision (i.e. Type II error) when the specific alternate hypothesis is true.

30 A Type II error is the probability of making the wrong decision when the specific alternate hypothesis is true.

31 Alpha (α) level is an error rate that you are willing to accept. It is also known as the Type I error rate and set as .05 or .01. The significance level α is the probability of making the wrong decision when the null hypothesis is true.

32 G*Power 3.1.9: http://www.gpower.hhu.de/en.html
According to Cohen (1992), the effect size conventions for the ANOVA test are: 1) $0.10 \leq \text{effect size} < 0.25$ for a small effect; 2) $0.25 \leq \text{effect size} < 0.40$ for a medium effect, and effect size $\geq 0.40$ for a large effect size. Otherwise, the influence by the independent variable will be inconsiderable when the effect size < 0.10. If we consider six groups of students’ results chronologically, in order to have enough power (.80) at .05 significance level, the relationship between effect size and sample size is shown in Table 8-4.

<table>
<thead>
<tr>
<th>Effect size</th>
<th>Sample Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (0.10)</td>
<td>1290</td>
</tr>
<tr>
<td>Medium (0.25)</td>
<td>216</td>
</tr>
<tr>
<td>Large (0.40)</td>
<td>90</td>
</tr>
</tbody>
</table>

$\alpha = 0.05$ and power = 0.80

However, in Table 8-4, we predict that we need to have at least 90 participants from these six groups in order to meet Cohen’s conventions of having enough power (.80), a large effect (.40), and at .05 significance level (see Appendix B-1). Recall that the $p$ value only indicates statistical significance whereas the effect size suggests substantive significance for understanding the magnitude of difference. We need to have a large enough sample size so that if there was a sufficient effect size and an acceptable statistical power (follow Cohen’s convention), we would be able to detect a valid $p$-value in our null hypothesis significance testing.

Moreover, due to the class size being too small and it being impractical to avoid communication between the groups, we were unable to divide each group for comparison between the effects of the old and new teaching approaches. Subsequently, we had to compare results from the group taught using the new approach against those taught using the old approach in previous years. Therefore, we have to consider chronologically combining the six small groups into three large groups based on the teaching method as two control groups and one experimental group for students in the final exam results. We also considered an extra control group for students in the mid-term exam results in 2011 and 2013 (see Table 8-5).
Table 8-5 Summary of combined student groups and results

<table>
<thead>
<tr>
<th>Method</th>
<th>Group</th>
<th>Total</th>
<th>Mean</th>
<th>95% Confidence Interval for Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lower Bound</td>
</tr>
<tr>
<td>Conventiona I Method</td>
<td>Group 1 (2006 &amp; 2007)</td>
<td>29</td>
<td>44.62</td>
<td>28.52</td>
</tr>
<tr>
<td></td>
<td>Group 3’ (Mid-term 2011 &amp; Mid-term 2013)</td>
<td>21</td>
<td>38.24</td>
<td>22.34</td>
</tr>
<tr>
<td>Experiment Method</td>
<td>Group 3 (2011 &amp; 2013)</td>
<td>22</td>
<td>88.73</td>
<td>79.99</td>
</tr>
</tbody>
</table>

We argue that combining two groups with the same treatment is valid not only because the chronologically combined two groups (taught by the same method) are similar in most aspects (e.g., similar backgrounds, examination questions and marking standard, plus the same computer language), but also because every pair of groups with the same treatment are not significantly different (according to our initial study (Hu, Winikoff, & Cranefield, 2012, 2013)). Finally, through the combination, the advantage is that our actual sample size is enough to carry on for further tests because it meets Cohen’s conventions of statistical power analysis in terms of p-value, power, and effect size. In this case, the combined groups indicate having better power than using the groups without combining.

After combining into three groups, we discover that we need only 66 or more participants to achieve Cohen’s conventions and that the power can even reach .82 according to G*Power (see Table 8-6). Furthermore, having a total of 72 students, we could expect not only to achieve Cohen’s conventions, but also to get even more power of .85 in the further statistical tests.

Table 8-6 The detection of relationship between effect size and sample size for three groups in the ANOVA

<table>
<thead>
<tr>
<th>Effect size</th>
<th>Minimum Sample Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (0.10)</td>
<td>969</td>
</tr>
<tr>
<td>Medium (0.25)</td>
<td>159</td>
</tr>
<tr>
<td>Large (0.40)</td>
<td>66</td>
</tr>
</tbody>
</table>

α = 0.05 and power = 0.80

Moreover, considering the minimal detectable effect (if there is a significant difference in our experiment), we need to determine what level of effect we could find from our combined data. Using G*Power, we detect the effect size from Means of our sample data in three groups.
(see Appendix B-1). According to the result (0.57 > 0.40), we conclude that the effect size that could be detected in our experiment is a large effect.

Finally, according to the minimal detectable effect size (0.57) from the sample size (72) and given $\alpha$ (5%), we compute the achieved power is 99.3% using G*Power (see Appendix B-1). That is to say if there is a large effect size from our method, our experiment should be able to detect it.

Therefore, the effect size detected in our pilot study was a large effect size, so we designed the full experiment to find a large effect. Next, we discuss whether or not we should use ANOVA before we conduct our experiment.

**8.1.5 Detecting and Measuring Normality**

Many commonly used statistical methods of analysis such as the t-test and ANOVA test require certain conditions such as that observations must be independent, samples must be normally distributed, and samples must have equal variances (Siegel, 1957). However, in the polytechnic and institute environments, the distributions of students from year by year can be greatly diverse in terms of academic backgrounds, range of ages, attitude of learning, etc. Consequently, their performance is not expected to follow a normal distribution. Although ANOVA is robust against the non-normal distribution, platykurtic distributions (with low degree of peakedness of the bell distribution shape) on small sample sizes can cause a profound effect on both the Type I error rate and power (Glass, Peckham & Sanders, 1972). Therefore, we consider whether ANOVA might still be applicable. Before proceeding to the null hypothesis significant testing, it is important to verify whether our data has met the assumptions of distribution required for using ANOVA. Otherwise, non-parametric methods are the options.

Firstly, we examine the non-normality of distribution primarily using a graphical representation of the data. If it has a normal distribution, the ANOVA test could be then directly used. However, if graphical techniques indicate a non-normal distribution, we can use statistical tests as supplements to measure how much our data differ from normality. The graphical techniques we used for detecting non-normality are histograms, boxplots, Q-Q plots and P-P plots. The measures of non-normality are skewness, kurtosis, Kolomogorov-Smirnov test and
A histogram shows how many times a given value appears in the sample. The shape of the normal distribution is symmetrical with a mound in the centre that gradually falls off to both left and right, which looks like a bell. However, none of four histograms of our individual group sample data are presented in a “bell” shape in Figure 8.3 by using IBM SPSS Statistics version 22 to fit a normal distribution. Particularly inspecting box plots (see Figure 8.4), we notice that in the group from years 2011 and 2013 there are two cases of outliers at the lower range of the data in graph. We also see that the median line does not evenly divide the box and the lower tail of the box plot is longer than the upper tail. Therefore, we detect the data distribution of the group in 2011 & 2013 is skewed to the left.

Our purpose is to detect the population distribution from which the data for each hypothesis test were collected. Therefore, having verified that all the numbers are entered correctly, we then proceed to take a further glance at four P-P and Q-Q plots by different combinations of these four groups in our four hypotheses in order to detect the skewness of population distribution. For example, by using the combination of three sample groups (Group 1, Group 2, and Group 3) for Hypothesis 4 for comparison of both old and the new methods (see Figure 8.5), the substantial deviation from the straight line in the plots suggests a skew in the distribution.

33 In box plots (also called box-and whisker plots), data are sorted and divided into four equal size groups (i.e. quartiles). The median value (middle quartile) is the line dividing the box into two parts. The data above or below the line represent that its value is greater or less than the median value. The top hinge of the box is the upper quartile while the bottom hinge of the box is the lower quartile. The top and the bottom of the lines (or whiskers) represent both maximum and minimum values. In SPSS, a dot “.” represents that an outlier falls more than 1.5 box lengths from the lower or upper hinge of the box (inner fence) while an asterisk “*” means an “extreme” outlier by identifying value more than 3 box lengths (outer fence) from either hinge.

34 Case 9 is an outlier value beyond the inner fence and case 21 is an outlier far outside the outer fence.

35 Skewness measures the degree of asymmetry of a distribution around its mean. When skewness is zero, the distribution is symmetrical as normal distribution. Positive skewness indicates an asymmetric tail extending towards positive values (right skewed) while negative skewness indicates an asymmetric tail extending towards negative values (left skewed).

36 The two plots are: a) Normal probability plot (Probability-probability plots or P-P plot): the expected cumulative probability against the observed cumulative provability; b) Quantile–Quantile (Q-Q) plot: is a plot of percentiles of a standard normal distribution against these of the observed data, i.e. the expected values against the observed values. For a normal distribution, the points in both a P-P plot and a Q-Q plot fall on a straight line. Otherwise, both plots are in a systematic deviation away from the line in an S-like pattern.
Figure 8-3 Histograms of four sample groups

Figure 8-4 Example of box plot showing outliers from individual group
Although we could conclude that the distribution based on the combination of the above three groups is non-normal, we concern the limitation of uncertainty by the visual method. Therefore, we explore further quantitative determination on the confidence of our conclusion in terms of skewness, kurtosis, and normality tests.

In the above example of combination by three groups for Hypothesis 4, the results show that the 95% confidence interval of the skewness score and kurtosis score ranges from -0.053 to -0.619, and from -1.214 to -2.332, respectively (see Appendix B-2). In both cases, the value of zero is out of the above bounds. We can then reject the null hypothesis that our statistic is not significantly different from the value of zero. Moreover, since the kurtosis value (-1.773) is greater than ±1 (i.e. from -1.214 to -2.332), the kurtosis for the distribution is outside the range of normality. Therefore, it is not a normal distribution because it excludes the possibility of zero values for either skewness or kurtosis.

Furthermore, since the kurtosis value (from -1.214 to -2.332) is less than zero, the data distribution is “flat” (termed “platykurtic”), which has a low degree of peakedness. According to Glass, Peckham, and Sanders (1972), with platykurtic distributions, the probability of a Type I error (α) is slightly higher than the normal α value. Furthermore, for a platykurtic distribution with a very small sample size, the actual power is less than the nominal power. Therefore, even though the t-test and ANOVA test can be tolerant of non-normality in

---

37 Kurtosis is used to measure the relative peakedness (i.e. narrow) or flatness (i.e. broad) of data distribution comparing to the normal distribution. Normal distributions produce a zero value of kurtosis. While positive kurtosis indicates a distribution more peaked than a normal distribution, named leptokurtic, negative kurtosis specifies a flatter distribution than a normal distribution with a wider peak, called platykurtic.

38 Type I error (α) is the probability of making the wrong decision when the null hypothesis is true.
determining significance levels (Glass, Peckham & Sanders 1972), platykurtosis on small sample sizes can cause a profound effect on both Type I error and power.

After measuring skewness and kurtosis, further statistical tests of the $p$-value of hypotheses (cf. Section 8.1.2) for deviation from normality are needed by using Kolomogorov-Smirnov test and Shapiro-Wilk test. For example, the results of both the Kolmogorov-Smirnov and the Shapiro-Wilk tests for the above combination of three groups ($< .001$, see Appendix B-2) indicate that the data are from a non-normal distribution. Similarly, the tests of normality by various combinations of sample groups under all the other three hypotheses show that assumptions of normality of our data must be rejected. All the results are less than the conservative alpha level, $p < .001$ (see Appendix B-2). Therefore, instead of assuming that our sample data comes from a normal distribution (or transforming our sample data into a normal distribution), two relevant non-parametric statistic tests, the Mann-Whitney U Test and the Kruskal-Wallis H Test, can be used for null hypothesis significance testing (NHST), which do not require a normal distribution.

In summary, using graphical techniques such as histograms, outliers in box plots, Q-Q and P-P plots, we detect non-normal distribution of our data. Having further measurement by skewness, kurtosis, Kolomogorov-Smirnov test and Shapiro-Wilk test, we identify the skewness, platykurtic and non-normality of our data distribution. Finally, although some parametric methods are tolerant to non-normal data, we decided to use non-parametric methods to test the hypotheses on our non-normally distributed platykurtic small sample data.

**8.2 Null Hypothesis Significance Testing**

**8.2.1 Non-parametric Test**

Specifically, we choose the Kruskal-Wallis H Test (K-W H test) for the null hypothesis that a set of independent samples come from the same identical distributions. The K-W H test is based on the ranks of the data rather than on the actual data as for the ANOVA test. We also use the Mann-Whitney U Test (Mann-Whitney rank sum test, or M-W U test) for the comparison of
two independent samples. The selection of both the K-W H\textsuperscript{39} test and M-W U\textsuperscript{40} tests for the comparisons of our sample data is based on the following assumptions:

1) The dependent variable of student examination scores is either continuous or ordinal. Students’ examination scores are numbers from 0 to 100 which satisfies the requirement;

2) The independent variable (teaching method) consists of two categorical independent groups (for the M-W U test), or more than two categorical independent groups (for the K-W H test). Each test we conduct compares groups of examination marks from students who studied subsets of the years considered in the study; thus the assumption of the of independence is satisfied (cf. Table 8-5); and

3) The independence of observations requires that no students are in more than one group. We notice that a few students re-entered into the course in different years. We only considered their first time enrolment as the key category to assign them into each relevant group. That is to say, they were excluded from consideration in subsequent years.

There is another assumption that is sometimes used in the K-W H and M-W U tests: that the distributions in all groups have the same shape. This is needed when these tests are used to compare the medians of the groups. However, the interpretation of the differences depends on whether the distributions have identical shapes but are shifted, or whether they may have a different shape. In this case we have no reason to expect that the distributions of our students’ performance would have the same shape across different years, with any difference being solely due to a shift in the median. That is to say, there is no need for any additional assumptions for testing differences in both the M-W U test and the K-W H test. Therefore, we need to interpret a finding of significant difference as being about the distribution in general, rather than specifically about its shape, and hence mean ranks from each groups are compared by the test results by the K-W H and by the M-W U tests (see Section 8.2.2).

Note that the K-W H test is an “omnibus” test, which means that a single test is used to compare a number of samples. It is common to follow a significant omnibus test (i.e. if the null hypothesis is rejected) by a family of pairwise tests to gain more precise information about the

\textsuperscript{40} https://statistics.laerd.com/spss-tutorials/mann-whitney-u-test-using-spss-statistics.php
causes of the significant difference. These pairwise tests are “post hoc” tests that were not pre-determined by the experimental design, and in this case it is necessary to adjust the individual test threshold values to reduce the overall chance of obtaining any false positive results (Type I errors) across the family of tests. In order to reduce the chance of Type I errors, the adjustment can be done by using various forms of the “Bonferroni correction”. We use the Bonferroni-Holm method (Holm, 1979), which is less conservative than some other forms of adjustment.

In the first three hypotheses, we assume that there is no significant difference for the group of years when using the same teaching method. If there is a significant difference between the results when using different teaching methods (Hypothesis 4), we would perform a post-hoc test using the Bonferroni-Holm method for pair-wise comparisons to identify which group of years contributes to the significant difference. The tests for the four hypotheses (cf. Figure 8-2) are summarised in Table 8-7.

<table>
<thead>
<tr>
<th>H</th>
<th>Control group</th>
<th>Experimental group</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>Group 1 (Years 2006 &amp; 07) and Group 2 (Years 2008 &amp; 09)</td>
<td></td>
<td>M-W U test</td>
</tr>
<tr>
<td>H2</td>
<td>Group 1 (Years 2006 &amp; 07), Group 2 (Years 2008 &amp; 09), and Group 3' (Mid-term 2011 &amp; 13)</td>
<td></td>
<td>K-W H test</td>
</tr>
<tr>
<td>H3</td>
<td>Within Group 3: Year 2011 and Year 2013</td>
<td></td>
<td>M-W U test</td>
</tr>
<tr>
<td>H4</td>
<td>Group 1 (Years 2006 &amp; 07) and Group 2 (Years 2008 &amp; 09)</td>
<td>Group 3 (Year 2011 &amp; 13)</td>
<td>K-W H test</td>
</tr>
<tr>
<td></td>
<td>Post hoc test</td>
<td></td>
<td>M-W U test with Bonferroni-Holm correction</td>
</tr>
</tbody>
</table>

- In the Table 8-7, each Hypothesis is tested by a non-parametric method (M-W U test or K-W H test);
- When using the H test, we face a choice between an asymptotic test and an exact test41. According to threshold values of sample sizes (N) and number of samples (k) (Siegel & Castellan Jr, 1988), an asymptotic test needs more than three sample groups (k > 3). We should choose an exact test when having three groups in the

---

41 Without combining the original six groups into three groups, we still need to use an exact test or Monte Carlo test when using a Mann-Whitney U test. The asymptotic test requires N1 = 3 or 4 and N2 > 12; or N1 > 4 and N2 > 10. We have only eight students (N2 ≤ 10) in both 2009 and 2011, which makes it impossible to use asymptotic testing when using a M-W U test.
test. An exact test also fits our use of a nonparametric test as well as our small data set containing many ties, e.g. a lot of marks of 100. However, SPSS failed to support the exact test option for the H test. Therefore, we used a Monte Carlo test for the simulations in order to approximate the real results;

- **Hypotheses 1, 2, and 3** are tested to reduce the chances of experimental error and alternative causes from the four sources of differences in student cohorts, exam questions, computer languages, and marking standards;
- **Hypothesis 4** is tested to find whether the dependent variable (final examination result) is influenced by the independent variable (teaching method);
- For Hypothesis 4, if the p-value is significant from the K-W H test, post hoc tests are applied to reduce the chances of a Type I error. Therefore, a Bonferroni-Holm correction is used in conjunction with M-W U test for the non-parametric method. Pairwise comparisons are tested by an M-W U test between two control groups and one experimental group;
- The effect size is advocated to be reported following up a statistically significant nonparametric p-values (Leech & Onwuegbuzie, 2002). However, K-W H test only provides a chi-squared value that is not straightforward to be converted into an effect size. Therefore, we calculate the effect size for the post-hoc comparisons when using the M-W U test;
- When using the M-W U test, we ensure that our sample size (Group 1: 29, Group 2: 21, and Group 3: 22) are large enough\(^\text{42}\) so that the sampling distribution of U can be approximated normal. That is to say, our sample sizes allow the appropriate Z value from SPSS results to be used for the estimation of effect sizes (Fritz, Morris, & Richler, 2012); and
- When the group samples sizes are large enough for the test statistic U to approximate a normal distribution, we can calculate effect size using the Pearson correlation coefficient as \(r = \frac{Z}{\sqrt{N}}\). According Cohen (1992), the effect size conventions of the Pearson correlation coefficient are: 1) \(.10 \leq \text{effect size} < .30\) for a small effect; 2) \(.30 \leq \text{effect size} < .50\) for a medium effect, and effect size \(\geq .50\)

\(^{42}\) Regarding the sample sizes in the M-W U test, Harnett and Murphy (1985) emphasised that one of the sample sizes is larger than 20 and the two sample sizes are not too different in size. Siegel (1990) complemented the conditions needed for U to closely approximate a normal distribution. He suggested that each of the two unpaired samples is more than 10. Again, Harraway (1993) supplemented the conditions that both sample sizes are larger than or equal to 15.
for a large effect size. Otherwise, the influence by the independent variable will be inconsiderable when the effect size < .10.

8.2.2 Results

**Hypothesis 1:**

The null hypothesis ($H_0$) is that the final examination scores among the years from 2006 to 2009, which used a conventional method, come from populations with identical “locations”. In other words, the mean ranks of these final examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the final examination scores among the years from 2006 to 2009, which used a conventional method, do not come from populations with identical “locations”. In other words, the mean ranks of the final examination scores among the years from 2006 to 2009 are expected to be significantly different.

We choose the exact test option for the Mann-Whitney U test in SPSS to test Hypothesis 1 (see Appendix B-3). The results of the Mann-Whitney U test show that the difference of examination scores between Group 1 (Year 2006 and 2007, Mean Rank: 26.91) and Group 2 (Year 2008 and 2009, Mean Rank: 23.69) is not significant, $U(29, 21) = 266.5$ ($Z = -.775$); exact $p = .45$ (two-tailed). We conclude that there is not enough evidence to reject the null hypothesis. In other words, the examination results are similar in the past years when teaching by the conventional method although there are differences of student cohorts, variations of examination questions and relevant marking schedules.

**Hypothesis 2:**

The null hypothesis ($H_0$) is that the mid-term examination scores from 2011 and from 2013, and the final examination scores among the years from 2006 to 2009, which used a conventional method, come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the mid-term examination scores from 2011 and from 2013, and the examination scores from 2006 to 2009, which used a conventional
method, do not come from populations with identical “locations”. In other words, the mean ranks of the mid-term examination scores in the year 2011 and 2013, and from the final examination scores among the years from 2006 to 2009 are expected to be significantly different.

The Kruskal-Wallis H test shows that although there are differences from the student results in the past (Group 1 Mean Rank: 37.41 and Group 2 Mean Rank: 32.60) and those from the middle term examination in 2011 and 2013 (Group 3’ Mean Rank: 37.45), these are not significant ($H(2) = .84, p = .66$). In other words, the examination results are similar in each year when still teaching by the conventional method although there are differences of student cohorts, variances of examination questions and relevant marking schedules, and a change in computer language used in 2011 and 2013.

**Hypothesis 3:**

The null hypothesis ($H_0$) is that the final examination scores from 2011 and from 2013, which used the experimental method, come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the final examination scores from 2011 and from 2013, which used the experimental method, do not come from populations with identical “locations”. In other words, the mean ranks of final examination scores in the years of 2011 and 2013 are expected to be significantly different.

Once again, we choose an exact test for the Mann-Whitney U test in SPSS to test Hypothesis 3. Effect is calculated as $r = Z / \text{square root (N)} = -.037 / \sqrt{22} = -.008$. The results show that the difference of examination scores between 2011 (Mean Rank: 11.44) and 2013 (Mean Rank: 11.54) is not significant, $U (8, 14) = 55.5 \ (Z = -.037)$; exact $p = .99$ (two-tailed). We conclude that there is not enough evidence to reject the null hypothesis. The ability of students to create and modify plans is similar in 2011 and 2013. In other words, it suggests that the examination results are similar in the years when teaching by the experimental method

---

43 Using a Kruskal-Wallis H test, since SPSS failed to perform an exact test because of running out of computer memory, we choose the Monte Carlo option instead to test Hypothesis 2. There is no need to report the effect sizes from its post-hoc tests because its $p$-value is $>.05$. 

242
although there are differences of student cohort and variations of examination questions and relevant marking schedule.

**Hypothesis 4:**

The null hypothesis ($H_0$) is that the final examination scores when using a conventional method (2006 to 2009) and those when using the experimental method (in 2011 and 2013) come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis ($H_1$) is that the final examination scores when using a conventional method (2006 to 2009) and those when using the experimental method (in 2011 and 2013) do not come from populations with identical “locations”. In other words, the mean ranks of final examination scores based on the experimental method and a conventional method are expected to be significantly different.

We also choose the Monte Carlo option for the Kruskal-Wallis H test to test Hypothesis 4. The results show that the final examination scores based on both the experimental (Group 3 Mean Rank: 53.11) and conventional methods (Group 1 Mean Rank: 30.43 and Group 2 Mean Rank: 27.48) are significantly different ($p < .001$). Following the discussion in Section 8.2.1, we have to verify which teaching method is associated with the significantly different results. We carry on with a post hoc test for testing Hypothesis 4 by using the M-W U test with Holm’s sequential Bonferroni method (Holm, 1979), i.e. the Bonferroni-Holm correction. Because there is no significant difference for the years by using the same teaching method (see results from Hypothesis (1)), the pair wise comparisons are only designed between the years that use different teaching methods (see Table 8-6). Moreover, when the degree of freedom is more than one, the overall effect size of the K-W H test is inconsiderable. Instead, we considered the effect size for each pair of groups that differs significantly by the post-hoc M-W U test.

In Table 8-6, we summarise the two hypotheses for the pair wise comparisons. Each hypothesis (Hypothesis 4‘ and Hypothesis 4”) is used for the comparison of final examination results from two groups, which were taught by different teaching methods (i.e. conventional and experimental methods), respectively. The hypotheses are described as follows.
### Table 8-8 Post-hoc tests by using M-W U test with Bonferroni-Holm correction

| Hypotheses | Pairwise Comparisons | Threshold p-value \( p = .05/(n - i + 1) \) | p-value from M-W U Test by SPSS Exact Test | Effect Size in M-W U Test 
\( r = \frac{Z}{\sqrt{N}} \) |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hypothesis 4’</td>
<td>Group 2 (Years 2008 &amp; 2009) vs Group 3 (Years 2011 &amp; 2013)</td>
<td>.025(=.05/2)</td>
<td>(&lt; .001 ) (0.00087)</td>
<td>-3.781/sqrt(43) = -.5766 (large effect, &gt; .5)</td>
</tr>
<tr>
<td>Hypothesis 4’’</td>
<td>Group 1 (Years 2006 &amp; 2007) vs Group 3 (Years 2011 &amp; 2013)</td>
<td>.050(=.05/1)</td>
<td>(&lt; .001 ) (0.000016)</td>
<td>-4.113/sqrt(51) = -.57594 (large effect, &gt; .5)</td>
</tr>
</tbody>
</table>

#### Hypothesis 4’:

The null hypothesis (\( H_0 \)) is that the final examination scores of Group 2 (Years 2008 and 2009, using a conventional method) and those of Group 3 (Years 2011 and 2013, using experimental method) come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis (\( H_1 \)) is that the final examination scores of Group 2 (Years 2008 and 2009, using a conventional method) and those of Group 3 (Years 2011 and 2013, using the experimental method do not come from populations with identical “locations”. In other words, the mean ranks of the final examination scores based on the experimental method and those based on the conventional method are expected to be significantly different.

#### Hypothesis 4’’:

The null hypothesis (\( H_0 \)) is that the final examination scores of Group 1 (Years 2006 and 2007, using conventional method) and those of Group 3 (Years 2011 and 2013, using experimental method) come from populations with identical “locations”. In other words, the mean ranks of these examination scores are not expected to be significantly different.

The alternative hypothesis (\( H_1 \)) is that the final examination scores of Group 1 (Years 2006 and 2007, using a conventional method) and those of Group 3 (Years 2011 and 2013, using experimental method) do not come from populations with identical “locations”. In other words, the mean ranks of examination scores based on the experimental method and those based on the conventional method are expected to be significantly different.
According to the Bonferroni-Holm method, the p-value (from smallest to largest) for the Mann-Whitney U Test for each paired comparison needs to be smaller than its threshold p-value to be significant. The results in Table 8-8 demonstrate significant differences of examination scores from both paired tests. The estimated effect sizes of post-hoc comparison are calculated in Table 8-8. The absolute values of both estimated effect sizes are both .58 (> .50), classified as large effects, and all effects are reported at a .05 significance level. For Hypothesis 4', the examination scores from the experimental group, Group 3 (Years 2011 and 2013) (Mean Rank: 28.89) is significantly higher than that from the control group, Group 2 (Years 2008 and 2009) (Mean Rank: 14.79), U (22, 21) = 79.50, r = -.58. For Hypothesis (4''), Groups 3 (Years 2011 and 2013) (Mean Rank: 35.73) is also significantly higher than the other control group, Group 1 (Years 2006 and 2007) (Mean Rank: 18.62), U (22, 29) = 105.00, r = -.58.

In summary, the test results from Hypothesis 4 indicate that there is a significant difference between groups taught by a conventional teaching method (2006-2009) and those taught using the experimental method (2011, 2013). When teaching by the conventional method, the test results from Hypotheses 1 and 2 suggest that there is no significant difference between the groups, which suggests that variations, e.g. in cohort across years, are not significant. Moreover, the test results from Hypothesis 2 suggest that there is no significant difference from having different examination questions, marking schedules, and computer languages. On the other hand, when teaching by the experimental method, the test results from Hypothesis 3 suggest that there is no significant difference between 2011 and 2013. The test results from Hypotheses 1, 2, and 3 can be summarised that when using the same teaching method, there is no significant difference in students’ performance. Furthermore, the test results from the post-hoc test for Hypothesis 4 support the indication from Hypothesis 4 that there is significant difference when teaching by different methods and also suggests that the significant difference is due to the teaching by the experimental methods. Therefore, we conclude that our experimental method in both year 2011 and year 2013 comparing to conventional method from year 2006 to year 2009 (in Hypothesis 4) did produce a significant difference at p=.05, with a large size effect and enough power (at least .80).
8.3 Threats to Validity

We now discuss threats to the validity of our experiment in terms of internal validity (i.e. interpretability) and external validity (i.e. generalisability). We start by identifying threats to internal validity to ensure that the difference of students’ performance in our conclusion is unlikely to be ascribed to factors other than our proposed method. According to the category of identified threats to internal validity (Onwuegbuzie, 2000), we mainly discuss those that may cause plausible rival explanations to our experimental results in terms of testing, instrumentation, statistical regression, dropout, implementation bias and researcher bias (see Table 8-9).

<table>
<thead>
<tr>
<th>Category</th>
<th>Threats</th>
<th>Mitigation / Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing</td>
<td>Uncontrolled assessment without time limit may produce better results.</td>
<td>• Controlled exam with time constraints can reduce the chance of students receiving significant assistance from others; • Having the exam in a computer lab directly provides evidence of students’ ability of practice.</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>Lacking appropriate consistency may cause invalid students’ scores</td>
<td>• Using similar programming questions; • Using the same re-marking criteria; • Have at least two marking the same questions.</td>
</tr>
<tr>
<td>Statistical regression</td>
<td>students’ extremely low or extremely high scores in both methods</td>
<td>• Caused by a “ceiling effect” • A new instrument may have better results</td>
</tr>
<tr>
<td>Dropout</td>
<td>Causing bias to one method because of losing students</td>
<td>• Checked, and there was only one student who missed out the mid-term exam.</td>
</tr>
<tr>
<td>Implementation bias researcher bias</td>
<td>Bias of teaching by the same researcher</td>
<td>• Using historical data from before invention of the experimental method</td>
</tr>
</tbody>
</table>

There are some limitations in the evaluation. We use examinations as an imperfect proxy measure for programming ability. Regarding testing, the argument could be made that students may perform better without time limits. Another limitation is that we only use a programming question in an examination to measure performance. Students who had done poorly on the programming question may still have done well on the other questions that did not assess the students’ ability to develop a complete program. In all years (2006-2009, 2011 and 2013 in the mid-term and final examinations) the programming question was done on a computer, using a programming environment, rather than on paper. We argue that an examination in a computer lab is an appropriate choice because it is conservative in directly providing evidence of students’ ability of practical programming. The controlled examination has time constraints on individual performance. It eliminates the possibility that exists in
uncontrolled assessments that significant assistance can be obtained from classmates, friends, family, or tutors.

Instrumentation with a lack of appropriate consistency (i.e. low reliability) may cause invalid students’ scores. For example, marking may not be consistent from the first student to the last one, or not consistent from two or more data collectors. We used similar programming questions from year to year, such as calculating the sum and (positive or negative) count, or the average of a sequence numbers. We argue that our marking schedules for re-marking different exam papers were based on the same criteria at the appropriate level of consistency in difficulty. We also had two people (the author and one of the supervisors) to independently remark students’ scores using the same marking schedule. These efforts support yielding valid scores for the evaluation.

The threat of statistical regression is relevant to students’ extremely low or extremely high scores in conventional and experimental methods. We argue that the extreme results from our students are because of the following “ceiling effect”. When using the experimental method in 2011 and 2013, there were a number of students who scored 100% (including five out of eight in the 2011 cohort, and seven out of fourteen in the 2013 cohort). The ceiling effect here is a bunching of scores at the upper level, which is not measured because our data-gathering instrument (i.e. exam) was too easy relative to the students’ skills. Note that we were constrained in that we needed to use an equivalent instrument i.e. similar questions to those in the experimental method, when teaching using a conventional method. We argue that this ceiling effect actually reduces the effect size of the difference between the experimental and conventional groups. If we had used an instrument that did not have a ceiling effect, there would be a more significant improvement in the above statistical results.

Considering the issue of students drop out, it may affect our results collected from either conventional or experimental group. In 2011, the loss of one participant from the mid-term examination (n = 7) compared to the final examination (n = 8) (see Table 8-3) does not necessarily produce a bias. In the combined groups for our statistical tests, the mid-term examination group (G3’) is 21 and the final examination group (G3) is 22 (see Table 8-5). Therefore, losing one participant in the mid-term is unlikely to be considered as a great loss in the control group. Moreover, this group is not directly used to compare with our intervention group (G3). It is only used to rule out other confounding factors by comparing to G1 and G2.
The potential issues with the evaluation are implementation bias and researcher bias. The course was taught by the author of this thesis, who might be expected to be enthusiastic about the new proposed approach. We argue that while this is certainly true, the author was also equally enthusiastic about the past teaching comprising the years from 2006 to 2009, when the approach described in this thesis had not yet been developed, or even conceived. Although the re-marking was not blind, it was double checked. We therefore argue that the scorings are consistent.

External validity means an experiment can be generalised to different subjects, settings, and experimenters (Bracht & Glass, 1968). Threats to external validity are mainly classified as those to both population validity and environmental/ecological validity.

One limitation of our evaluation is that the number of students was low in each year. However, using appropriate statistical methods, there were still statistically significant results. We take advantage of using existing historic data as samples taught by conventional methods. We argue that using a set of extra hypotheses (Hypotheses 1, 2, and 3) can rule out effects from confounding variables.

There are possible threats to population validity in our use of quasi-related samples in 2011 and 2013 by collecting data from both mid-term and final examinations. However, instead of comparing both samples in one hypothesis, we used them separately in different hypotheses. The samples from the mid-term examination are used for ruling out the effect from confounding variables, particularly in Hypothesis 2. The samples from the final examination are used for comparing the performance from both new and old methods in Hypothesis 4. We argue that the impact of a carry-over effect in our final examination results were reduced by using explicit instructions to keep the track of programming process under our experimental method.

In all the years, the author of this thesis who designed the new method taught the course interactively in a small class (held in a lab class) with a mixture of theory and practice. We argue that if there is any bias that may have affected the new method, this bias would have also affected the conventional method in 2006-09 as we discuss in internal validity.
Moreover, any form of Hawthorne effect\textsuperscript{44} would be expected in the evaluations in 2011 and 2013 because the students were aware that they were being taught using a modified experimental method (since they had to sign ethics approval forms). Since students did not know which part of the course was traditional and which was novel, the effect would apply to the whole course, including performance in the mid-term test, which was not observed.

Additionally, our evaluation only considered the writing of a single program. One area for future work is to consider more programs. Furthermore, our evaluation was based on the quantitative results of students’ performance. On the other hand, qualitative analysis has been widely used to identify students’ perspective and factors of their success in learning. For example, Teague et al. (2012) and Teague and Lister (2014) had not only used quantitative analysis for their in-class test results, but also applied qualitative analysis of the recorded “think aloud” verbal reports from students, in order to extract information about their reasoning when solving the problems of in-class tests. Based on the mixed quantitative and qualitative analyses, they concluded that it gave them a better understanding of why their students were struggling. Therefore, in the future work, we could use mixed method having both quantitative and qualitative analyses.

Finally, although our results have shown a large effect, our conclusion is still based on constraints of small samples. Slavin and Smith (2009) argued that small samples are generally associated with larger effect sizes, but their effect sizes are more variable. They suggested that although small studies may be valuable at the early stage, for the purpose of generalisability, large studies should produce more reliable and replicable estimates to the program effects in education. Therefore, we were cautious with our experimental designs in order to minimise small sample bias. We developed an appropriate design to rule out influences by any plausible factors that threatened the validity of our experiments. We also conducted extensive statistical measurements of not only $p$-value, but also required sample size, effect size, statistical power, and chose appropriate statistical test methods, particularly for small sample groups.

\textsuperscript{44} The Hawthorne effect is a phenomenon that individuals improve their behaviour or performance when they aware of being observed.
8.4 Summary

We have evaluated our new approach by collecting final exam results from small samples in 2011 and 2013. We also collected small samples in several years when teaching through conventional method consisting of final exam results from 2006 to 2009 and mid-term exam results in 2011 and 2013. In all these years, both methods had been taught by the author of the thesis on the same campus of an institute. All the programming questions were similar from year to year. We re-marked one programming question in these exams under the same criteria.

We considered relevant statistical issues against our small sample sizes. Before testing our sample data, we performed an a priori statistical power analysis to predict our required sample size based on Cohen’s convention. In order to ensure providing a significant $p$-value that has a sufficient effect size for an acceptable statistic power when using the one-way ANOVA test, we combined our collected sample data into three groups. We then detected the effect size from the combined sample data and computed the achieved power. Since the detection result of effect size was large from our small sample size, we have enough reason for making the large effect size assumption of our experimental design.

We intended to use a one-way ANOVA test and independent t-tests to compare student results across groups. Before using them, we needed to ensure that our data met the assumption of normal distribution. Subsequently, we visually detected the breach of a normality assumption by outliers in box plots, histograms, and P-P and Q-Q plots. Although the visual results indicated non-normal distribution, we may have been able to use both the ANOVA and t-tests because they are quite robust against violations of the normality assumption. However, when the distribution of small size samples of data is platykurtic, it has an adverse effect on Type I errors and statistical power. Therefore, we took further measurement of skewness, kurtosis and a normality test before selecting appropriate inferential statistical tests. We concluded that our data were non-normal and also from a platykurtic distribution. Consequently, we decided to use a non-parametric method in our null hypothesis significance testing. When using SPSS statistical package for the nonparametric tests, we chose the exact test option or Monte Carlo option for small samples with many ties instead of the default option using asymptotic testing results for large samples.
In the comparison, there are other differences such as different student cohorts, exam questions, computer languages, and marking standards. Therefore, we defined a number of hypotheses that corresponded to different plausible explanations for variations in student performance due to these confounding factors. Before proceeding to test the difference between control and experimental groups (H4), we tested three Hypotheses (H1, H2, and H3) to rule out influences by any plausible factors when conducting evaluations.

1. The test results from Hypothesis 1 indicate that although there were differences of student cohorts, variations of examination questions and relevant marking schedules in the past (2006-2009), under the conventional method there were not significant differences of examination scores during these years;
2. The test results from Hypothesis 2 also indicate that although there are differences of student cohorts, variations of examination questions, relevant marking schedules, and changes in computer languages in the recent years (the mid-term of 2011 and 2013), under the conventional method there were not significant differences of examination scores during all these years (2006-2009 as well as the mid-term of 2011 and 2013);
3. The test results from Hypothesis 3 suggest that although there were differences of student cohorts, variations of examination questions, and relevant marking schedules in the final examinations of 2011 and 2013, under the experimental method there were not significant differences of examination scores during these two years; and
4. The test results from Hypothesis 4 including post hoc tests indicate that the experimental method in both years 2011 and year 2013 compared to the conventional method from years 2006 to 2009 did yield large effect sizes to significantly improve examination scores at .05 levels. By using Holm-Bonferroni corrections with Mann-Whitney U test pairwise comparisons, the post hoc tests also verified the improvement when teaching using the experimental method.

Therefore, we conclude that the new proposed method makes a significant difference to the improvement of final examination results comparing to those when using the conventional method, although there are differences in student cohorts, examination questions, relevant marking schedules, and computer languages. Finally, the threats to validity of our conclusion were discussed in terms of internal validity and external validity.
Chapter 9 Conclusion and Contribution

9.1 How the Objectives Were Met

This study has explored the issue that there continues to be a high rate of failing or withdrawing from the first programming course. Novices can understand individual statements of a computer programming language, but they struggle in combining the statements into a program for solving a simple problem. In the literature, a number of reasons have been proposed for this in terms of “fragile” knowledge of programming concepts, lack of problem-solving strategies and plans, and lack of detailed mental models. Although a wide range of approaches have been proposed to improve novices’ learning of programming, the above problem of how to combine diverse programming statements into a valid program is still unsolved in the teaching of novices programming. Therefore, our objectives in this research were how to teach novices to construct code to solve a problem.

The achievements of our objectives consist of: 1) developing programming strategies using goals and plans, 2) developing a framework to support plans in a visual programming environment, and 3) having applicable approaches of teaching programming. To apply the programming strategy of using goals and plans for novices, we have developed a visual notation to represent the goal-plan model in a data-flow paradigm. To provide a VPE for using goals and plans, we have developed a visual data-flow framework to implement the design at plan level and to provide immediate feedback. To support teaching of programming, we developed a detailed programming process to use goals and plans in the VPE. Finally, an experimental evaluation provided empirical evidence for the usability and effectiveness of our approach.

**Research Question 1:** How can goal and plan representations be integrated into a VPL?

We considered using experts’ knowledge and strategies in the form of goals and plans to solve programming problems, particularly using the data-flow relations proposed by Pennington (1987). We also considered using visual notation to represent goals, plans and dataflow as a “hand solution”, i.e. programming analysis and design for novices. We concluded that a data-flow representation is better for high-level design, whereas control flow deals well with code details at a low level. We therefore developed a data-flow framework in a VPE for mapping the
design in terms of plans in a data-flow diagram, using a provided plan library, and having program code details in the control flow paradigm. Significantly, the unmerged plans in the data-flow paradigm can be executable, which makes it possible to provide feedback to students at the design level.

**Research Question 2:** How can we teach novice programmers to use goals and plans and to merge plans?

One reason why it is hard for novices to learn programming is that their working memory can be easily overloaded. According to cognitive load theory, using process-oriented worked examples can provide a scaffold for novices to learn programming. Therefore, in our pedagogy, we provide students with a step-by-step programming process and also provide them with means to get feedback at the intermediate process levels.

A solution often needs multiple plans and these plans need to be merged into one program. Based on the existing strategies (Soloway, 1986) for combining plans, our research explored how to effectively merge our plans from the data-flow design into control flow code in the VPL environment. Therefore, both a step-by-step process and a set of principles are applied to provide guidance to support merging plans.

**Research Question 3:** How can the teaching process for programming be improved and evaluated?

Based on the previous two research questions, we integrated goals and plans into a VPE by developing visual notations and a data-flow framework. We also developed a detailed process for using goals and plans and merging plans. In order to construct a new teaching process for programming, we chose five pedagogical methods and assessed them against cognitive load theory. The five pedagogical methods applied in our teaching process were: 1) worked examples and fading worked examples, 2) progressive methods, 3) self-explanation by using sub-goal labelling, visual notation, and desk-check, 4) process-oriented worked examples and scaffold fading, and 5) feedback from each phase of programming process.

We chose to use a real teaching setting, rather than a short workshop to evaluate the effectiveness of this new proposed approach. The interactive teaching was in a small class with
a mixture of theory and practice in a computer lab. We evaluated our approach by comparing traditionally taught introductory programming (2006-2009) with modified teaching in 2011 and 2013, which retained a conventional approach for the first half of the course, but after the mid-term examination followed the new approach for the second half of the course.

We used the desired significant level, the expected effect size, the desired statistical power on main hypothesis to determine the needed sample size; chose appropriate statistical tests; tested the null hypothesis significance and estimated the effect size. We also defined a number of hypotheses that correspond to different plausible explanations for variations in student performances due to confounding factors such as differences in student cohorts, exam questions, computer languages, and marking standards from different years. Our evaluation results suggest that our new approach in both 2011 and 2013 compared to the conventional approach from year 2006 to year 2009 did significantly improve the student results.

9.2 Contribution

Teaching novices programming is hard. Although novices can understand the syntax and semantics of a computer programming language, they have difficulties in using the programming language to write a valid program. The work of this thesis — teaching novices programming through a programming process using goals and plans with a visual programming environment — contributes to the field of computer education research in a number of ways:

1. The development of visual representation of goals and plans that supports early feedback from the program design using the unmerged plans;

Although using goals and plans in teaching novices programming has been proposed over many years, it is still not widely applied in teaching practice. There are even fewer studies of visual representation of goals and plans in the data-flow paradigm. This study proposes explicitly using visual notation of goals and plans (representing strategies that experts have for novices to solve problems) in the data-flow paradigm. The development of visual notation provides the mental representation of plan knowledge that Letovsky and Soloway (1986) and Pennington (1987) proposed.
Up until now, there was no implementation of plans in a data-flow paradigm. Our data-flow framework developed in a VPE in this study is not only used to reduce syntax errors and motivate novices’ learning, but is also used to provide a plan library and to support the design of a visual plan network allowing the problem solution to be executable without merging plans at the design level. The development of our data-flow framework provides a tool to support novices in understanding experts’ mental representation as Wiedenbeck et al. (1993) proposed.

Our data-flow framework also provides immediate feedback, not only for the unmerged plans, but also from all the intermediate level phases in the programming process up to the final program code. Subsequently, the feedback from the intermediate levels of the process motivates and engages the novices in the process. It also extends the development of plan-based tools (Bonar & Cunningham, 1988; Bonar & Lifick, 1990; Guzdial et al. 1998) from the conventional control flow paradigm to the data-flow paradigm.

2. The development of a detailed goal and plan based programming process;

A key feature of this study is that within a visual programming environment it provides a detailed programming process that guides novices to develop a program. The process starts from analysis by using the visual notation of goals and plans in the dataflow paradigm and progresses over several steps to the final program code in the control flow paradigm.

From the literature of computer science education, this study applies the theory of using chunks of knowledge (e.g. goals and plans, templates, schemas, and patterns) that experts have to novices’ programming (Soloaway, 1986; Linn, 1986; Détienne, 1990; Wallingford, 1996; Porter & Calder, 2003), particularly in the use of goals and plans. It provides supports by using a programming process and a VPE to minimise the concerns of Gilmore (1990) about how to use the knowledge of plans and of Rist (1989) and Wallingford (1996) about how to map the plans into program code. When using a VPE to teach novices programming, it is promising to consider deeper intellectual merit in terms of abstraction of solutions (Gibbs & Coady, 2010) and to have a pedagogical rethink (Lister, 2011a). From the point of this view, the use of a programming process for goals and plans to teach novices programming in a VPE has filled the gaps for the above pedagogical consideration rather than using an unguided “tinkering” approach (Perkins et al. 1986).
3. The development of CLT-based teaching methods;

This study proposed and evaluated a programming process as scaffolding to support novices’ learning and reduce their cognitive load. The development of a programming process of using goals and plans in the VPE provides this support as scaffolding for novices’ learning to minimise the concerns arising when novices are taught through problem solving with minimum guidance (Guzdial, 2009) and from overloading novices’ working memory without helping them form “chunks” of knowledge (Lister, 2011b).

4. Empirical Evaluation

Our evaluation of using goals and plans in real teaching practice extends de Raadt’s (2008) study by adding visual notation, a framework of plans in the data-flow paradigm, and a programming process in a VPE. The results indicated that the integration of visual programming language, goals and plans, and programming process can be explicitly fitted into the curriculum of novice programming.

Considering the poor statistical power levels reported in IS research (Dybå et al, 2006), a comprehensive statistical analysis was applied in the evaluation of our method in terms of sample size, statistical power, significance level, and effect size. The results indicate that tests could be applied to different small samples using well-defined hypotheses to rule out influences from any plausible factors. The experimental research suggested that the new teaching approach in this study has the potential to significantly improve novices’ programming learning.

9.3 Limitations and Future Work

A limitation of this study is that the model of starting from data-flow in a VPL is not expected to be applicable to all programming tasks. The study only explored and tested a particular type of programs: those that process input (batch input). Other types of programs, for example, object-oriented programs, reactive programs, concurrent programs, and distributed software are not covered. Additionally, there may be limits on the complexity of the programs that can be tackled (but arguably for a first programming course this isn’t a problem since we would not want to have very complicated programs).
Our strategies of using goals and plans, a programming process, and a VPE, specifically the visual notation of goals and plans, could be applied to other programming languages. However, the implementation of a new plan library and programming process depends on the programming paradigm, e.g. procedural programming. Some features of the programming language also need to be considered, e.g. whether it is a visual programming language.

It is true that our approach starts from goals and plans based on the top-down and stepwise refinement methods, which had been criticised as not being suitable for novices (Caspersen, 2007; Caspersen & Kölling, 2009). Caspersen and Kölling (2009) and Caspersen (2007) argued that stepwise refinement represented a strict top-down programming process from abstract non-executable program to concrete program language code. Instead, they advocated a conceptual framework of stepwise improvement including three aspects: extension (extending a working, but incomplete program, to handle additional requirements), refinement (refining from abstract to concrete code), and restructuring (improving non-functional aspects, such as design or portability). The conceptual framework emphasised incremental development, with testing of every increment. Moreover, they criticised the traditional programming approach of students solving problem on their own as being similar to a random walk rather than a guided tour, which produced high cognitive load. Instead of a top-down process from goals to programs, they proposed a process inspired by agile methods, where functionality is added to a working program. They then provided a guided programming process named STREAM (stubs, tests, representation, evaluation, attributes, and methods).

We agree with their arguments against top-down refinement. However, for the very first introductory programming course, the programs being developed are sufficiently simple to allow top-down refinement to be used. We argue that for these courses, what we need is a basic (non-iterative) process that supports students to reach the point of being able to write very simple working programs. We also agree with their second point, and consequently we also provide concrete and detailed guidance to novices. We argue that our approach not only include stepwise refinement together with a programming process, but also provides strategies to support novices learning programming. Firstly, our visual notation of goals and plans has simplified the complexity of design. Our executable dataflow framework provides immediate feedback for the purpose of stepwise refinement. Secondly, the progress of our programming process is from simple notation and existing plans in the dataflow paradigm, to
complex plan details in the control flow paradigm with feedback from every phase. Moreover, we had developed a tool to support our programming process for the implementation of stepwise improvement. Finally, we have used a visual programming environment for our framework to support novices’ learning.

Additionally, our approach relies on a plan library (provided in our dataflow framework). This means that novices are not developing programs from scratch. However, the aim is to help novices to learn programming skills in a progressive fashion from simple to complex. The plan library and the programming process are a scaffold for novices to learn programming. The programming process is a trade off in that the process supports effective learning from the beginning, but it is also more complex than unguided programming. Our experiences clearly show that the benefits from having a structured process outweigh the costs of the additional complexity. We are also aware that the scaffolding will fade after novices can self-explain the worked examples. We only evaluated students’ performance in the current course. The effect of our experimental method on students’ further study may need to be traced in other courses in future work.

Another limitation is that the number of students in the evaluation was low in each year. Also, as noted earlier, in the teaching environment (a New Zealand institute of technology) students are able to re-sit an exam, and we consequently only considered students’ results in their first trial. However, this means that our evaluation data may not reflect students’ best performance.

Our evaluation was based on the quantitative results of students’ performance on a programming question. In the future work, firstly, we could compare students’ performance on different problems using different approaches. Secondly, we could use a mixed method having both quantitative and qualitative analyses. In the qualitative study, we could collect data in terms of “think aloud” verbal reports when students are using goals and sub-goals to solve problems. We could further evaluate students’ design process using verbal protocol analysis.

One direction for further work is to provide many different plans to choose for the same goal in order to better understand how novices select plans. Another area for future work is to investigate how to better support Phase 4 (plan expanding) and Phase 5 (merging) in the
programming process, because the current process is somewhat tedious and complex. In Phase 4, in order to help explain this process, we have provided students with video clips of a screen capture that demonstrates how to firstly duplicate plan details from each plan block and then how to replace the parameters by copying-and-pasting a port name from the plan block. Phase 4 is purely mechanical and could be automated in future work. However, in Phase 5, the support of merging needs to let students gain insight into the merging process, rather than just providing a “wizard”.

To conclude, our results show that the combination of goals and plans, programming process, and visual programming environment is a promising first step on the path to successfully teach novices programming. It provides scaffolding for novices learning towards becoming experts.
References


ACM, & IEEE. (2013). Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science: Association for Computing Machinery (ACM).


Dehnadi, S., & Bornat, R. (2006). The camel has two humps (working title). *Middlesex University, UK.*


Appendix A—Examination Questions and Marking Schedule

A-1: Programming Question in the Examinations

Year 2006 Final Examination
You are asked to write a program to input a series of numbers. The program displays the double value of each positive number you input. It displays “Enter a positive number.” if you enter a negative number. The program stops when you enter a zero (0). At the end, the program displays how many positive numbers you have entered and what the total value of these positive numbers is.

Year 2007 Final Examination
You are asked to write a program to let a user input a series of numbers. At the end, the program displays both count positive numbers and negative numbers that were originally entered by the user. When the user enters a zero (0), the program stops. For example, after the user enter numbers, 5, -2, 3, 4, -1, 0, the program displays that you have entered 3 positive numbers and 2 negative numbers.

Year 2008 and 2009 Final Examination
You are asked to write a program to let a user input a series of numbers. At the end, the program displays the count and the sum of these positive numbers that were originally entered by the user. However, when the user enters a negative number, this number will be displayed as being doubled and the program will ask the user to enter again for a positive number. When the user enters a zero (0), the program stops. For example, after you entered numbers, 5, -2, 3, and 0, the program displays that the sum of positive numbers you have entered is 8. When you enter -2, it also shows -4 and “Enter a positive number”.

Year 2011 Mid-term Examination
Write a program that will read in a sequence of integers and output the smallest positive integer. Stop reading when the value -1 is input. For example, when the data are 5, -2, 3, 7, 0, 2, -4, 6, -1, the smallest positive integer is 2.
Year 2011 Final Examination
Write a program that will read in integers and output their average. Stop reading when the value -1 is input. For example, after entering data 1, 2, 3, and -1, the average is 2.
1) Identify and analysis goals and plans by which you are going to solve this problem.
2) Design your plans for the solution. If necessary, build up your own plans in BYOB.
3) Merge and implement your plans to complete the BYOB program.
4) Test your program at every stage.

Year 2013 Mid-term Examination
Write a program to calculate and output the wage for each member of a group of people and the total wages of the group of people.

Input the pay rates and working hours for each person. The formula for wages is wages = rates * hours. It stops when the rates or hours is -1.
For example, when John’s pay rate is 20; working hours is 10, his wage is $200. When Mary’s pay rate is 15; working hours is 20, her wage is $300. Therefore, the total wages is $500. Remember, your program should allow user to input details for a group of people to calculate wages.

Year 2013 Final Examination
Write a program that will read in integers and output their average. Stop reading when the value -1 is input. For example, after entering data 1, 2, 3, and -1, the average is 2.

1) Predict the results from the following test cases and complete the following table.
Predicting answers: (Fill in the table by your predicated answers)

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Testing Data</th>
<th>Predicted Answers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>1, 2, 3, -1</td>
<td>2 (= (1 + 2 + 3)/(1 + 1 + 1) = 6/3)</td>
</tr>
<tr>
<td>Test Case (1)</td>
<td>2, 3, 4, -1</td>
<td></td>
</tr>
<tr>
<td>Test Case (2)</td>
<td>2, 3, 7, 8, -1</td>
<td></td>
</tr>
</tbody>
</table>
Analysis

2) Based on the process of your prediction, draw a goal diagrams to analysis goals that you are going to achieve in the way of input-process-output.

3) Map your goal diagram into a plan network by the symbol of plan. Write a unique name for every port in each plan.

4) Deck-check your plan network by completing the following table.

<table>
<thead>
<tr>
<th>Plan Name</th>
<th>Port Name</th>
<th>Dataflow</th>
<th>Dataflow</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5) Fill your predicted answers and desk-checking results in both columns of Predicted Answers and Phase 1 in the following test schedule table.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case (1)</th>
<th>Test Case (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data:</td>
<td>2, 3, 4, -1</td>
<td>2, 3, 7, 8, -1</td>
</tr>
<tr>
<td>Predicted Answers:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Note: You will use this table again for Question 7, 9, 11, and 13)
**Design by plan blocks**

6) According to your plan network in question 3), implement the diagram by plan blocks from the plan library. Save your BYOB program and name it as Plan.

7) Use test schedule table in Question 5) to test and debug your Plan program. Ensure there are no bugs in the plan program 3. Fill your testing results in the column Phase 2.

**Expand plan blocks**

8) Save your Plan program first for the final submission and then choose “Save As” option to save your plan program into a new name, Expand. Then expand your new named program from the plan blocks.

9) Use test schedule table in Question 5) to test and debug your Expand program. Ensure there are no bugs in the expanded program 3. Fill your testing results in the column Phase 3.

**Merge expanded details**

10) Save your Expand program first for the final submission and then choose “Save As” option to save your expanded program into a new name, Merge. Then merge your new named program from the expanded plan details.

11) Use test schedule table in Question 5) to test and debug your Merge program. Ensure there are no bugs in the merged program 3. Fill your testing results in the column Phase 4.

**Simplifying merged details**

12) Save your Merge program first for the final submission and then choose “Save As” option to save your merged program into a new name, Final. Then simplify your new named program from the merged plan details.

13) Use test schedule table in Question 5) to test and debug your Final program. Ensure there are no bugs in the merged program 3. Fill your testing results in the column Phase 5.
A-2: Marking Schedule for Programming Question in the Examinations

Table A-2-1 Year 2006 Final Examination Marking Schedule

<table>
<thead>
<tr>
<th>Task Completed</th>
<th>Evidence Statement</th>
<th>Possible Marks</th>
<th>Actual Marks</th>
</tr>
</thead>
</table>
| 1. Identify all the variables correctly | Four variables are identified and used correctly (may including initial value):  
  • Number  
  • Count  
  • Sum (or Total) | 10 | |
| 2. Correctly use fragments of key code | Fragments are used correctly:  
  1) Input a serials of numbers  
  2) Identify both positive and negative numbers  
  3) Compute and output double positive numbers  
  4) Compute Count  
  5) Compute Sum  
  6) Output message for negative numbers  
  7) Output both Count and Sum | 40 | |
| 3. Combine code of fragments correctly | All the fragment must be combined in the correct order  
  • Fragments 3), 4), and 5) can be combined in any sequential order under the fragment 2) of selection of positive numbers  
  • Fragment 6) must be under the fragment 2) of selection of negative numbers  
  • The fragment 2) of selection of both positive and negative numbers is combined inside the fragment 1) Input numbers  
  • The output fragment 7) must be put in the last | 30 | |
<p>| 4. The final program is testable and bug free | The final program must be executable and bugs free | 20 | |
| | | <strong>Total Marks:</strong> | 100 |</p>
<table>
<thead>
<tr>
<th>Task Completed</th>
<th>Evidence Statement</th>
<th>Possible Marks</th>
<th>Actual Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify all the variables correctly</td>
<td>Four variables are identified and used correctly (may including initial value):</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>• Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Positive Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Negative Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Correctly use fragments of key code</td>
<td>Five fragments are used correctly:</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>1) Input a serials of numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2) Identify both positive and negative numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3) Compute Positive Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4) Compute Negative Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5) Output both counts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Combine code of fragments correctly</td>
<td>All the fragment must be combined in the correct order</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>• Both fragment 3) of compute Positive Count and fragment 4) of compute Negative Count are combined inside the fragment 2) of selection of different numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The fragment 2) of selection of different numbers is also combined inside the fragment 1) of Input numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The output fragment 5) must be put in the last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. The final program is testable and bug free</td>
<td>The final program must be executable and bugs free</td>
<td></td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Total Marks:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task Completed</td>
<td>Evidence Statement</td>
<td>Possible Marks</td>
<td>Actual Marks</td>
</tr>
<tr>
<td>----------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>1. Identify all the variables correctly</td>
<td>Four variables are identified and used correctly:</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>- Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Sum (or Total)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Correctly use fragments of key code</td>
<td>Five fragments are used correctly:</td>
<td></td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>1) Input a serial of numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2) Identify both positive and negative numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3) Compute Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4) Compute Sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5) Compute and output double negative numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6) Output message for negative numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7) Output both Count and Sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Combine code of fragments correctly</td>
<td>All the fragment must be combined in the correct order</td>
<td></td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>- Fragments 3) and 4) can be combined in any sequential order under the fragment 2) of selection of positive numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Fragment 5) and 6) must be under the fragment 2) of selection of negative numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- The fragment 2) of selection of both positive and negative numbers is combined inside the fragment 1) Input numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- The output fragment 7) must be put in the last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. The final program is testable and bug free</td>
<td>The final program must be executable and bugs free</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

Total Marks: 100
<table>
<thead>
<tr>
<th>Task Completed</th>
<th>Evidence Statement</th>
<th>Possible Marks</th>
<th>Actual Marks</th>
</tr>
</thead>
</table>
| 1. Identify all the variables correctly | Four variables are identified and used correctly (may including initial value):  
   - Number  
   - Min | 10                          |               |
| 2. Correctly use fragments of key code       | Five fragments are used correctly:  
   1) Input a serials of numbers  
   2) Identify both positive and negative numbers  
   3) Compute Min  
   4) Output Min | 40                          |               |
| 3. Combine code of fragments correctly       | All the fragment must be combined in the correct order  
   - Fragment 3) must be under the fragment 2) of selection of positive numbers  
   - The fragment 2) of selection of both positive and negative numbers is combined inside the fragment 1) Input numbers  
   - The output fragment 4) must be put in the last | 30                          |               |
<p>| 4. The final program is testable and bug free | The final program must be executable and bugs free | 20                          |               |
|                                              | <strong>Total Marks:</strong> 100                                                              |                |              |</p>
<table>
<thead>
<tr>
<th>Task Completed</th>
<th>Evidence Statement</th>
<th>Possible Marks</th>
<th>Actual Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify all the variables correctly</td>
<td>Four variables are identified and used correctly (may including initial value):</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Hours</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Rate</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Wages</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Sum (Sum = 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Correctly use fragments of key code</td>
<td>Six fragments are used correctly:</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1) Input a serials of Hours (two Inputs, one inside and one outside loop)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2) Input a serials of Rates(two Inputs, one inside and one outside loop)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3) Compute Wages</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4) Compute Sum = Sum + Wages</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5) Display Wages</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6) Display Sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Combine code of fragments correctly</td>
<td>All the fragment must be combined in the correct order:</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Two Inputs 1) and 2) must be nested</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The fragment 3) must be processed inside loop and before next Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The fragment 4) must be processed after 3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The output fragment 5) must be put inside the loop.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The output fragment6) must be put outside the loop.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. The final program is testable and bug free</td>
<td>The final program must be executable and bugs free</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Marks:</td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Task Completed</td>
<td>Evidence Statement</td>
<td>Possible Marks</td>
<td>Actual Marks</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------</td>
<td>----------------</td>
<td>--------------</td>
</tr>
<tr>
<td>1. Identify all the variables correctly</td>
<td>Four variables are identified and used correctly (may including initial value):</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Average</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Correctly use fragments of key code</td>
<td>Five fragments are used correctly:</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1) Input a serials of numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2) Compute Count</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3) Compute Sum</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4) Divide</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>5) Output Average</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Combine code of fragments correctly</td>
<td>All the fragment must be combined in the correct order:</td>
<td>30</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Both fragment 2) of compute Count and fragment 3) of compute Sum are combined inside the fragment 1) of Input numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Fragment 4) of Divide must be placed after the fragment 1) of Input numbers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- The output fragment 5) must be put in the last</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. The final program is testable and bug-free</td>
<td>The final program must be executable and bugs free</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Total Marks:</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B—Evaluation Data Analysis

B-1: Statistic Analysis by G*Power

1. Predicting Sample Size

According to conventions by Cohen (1992), we set up effect size = .10, .25, and .40, \( \alpha = .05 \), Power \( (1 - \beta) = .80 \), Number of groups = 6 and 3 for the “Input Parameters” in G*Power. The results are as follows:

<table>
<thead>
<tr>
<th>Effect size</th>
<th>Minimum Sample Size (6 groups)</th>
<th>Minimum Sample Size (3 groups)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small (0.10)</td>
<td>1290</td>
<td>969</td>
</tr>
<tr>
<td>Medium (0.25)</td>
<td>216</td>
<td>159</td>
</tr>
<tr>
<td>Large (0.40)</td>
<td>90</td>
<td>66</td>
</tr>
</tbody>
</table>

1) Predicting sample size from original six groups

The smallest sample number is 90. However, we have actually collected 72.

Figure B-1-1 Example of predicting required sample size from six groups

45 http://www.gpower.hhu.de/en.html
2) Predicting required sample size from combined three groups

After combing into three groups, we change the “Input Parameters” to: effect size = .40, \( \alpha = .05 \), Power \((1 - \beta) = .80\), Number of groups = 3. G*Power displays the predicted total sample size of participants as 66 and the actual power as .82 in the “Output parameters”.

![Figure B-1-2 Example of predicting required sample size from combined three groups](image)

3) Foreseeing power from current total sample size

Having total of 72 students, we attempt to change “Input Parameters” to: effect size = .40, \( \alpha = .05 \), Power \((1 - \beta) = .85\), Number of groups = 3. G*Power displays the predicted total sample size of participants as 72 and the actual power as .85 in the “Output parameters”.

![Figure B-1-3 Example of foreseeing power from current total sample size](image)

The final results show, after combining our data into three groups, the total sample sizes (72) can meet the requirements of one-way ANOVA analysis for the statistical power analysis.
2. Detect effect size and compute the achieved power

1) Detect effect size from combined sample data

<table>
<thead>
<tr>
<th></th>
<th>G3</th>
<th>G2</th>
<th>G1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>88.7272</td>
<td>37.7619</td>
<td>44.6206</td>
</tr>
<tr>
<td>SE</td>
<td>4.2032</td>
<td>9.9658</td>
<td>7.8617</td>
</tr>
<tr>
<td>Median</td>
<td>100</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>Mode</td>
<td>100</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SD</td>
<td>19.7150</td>
<td>45.6693</td>
<td>42.3365</td>
</tr>
<tr>
<td>SV</td>
<td>388.684</td>
<td>2085.69</td>
<td>1792.38</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>6.4278</td>
<td>-1.9025</td>
<td>-1.9166</td>
</tr>
<tr>
<td>Skewness</td>
<td>-2.4563</td>
<td>0.4433</td>
<td>0.1600</td>
</tr>
<tr>
<td>Range</td>
<td>79</td>
<td>100</td>
<td>96</td>
</tr>
<tr>
<td>Minimum</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>100</td>
<td>100</td>
<td>96</td>
</tr>
<tr>
<td>Sum</td>
<td>1952</td>
<td>793</td>
<td>1294</td>
</tr>
<tr>
<td>n</td>
<td>22</td>
<td>21</td>
<td>29</td>
</tr>
<tr>
<td>Conf Level(95.0%)</td>
<td>8.741172</td>
<td>20.78844</td>
<td>16.10398</td>
</tr>
<tr>
<td>SD*SD</td>
<td>388.684</td>
<td>2085.69</td>
<td>1792.38</td>
</tr>
<tr>
<td>n-1</td>
<td>21</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>(n-1)<em>SD</em>SD</td>
<td>8162.364</td>
<td>41713.81</td>
<td>50186.83</td>
</tr>
</tbody>
</table>

WSS (within sum of squares) 100063
N - k (72 - 3) 69
Within Mean Squares 1450.188
SD σ within each group 38.08134

Figure B-1-4 Example of detecting effect size from samples

From the result (0.57 > 0.40), we concluded it is large effect.
2) Compute the possibly archived power

![Diagram showing statistical analysis parameters]

Figure B-1-5 Example of computing the achieved power

Therefore, we concluded that if there is a significant difference (5%) from our collected sample size, the results will be large effect with enough power.
### 1. Descriptives for individual sample groups

<table>
<thead>
<tr>
<th>Year of the Test</th>
<th>Statistic</th>
<th>Std. Error</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Year 2006 &amp; 2007</strong></td>
<td>Mean</td>
<td>44.62</td>
</tr>
<tr>
<td></td>
<td>95% Confidence Interval for Mean</td>
<td>Lower Bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper Bound</td>
</tr>
<tr>
<td></td>
<td>5% Trimmed Mean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Std. Deviation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interquartile Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Skewness</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kurtosis</td>
<td></td>
</tr>
<tr>
<td><strong>Year 2008 &amp; 2009</strong></td>
<td>Mean</td>
<td>37.76</td>
</tr>
<tr>
<td></td>
<td>95% Confidence Interval for Mean</td>
<td>Lower Bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper Bound</td>
</tr>
<tr>
<td></td>
<td>5% Trimmed Mean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Std. Deviation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interquartile Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Skewness</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kurtosis</td>
<td></td>
</tr>
<tr>
<td><strong>Year 2011 &amp; 2013</strong></td>
<td>Mean</td>
<td>88.73</td>
</tr>
<tr>
<td></td>
<td>95% Confidence Interval for Mean</td>
<td>Lower Bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper Bound</td>
</tr>
<tr>
<td></td>
<td>5% Trimmed Mean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Variance</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Std. Deviation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Minimum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Maximum</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interquartile Range</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Skewness</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Kurtosis</td>
<td></td>
</tr>
<tr>
<td><strong>Mid-term Year 2011 &amp; 2013</strong></td>
<td>Mean</td>
<td>38.24</td>
</tr>
<tr>
<td></td>
<td>95% Confidence Interval for Mean</td>
<td>Lower Bound</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Upper Bound</td>
</tr>
<tr>
<td></td>
<td>5% Trimmed Mean</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Median</td>
<td></td>
</tr>
</tbody>
</table>
2. Descriptives of combination of three groups (Group 1, 2 & 3)

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Stat. Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>56.10</td>
</tr>
<tr>
<td>95% Confidence Interval for Mean</td>
<td>5.127</td>
</tr>
<tr>
<td>Lower Bound</td>
<td>45.87</td>
</tr>
<tr>
<td>Upper Bound</td>
<td>66.32</td>
</tr>
<tr>
<td>5% Trimmed Mean</td>
<td>56.77</td>
</tr>
<tr>
<td>Median</td>
<td>85.00</td>
</tr>
<tr>
<td>Variance</td>
<td>1892.483</td>
</tr>
<tr>
<td>Std. Deviation</td>
<td>43.503</td>
</tr>
<tr>
<td>Minimum</td>
<td>0</td>
</tr>
<tr>
<td>Maximum</td>
<td>100</td>
</tr>
<tr>
<td>Range</td>
<td>100</td>
</tr>
<tr>
<td>Interquartile Range</td>
<td>95</td>
</tr>
<tr>
<td>Skewness</td>
<td>-.336</td>
</tr>
<tr>
<td>Kurtosis</td>
<td>-1.773</td>
</tr>
</tbody>
</table>

3. Measuring skewness and kurtosis for combination of three groups (Group 1, 2 & 3)

From the above table, skewness is -.336; standard error is .283. For assessing skewness:

\[-.336 + .283 = -.053; \quad -.336 - .283 = -.619\] (negative skewness or left skewed)

Therefore, the 95% confidence interval for the skewness score ranges from -.053 to -.619.

Again, from the above table, Kurtosis is -1.773; standard error is .559. For assessing kurtosis:

\[-1.773 + .559 = -1.214; \quad -1.773 - .559 = -2.332\] (platykurtic)

Therefore, the 95% confidence interval for the kurtosis score ranges from -1.214 to -2.332.
4. Tests of Normality for Combinations of Groups in Four Hypotheses

In general, Group 1, 2, and 3 are non-normal distribution because of both Kolmogorov-Smirnov and Shapiro-Wilk tests results < .001. However, Group 3’ is not non-normal distribution because of both Kolmogorov-Smirnov and Shapiro-Wilk tests results > .05. Therefore, we combine different combination of groups in each hypothesis to test normality of our data in four tests.

<table>
<thead>
<tr>
<th>Year of the Test</th>
<th>Kolmogorov-Smirnov Statistic</th>
<th>Kolmogorov-Smirnov df</th>
<th>Kolmogorov-Smirnov Sig.</th>
<th>Shapiro-Wilk Statistic</th>
<th>Shapiro-Wilk df</th>
<th>Shapiro-Wilk Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group 1 (Year 2006 &amp; 2007)</td>
<td>.212</td>
<td>29</td>
<td>.002</td>
<td>.773</td>
<td>29</td>
<td>.000</td>
</tr>
<tr>
<td>Group 2 (Year 2008 &amp; 2009)</td>
<td>.348</td>
<td>21</td>
<td>.000</td>
<td>.695</td>
<td>21</td>
<td>.000</td>
</tr>
<tr>
<td>Group 3 (Year 2011 &amp; 2013)</td>
<td>.284</td>
<td>22</td>
<td>.000</td>
<td>.642</td>
<td>22</td>
<td>.000</td>
</tr>
<tr>
<td>Group 3’ (Mid-term Year 2011 &amp; 2013)</td>
<td>.178</td>
<td>21</td>
<td>.080</td>
<td>.922</td>
<td>21</td>
<td>.097</td>
</tr>
</tbody>
</table>

Figure B-2-3 Test result of normality of Individual group

1) For history data (Group 1 and Group 2, from year 2006 to 2009) in Hypothesis 1, it is non-normal distribution because of both Kolmogorov-Smirnov and Shapiro-Wilk tests results < .001. Therefore, we need to use non-parametric M-W U test for Hypothesis 1.

<table>
<thead>
<tr>
<th>Kolmogorov-Smirnov Statistic</th>
<th>Kolmogorov-Smirnov df</th>
<th>Kolmogorov-Smirnov Sig.</th>
<th>Shapiro-Wilk Statistic</th>
<th>Shapiro-Wilk df</th>
<th>Shapiro-Wilk Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>.254</td>
<td>50</td>
<td>.000</td>
<td>.745</td>
<td>50</td>
</tr>
</tbody>
</table>

Figure B-2-4 Tests of Normality for Groups in Hypothesis 1

2) For mid-term results (Group 3’) and history data (Group 1 and Group 2) in Hypothesis 2, it is non-normal distribution because of both Kolmogorov-Smirnov and Shapiro-Wilk tests results < .001. Therefore, we need to use non-parametric K-W H test for Hypothesis 2.

<table>
<thead>
<tr>
<th>Kolmogorov-Smirnov Statistic</th>
<th>Kolmogorov-Smirnov df</th>
<th>Kolmogorov-Smirnov Sig.</th>
<th>Shapiro-Wilk Statistic</th>
<th>Shapiro-Wilk df</th>
<th>Shapiro-Wilk Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test</td>
<td>.195</td>
<td>71</td>
<td>.000</td>
<td>.809</td>
<td>71</td>
</tr>
</tbody>
</table>

Figure B-2-5 Tests of Normality for Groups in Hypothesis 2
3) For experimental data (Group 3, i.e. 2011 & 2013) in Hypothesis 3, it is non-normal distribution because of both Kolmogorov-Smirnov and Shapiro-Wilk tests results < .001. Therefore, we need to use non-parametric M-W U test for Hypothesis 3.

<table>
<thead>
<tr>
<th>Kolmogorov-Smirnova</th>
<th>Shapiro-Wilk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistic</td>
<td>df</td>
</tr>
<tr>
<td>Test</td>
<td>.284</td>
</tr>
</tbody>
</table>

a. Lilliefors Significance Correction

Figure B-2-6 Tests of Normality for Groups in Hypothesis 3

4) For the final results from all the experimental (Group 3) and control data (Group 1 and Group 2) in Hypothesis 4, it is also non-normal distribution because of both Kolmogorov-Smirnov and Shapiro-Wilk tests results < .001. Therefore, we need to use non-parametric K-W H test for Hypothesis 4.

<table>
<thead>
<tr>
<th>Kolmogorov-Smirnova</th>
<th>Shapiro-Wilk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistic</td>
<td>df</td>
</tr>
<tr>
<td>Test</td>
<td>.261</td>
</tr>
</tbody>
</table>

a. Lilliefors Significance Correction

Figure B-2-7 Tests of Normality for Groups in Hypothesis 4
B-3: Results of Null Hypothesis Significance Testing from Non-parametric Method by Using SPSS v22

1. Results of *Hypothesis 1* from Mann-Whitney U test when choosing exact test option

<table>
<thead>
<tr>
<th>Ranks</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Combined Year</td>
<td>N</td>
<td>Mean Rank</td>
</tr>
<tr>
<td>Test</td>
<td>Group 1 (Years 2006 &amp; 2007)</td>
<td>29</td>
<td>26.81</td>
</tr>
<tr>
<td></td>
<td>Group 2 (Years 2008 &amp; 2009)</td>
<td>21</td>
<td>23.69</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Statistics(^a)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mann-Whitney U</td>
<td>266.500</td>
</tr>
<tr>
<td>Wilcoxon W</td>
<td>497.500</td>
</tr>
<tr>
<td>(Z)</td>
<td>-.775</td>
</tr>
<tr>
<td>Asymp. Sig. (2-tailed)</td>
<td>.439</td>
</tr>
<tr>
<td>Exact Sig. (2-tailed)</td>
<td>.445</td>
</tr>
<tr>
<td>Exact Sig. (1-tailed)</td>
<td>.222</td>
</tr>
<tr>
<td>Point Probability</td>
<td>.003</td>
</tr>
</tbody>
</table>

\( a\). Grouping Variable: Combined Year

Figure B-3-1 Results of *Hypothesis 1* from Mann-Whitney U test

2. Results of *Hypothesis 2* from K-W H test when choosing Monte Carlo option

<table>
<thead>
<tr>
<th>Ranks</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Year</td>
<td>N</td>
</tr>
<tr>
<td>Test</td>
<td>Group 1 (Years 2006 &amp; 2007)</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>Group 2 (Years 2008 &amp; 2009)</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Group 3’ (Mid-years 2011 &amp; 2013)</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td>71</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Statistics(^ab)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chi-Square</td>
<td>.838</td>
</tr>
<tr>
<td>Df</td>
<td>2</td>
</tr>
<tr>
<td>Asymp. Sig.</td>
<td>.658</td>
</tr>
<tr>
<td>Monte Carlo Sig.</td>
<td>Sig. .664(^c)</td>
</tr>
<tr>
<td>99% Confidence Interval</td>
<td>Lower Bound .652</td>
</tr>
<tr>
<td></td>
<td>Upper Bound .676</td>
</tr>
</tbody>
</table>

\( a\). Kruskal Wallis Test
\( b\). Grouping Variable: Year
\( c\). Based on 10000 sampled tables with starting seed 2000000.

Figure B-3-2 Results of *Hypothesis 2* from Kruskal-Wallis H test
3. Results of Hypothesis 3 from Mann-Whitney U Test when choosing exact test option

<table>
<thead>
<tr>
<th>Year</th>
<th>N</th>
<th>Mean Rank</th>
<th>Sum of Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011</td>
<td>8</td>
<td>11.44</td>
<td>91.50</td>
</tr>
<tr>
<td>Mark 2013</td>
<td>14</td>
<td>11.54</td>
<td>161.50</td>
</tr>
<tr>
<td>Total</td>
<td>22</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Test Statistics**

<table>
<thead>
<tr>
<th>Mark</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Mann-Whitney U</td>
<td>55.500</td>
</tr>
<tr>
<td>Wilcoxon W</td>
<td>91.500</td>
</tr>
<tr>
<td>Z</td>
<td>-.037</td>
</tr>
<tr>
<td>Asymp. Sig. (2-tailed)</td>
<td>.970</td>
</tr>
<tr>
<td>Exact Sig. 2*(1-tailed Sig.)</td>
<td>.973b</td>
</tr>
<tr>
<td>Exact Sig. (2-tailed)</td>
<td>.985</td>
</tr>
<tr>
<td>Exact Sig. (1-tailed)</td>
<td>.489</td>
</tr>
<tr>
<td>Point Probability</td>
<td>.007</td>
</tr>
</tbody>
</table>

a. Grouping Variable: Year
b. Not corrected for ties.

Figure B-3-3 Results of Hypothesis 3 from Mann-Whitney U Test

4. Results of Hypothesis 4 from Kruskal-Wallis H test when choosing Monte Carlo option

<table>
<thead>
<tr>
<th>Year of the Test</th>
<th>N</th>
<th>Mean Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Result Group 1 (Years 2006 &amp; 2007)</td>
<td>29</td>
<td>30.43</td>
</tr>
<tr>
<td>Group 2 (Years 2008 &amp; 2009)</td>
<td>21</td>
<td>27.48</td>
</tr>
<tr>
<td>Group 3 (Years 2011 &amp; 2013)</td>
<td>22</td>
<td>53.11</td>
</tr>
<tr>
<td>Total</td>
<td>72</td>
<td></td>
</tr>
</tbody>
</table>

**Test Statistics**

<table>
<thead>
<tr>
<th>Test Result</th>
<th>Test Statistic</th>
<th>df</th>
<th>Asymp. Sig.</th>
<th>Sig.</th>
<th>99% Confidence Interval</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Chi-Square</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monte Carlo Sig.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a. Kruskal Wallis Test
b. Grouping Variable: Year of the Test
c. Based on 10000 sampled tables with starting seed 2000000.

Figure B-3-4 Results of Hypothesis 4 from Kruskal-Wallis H test
5. Results of post hoc tests for Hypothesis 4 from M-W U test when choosing exact option

The above Kruskal-Wallis H test indicates that overall groups come from different populations. However, it doesn’t suggest specifically which groups differ. Therefore, post hoc tests for Hypothesis 4 are needed.

Initially, an option “Pairwise Comparison” under “All pairwise” had been used to follow up H test in SPSS. Although there are two outliers in Years 2011 & 2013, it is too subject make any conclusions from the following boxplot because it perhaps is where the difference lies (see left figure in Figure B-3-5). The differences of average rank from Years 2011 & 2013 are highlighted between other two groups respectively (see right figure in Figure B-3-5). There is no significant difference between these other two groups (between Years 2008 & 2009 and Years 2006 & 2007).

Figure B-3-5 Direct results of post hoc tests for Hypothesis 4

Although the results of asymptotic significances can be get directly form H test (see Figure B-3-5), they were unable to provide enough evidence of matching previous results from both exact
test and Monte Carlo test option. Therefore, pair comparisons of post hoc test for Hypothesis (4) by exact test have to be performed by Mann-Whitney U test as follows.

1) Group 1 vs. Group 3

<table>
<thead>
<tr>
<th>Ranks</th>
<th>Year of the Test</th>
<th>N</th>
<th>Mean Rank</th>
<th>Sum of Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Result</td>
<td>Years 2006 &amp; 2007</td>
<td>29</td>
<td>18.62</td>
<td>540.00</td>
</tr>
<tr>
<td></td>
<td>Years 2011 &amp; 2013</td>
<td>22</td>
<td>35.73</td>
<td>786.00</td>
</tr>
<tr>
<td>Total</td>
<td>51</td>
<td>35.73</td>
<td>786.00</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Statistics&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mann-Whitney U</td>
<td>105.000</td>
</tr>
<tr>
<td>Wilcoxon W</td>
<td>540.000</td>
</tr>
<tr>
<td>Z</td>
<td>-4.113</td>
</tr>
<tr>
<td>Asymp. Sig. (2-tailed)</td>
<td>.000</td>
</tr>
<tr>
<td>Exact Sig. (2-tailed)</td>
<td>.000</td>
</tr>
<tr>
<td>Exact Sig. (1-tailed)</td>
<td>.000</td>
</tr>
<tr>
<td>Point Probability</td>
<td>.000</td>
</tr>
</tbody>
</table>

<sup>a</sup> Grouping Variable: Year of the Test

Figure B-3-6 Pair comparison (G1 vs. G2) of post hoc test for Hypothesis 4 by exact test

2) Group 2 vs. Group 3

<table>
<thead>
<tr>
<th>Ranks</th>
<th>Year of the Test</th>
<th>N</th>
<th>Mean Rank</th>
<th>Sum of Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Result</td>
<td>Years 2008 &amp; 2009</td>
<td>21</td>
<td>14.79</td>
<td>310.50</td>
</tr>
<tr>
<td></td>
<td>Years 2011 &amp; 2013</td>
<td>22</td>
<td>28.89</td>
<td>635.50</td>
</tr>
<tr>
<td>Total</td>
<td>43</td>
<td>28.89</td>
<td>635.50</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Statistics&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mann-Whitney U</td>
<td>79.500</td>
</tr>
<tr>
<td>Wilcoxon W</td>
<td>310.500</td>
</tr>
<tr>
<td>Z</td>
<td>-3.781</td>
</tr>
<tr>
<td>Asymp. Sig. (2-tailed)</td>
<td>.000</td>
</tr>
<tr>
<td>Exact Sig. (2-tailed)</td>
<td>.000</td>
</tr>
<tr>
<td>Exact Sig. (1-tailed)</td>
<td>.000</td>
</tr>
<tr>
<td>Point Probability</td>
<td>.000</td>
</tr>
</tbody>
</table>

<sup>a</sup> Grouping Variable: Year of the Test

Figure B-3-7 Pair comparisons (G2 vs. G3) of post hoc test for Hypothesis 4 by exact test
B-4: Results of *Hypothesis 4* from Parametric Method by One Way ANOVA Test Using SPSS v22

1. One way ANOVA Tests

<table>
<thead>
<tr>
<th>Test Result</th>
<th>N</th>
<th>Mean</th>
<th>Std. Deviation</th>
<th>Std. Error</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Years 2006 &amp; 2007</td>
<td>29</td>
<td>44.62</td>
<td>42.337</td>
<td>7.862</td>
<td>28.52</td>
<td>60.72</td>
<td>0</td>
<td>96</td>
</tr>
<tr>
<td>Years 2008 &amp; 2009</td>
<td>21</td>
<td>37.76</td>
<td>45.669</td>
<td>9.966</td>
<td>16.97</td>
<td>58.55</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>Years 2011 &amp; 2013</td>
<td>22</td>
<td>88.73</td>
<td>19.715</td>
<td>4.203</td>
<td>79.99</td>
<td>97.47</td>
<td>21</td>
<td>100</td>
</tr>
<tr>
<td>Total</td>
<td>72</td>
<td>56.10</td>
<td>43.503</td>
<td>5.127</td>
<td>45.87</td>
<td>66.32</td>
<td>0</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sum of Squares</th>
<th>df</th>
<th>Mean Square</th>
<th>F</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between Groups</td>
<td>34303.319</td>
<td>2</td>
<td>17151.659</td>
<td>11.827</td>
</tr>
<tr>
<td>Within Groups</td>
<td>100063.001</td>
<td>69</td>
<td>1450.188</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>134366.319</td>
<td>71</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Result * Year of the Test</th>
<th>Eta</th>
<th>Eta Squared</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.505</td>
<td>.255</td>
</tr>
</tbody>
</table>

Figure B-4-1 Results of *Hypothesis 4* by one way ANOVA test

Eta squared = Sum of squares between groups / Total sum of squares = 34303.319/134366.319 = .255297

It is greater than .14. So, the result is large effect.
2. Post Hoc Tests

Test of Homogeneity of Variances

<table>
<thead>
<tr>
<th>Test Result</th>
<th>Levene Statistic</th>
<th>df1</th>
<th>df2</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>36.513</td>
<td>2</td>
<td>69</td>
<td>.000</td>
</tr>
</tbody>
</table>

Figure B-4-2 Selecting post hoc test from ANOVA

Since the test has reject the assumption of homogeneity of variances (p < .001), Tamhane T2 post hoc comparison tests are used for the most powerful adjustment.

Multiple Comparisons

<table>
<thead>
<tr>
<th>Year of the Test</th>
<th>Year of the Test</th>
<th>Mean Difference (I-J)</th>
<th>Std. Error</th>
<th>Sig.</th>
<th>95% Confidence Interval</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>2011 &amp; 2013</td>
<td>2008 &amp; 2009</td>
<td>50.965*</td>
<td>10.816</td>
<td>.000</td>
<td>23.43 - 78.50</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* The mean difference is significant at the 0.05 level.

Figure B-4-3 Post hoc test from ANOVA
Appendix C—Consent Form for Participants

Teaching Novice Programming

INFORMATION SHEET FOR PARTICIPANTS

Thank you for showing an interest in this project. Please read this information sheet carefully before deciding whether or not to participate. If you decide to participate we thank you. If you decide not to take part there will be no disadvantage to you of any kind and we thank you for considering our request.

What is the Aim of the Project?

Computer programming is a core skill for computing professionals, and the foundation of the ICT revolution. However, the introductory programming course is well known as “hard”. Our research aims to develop a new approach for teaching novice programming, which is both easy to introduce and effective in improving novice learning.

What Type of Participants are being sought?

We evaluate the framework by analysing the assessment materials produced as part of PP490.

Whether you agree to participate or not will not affect what you are required to do for PP490, nor will it affect your mark in PP490.

What will Participants be Asked to Do?

Nothing. Participation simply means that you consent for your PP490 assessments to be analysed and compared with other work.

Please be aware that you may decide not to take part in the project without any disadvantage to yourself of any kind.
Can Participants Change their Mind and Withdraw from the Project?

You may withdraw from participation in the project at any time and without any disadvantage to yourself of any kind.

What Data or Information will be Collected and What Use will be Made of it?

The assessment materials that you need to produce for PP490 will be collected and analysed. The analysis will consider evidence of use of goals and plans, and will be compared to assignments done by other students.

Participant data is being collected so that we can explore how student learning by using the new approach can be enhanced. This data will be recorded in a database using a randomly assigned ID code, with the association between these codes and actual names being stored separately in a locked filing cabinet.

The results of the project may be published and will be available in the University of Otago Library (Dunedin, New Zealand) but every attempt will be made to preserve your anonymity. You are most welcome to request a copy of the results of the project should you wish.

The data collected will be securely stored in such a way that only those mentioned below will be able to gain access to it. At the end of the project any personal information will be destroyed immediately except that, as required by the University’s research policy, any raw data on which the results of the project depend will be retained in secure storage for five years.

Reasonable precautions will be taken to protect and destroy data gathered by email. However, the security of electronically transmitted information cannot be guaranteed.

What if Participants have any Questions?

If you have any questions about our project, either now or in the future, please feel free to contact either:-

Minjie Hu or Professor Michael Winikoff, and Professor Stephen Cranefield

Department of Information Science

University Telephone Number:

479-8386, 479 8083
Teaching Novice Programming

CONSENT FORM FOR

PARTICIPANTS

I have read the Information Sheet concerning this project and understand what it is about. All my questions have been answered to my satisfaction. I understand that I am free to request further information at any stage.

I know that:

1. my participation in the project is entirely voluntary;

2. I am free to withdraw from the project at any time without any disadvantage;

3. the data (collected assessment materials) will be retained in secure storage for five years and then destroyed;

4. the results of the project may be published and available in the University of Otago Library (Dunedin, New Zealand) but every attempt will be made to preserve my anonymity.

I agree to take part in this project.

.................................................................................. ........................................
(Signature of participant) ........................................
(Date)
Appendix D—Student Workbook

Contents

1. Introduction to Programming
   1.1 History of Programming
   1.2 Computer Languages
   1.3 Problem Solving
   1.4 Programming Tools
   1.5 Flowcharts
   1.6 BYOB (Build Your Own Blocks)
   1.7 Pseudocode
   1.8 Summary
   1.9 Exercises I
   Quiz 1 Sequence

2. Making Decision
   2.1 Boolean Conditions
   2.2 The “if” Block
   2.3 Exercise II
   2.4 The Nesting of Selection
   2.5 Exercises III
   Quiz 2 Selection

3. Repetition
   3.1 Repeat for Certain Number of Times
   3.2 Repeat for Uncertain Number of Times
   3.3 Application with both Repetition and Selection
   3.4 Exercises IV
   Quiz 3. Repetition

4. Solving Problems with Goal and Plans
   4.1 Goal
   4.2 Plan
   4.3 Plan Blocks

5. A Programming Process Using Goals and Plans
   5.1 Devising Test Cases
   5.2 Analysis of Goals
   5.3 Design Using Plan Blocks
   5.4 Expanding Plan Blocks
   5.5 Merging Plan Details
   5.6 Simplifying the Merged Details
   5.7 Summary
   5.8 Exercise V

6. BYOP (Build Your Own Plans)
   6.1 Overview of Building Process Plan Blocks
   6.2 Scaffolding Blocks for Building Plans
   6.3 Defining New Plan Blocks Using a Process Pattern
   6.4 Other Patterns of Process Plan Blocks
   6.5 Summary
   6.6 Exercise VI
   6.7 Have Fun with Goals and Plans
1. Introduction to Programming

Programming is a core skill for computing professionals and the foundation of the ICT revolution. It is the compulsory course for Diploma of ICT programme.

1.1 History of Programming

A loom is a device used to weave cloth. In 1801, Frenchman Joseph Jacquard made a loom controlled by a set of punched hole cards (see the left of Figure 1-1). One raw of punched hole card is used as a command to control a certain action. A set of the punched hole cards is the program of the whole procedure. When the hole is implemented by the switch (on or off), a set of switches became the modern electronic computer program (see the right of Figure 1-1).

In 1940s, John von Neumann – developed stored program concept. After a program is loaded to the computer memory (RAM, or Random Access Memory), the CPU (Central Process Unit) follows its commands to process data. Accordingly, the task for the people to manage the computer commands into a logical order in order to solve a certain problem is called programming, which is one of the most important computer courses today.

Figure 1-1. The Jacquard Loom corresponds to one row of punched hole card (1801) and the main control panel of the first computer, ENIAC (1946). (Source: www.wikimedia.org)
1.2 Computer Languages

- **Low Level languages** are used more directly closing to computer hardware.

**Machine language:** is the native language of a computer CPU. All the instructions and data are binary which represented as zeros and ones.

**Assembly language:** is functionally equivalent to machine language but easier for people to read, write, and understand in English word and letters. However, the instructions are different according the CPU structure. Different CPU has different assembly language, i.e. iMac has different CPU comparing to IBM-PC; Mainframe, mini-computer and PC have different CPUs.

**Assembler:** translates the assembly language into the corresponding machine language code.

- **High-level languages** are generally applied to any computers regardless which CPU the computer is made of. The languages are more closing to natural human languages rather than computer hardware. Examples: COBOL, FORTRAN, Basic, Pascal, C, C++, Java, Python, Visual J++, Visual Basic, C#, etc.

**Compiled and Interpreted Languages:** there are two ways to translate the high level language into the low level language. Most of them compiled the whole high level language code into level language in once. They are also called the compiled languages. Others, i.e. BASIC, JavaScript, etc. Interpret the high language into low level language sentence by sentence. They are also called interpreted languages.

1.3 Problem Solving

**What is a program?**
A program is a set of instructions and data that enable computer to solve a problem.

**What is Programming?**
Programming is the process of developing a program.

Since program, or software, refers to a set of instructions for the computer, the programmer has to know how to solve problems in order to tell computer what to do.

- **Developing the solution to a problem**

**Algorithm** – The order of instructions to solve a problem.

- Problems are solved by finding out what data are known and what outputs are asked for. Then set up the relationship the given data and the requested output in order to produce the output from the given data. This procedure is also called an algorithm.
Basic Steps of Problem Solving

1. Determine Output
2. Identify Input
3. Determine process necessary to turn given Input into desired Output

Example 1-1: What is the wages for someone working for 10 hours with the pay rate of $15 per hour?

Output: a number showing the wages for the income.
Input: the working hours and time the pay rate
Process: wages = hours x rate

1.4 Programming Tools

Three tools used in this course to convert algorithms into computer programs:

Flowcharts - Graphically depict the logical steps to carry out a task and show how the steps relate to each other.

BYOB (Build Your Own Blocks) – is a visual programming language for beginners, which is also an extension to Scratch. The program can be built up through drag-and-drop. It is easy for beginners to try and test their design of algorithm.

Pseudocode - Uses English-like phrases to outline the program for the transition to the real text based computer language, i.e. VB, C#, etc.

1.5 Flowcharts

Graphically demonstrate the logical steps to carry out a task and show how the steps relate to each other.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Flowline" /></td>
<td>Flowline</td>
<td>Connecting symbols and indicating logic flow</td>
</tr>
<tr>
<td><img src="image" alt="Terminal" /></td>
<td>Terminal</td>
<td>Representing the start or the end of a task</td>
</tr>
<tr>
<td><img src="image" alt="Input or Output" /></td>
<td>Input or Output</td>
<td>Representing input (or entering) and output (or displaying) operations.</td>
</tr>
<tr>
<td><img src="image" alt="Process" /></td>
<td>Process</td>
<td>Representing data processing (or manipulation) operations.</td>
</tr>
<tr>
<td><img src="image" alt="Decision" /></td>
<td>Decision</td>
<td>Representing logic (or comparison) operations. It has two exit paths for the answers (“yes” or “no”) to a question for it. The one of usage depends on the size of contents inside the symbol.</td>
</tr>
</tbody>
</table>

**Figure 1-3. Flowchart symbols**

**Flowchart example:**

![Flowchart example](image)

**Figure 1-4. The Flowchart of wages**

Flowchart is so clearly illustrating the logical flow of programming. Thus, it is a valuable tool in the education for the beginners. However, it is time-consuming to draw and difficult to update for the complicated algorithm. Professional programmers are more likely to favour pseudocode.
1.6 BYOB (Build Your Own Blocks)

**What is BYOB?**

*BYOB* is an extension to Scratch which allows building a subprogram as a block.

Recently, visual programming environments (VPEs) such as Scratch, etc. provide an attractive, easy, and fun way for students to learn programming. These VPEs are similar in having pre-built blocks and a program is built up by dragging-and-dropping different blocks. This tutorial aims to teach you how to program in such an environment without considering syntax errors, but focusing on the programming skills. A new powerful language, BYOB, is used instead of Scratch in this Module.

![Visual Programming Environment of BYOB](image)

**Figure 1-5.** The Visual Programming Environment of BYOB

In BYOB (see Figure 1-5), a new project/program is started from the "Menu of Project/Program". All the blocks are classified into different “Block Palette” under the “Menu of Blocks”.

- **Menu of Blocks:** Different categories of blocks
- **Stage:** Displaying results
- **Menu of Project/Program:**
- **Block Palette or Button Screen:** The source place where the Blocks are dragged from for programming
- **Code Screen:** The destination where the dragged blocks are dropped to for assembling a program
- **Start/Pause/Stop the project/program**
Blocks”. These blocks can be dragged from these “Block Palette” and dropped into the “Code Screen” in order to be assembled into a program. Through the command of “Start/Pause/Stop”, the results of the program are displayed on the “Stage” screen.

- Variables

**What is a Variable?**

A variable is a location where data can be stored, accessed and modified by computer instructions.

A variable has a name, type (e.g. string, number), and a value. For example, the daily work hours are represented by a variable named Hours, which also has a data type of Integer (computers treat whole numbers – i.e. Integers such as 0, 42, and -12 – and floating point numbers – such as 1.3 and -3.14159 – differently) and a possible value of 8. In the example of calculating wages, variables Hours and Rate are used to keep the input data. A variable wages is also needed to keep the processed result and is then used for output.

In BYOB, a variable can be created by clicking the button “Make a variable” (see Figure 1-5) in the Block Palette under the menu of button “Variable”. For example a variable named “Result” can be created as in Figure 1-6. The variable “result” will be listed in the Block Palette (see the left in Figure 1-7). By default, a box next to the variable is ticked for displaying it on the BYOB Stage (see the right in Figure 1-7). Clicking the box will remove the tick and hide the variable from the stage. The variable can be used by dragging and dropping it into the Code Screen (see the middle in Figure1-7). After double clicking the variable in the Code Screen, its value will be shown next to it.

![Figure 1-6. Example of creating a variable](image)

![Figure 1-7. What you can see about a variable](image)
• Block

What is a Block?
A command or instruction in the program is also called a block in Scratch and BYOB.

A Block changes the status of variables that are used by the program, e.g. the command “set Sentinel to -1” means after the execution of this command the variable “Sentinel” has a value of “-1”. In this command, variable “Sentinel” and value “-1” are also components of the block. (Using BYOB, a group of commands can also be combined by the user to form a new singular block. The new block contains the details of this group of commands like a subprogram.)

In BYOB, a common way to start a program is by clicking the “flag” button so that the program begins from the block “When [flag] clicked”. Furthermore, during the processing of variables, blocks are executed in a given order, and there are three basic order types: sequence, where blocks are executed one after another in a sequential order; selection, where a number of blocks are options, and only one of them is selected; and loop, where the same group of blocks is executed repeatedly. We return to the processing order in Section 2 when we describe the various so-called “control” blocks which BYOB provides. These control blocks are what you use to specify the order of execution of blocks.

• Input, Process, and Output Blocks

All the rest of blocks in the program are dealing with the data in the variables. These blocks can be classified as being either input, output, or processing blocks. Hence, generally there are three types of blocks: Input, Process, and Output (I-P-O).

1) Output Block

What is an Output Block?
The Output Block displays the result on the “Stage”.

The result can be a single text message (see Figure 1-5) or a combination of text and the value of a variable. The results can be staying on the “Stage” screen until the next program starts or only for a certain period of time, e.g. 2 seconds. When the value of variable “Result” is 5, the following example (see Figure 1-8) will display the message “Result is 5” for only 2 seconds on the “Stage” screen.

![Figure 1-8. Example of output block](image-url)
2) Input Block

**What is Input Block?**
The Input Block allows user to input a value through keyboard to the temporary variable “answer”.

The Input Block asks the user to input a value as the answer to a certain provided message. The user’s answer will typically be stored in a variable in order to keep it for further processing. For example, in the following human-computer dialog (see Figure 1-9), the “answer” of name “John” is stored in the variable “Name” and then the variable “Name” is combined as part of the output “hello John”.

![Figure 1-9. Example of input block](image)

3) Process Block

**What is Process Block?**
The Process Block changes the value of a variable.

Except for the Input and Output blocks, the rest of blocks in BYOB are process blocks. Most of the process blocks are applied to directly process data in terms of values, variables, colours, sound, etc. A certain number of them (all the yellow colour blocks in the Block Palette under the “Control” menu block) are used to control the order of block such as selection and loop (we will discuss them in the next section).

The example of Wages by BYOB is shown in Figure 1-10. The process blocks in this program are “set” blocks. The first two “set” blocks store the values of answer into variables Rate and Hours, and the last “set” block changes the value of variable Wages after the calculation.

![Figure 1-10. The example of Wages in BYOB](image)
• Desk-check

*What is Desk-check?*
Desk checking is a way of tracing through instructions/commands while using a table to track the current value of variables. These instructions/commands can be the statements in the Pseudocode or the blocks in the flowchart or BYOB. The process is done manually by using only pen and paper rather than computer.

A Desk-check Table typically starts with a column of all the command blocks from a program. It follows multiple columns of variables against every command blocks in each row of the table. These columns are put in the order of variables being executed, such as input, process, and output. Finally, it has an input/output column, which shows the input data and the end results. In the example of Wages calculation (see Figure 1-10), its test cases with input testing data and expected results are listed in Table 1-1. A Desk-check Table for each test case is designed as Table 1-2, which has three major columns, program blocks, variables, and input/output data.

The desk-checking for the Test Case (1) starts from the first block “ask”. This block does not affect the value of any variables, but displays a message to ask the user to input a value of pay rate. In this test case, a value of 15 can be entered into a variable “answer” shared by all the “ask” blocks. The second block “set Rate to answer” assigns the value 15 from “answer” to variable Rate. Similarly, when the third block displays a message to ask the user to input a value of working hours, a value of 20 will be entered into the shared variable “answer”. Then the fourth block “set Hours to answer” assigns the value 20 from “answer” to variable Hours.

After completing the input, the fifth block “set Wages to Rate * Hours” starts processing the values of both Rate and Hours and then assigning the result to the variable Wages (300 = 15 * 20). Finally, the last block “say” displays a message combing with the text and the value of Wages as “Your wage is $300”. The desk-check results are matching to those we expected in the test case. We could only claim that there is no logic error under this case.

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Input Data</th>
<th>Expected Output</th>
<th>Rate</th>
<th>Hours</th>
<th>Wages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Case (1)</td>
<td>15</td>
<td>20</td>
<td>300</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test Case (2)</td>
<td>15</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
Table 1-2. Desk-check Table for Test Case (1)

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Variables</th>
<th>Input/Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ask What's your pay rate ($/hr)? and wait</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set Rate to answer</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>ask What are your working hours? and wait</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>set Hours to answer</td>
<td>15 20</td>
<td></td>
</tr>
<tr>
<td>set Wages to Rate * Hours</td>
<td>15 20 300</td>
<td></td>
</tr>
<tr>
<td>say join Your wage is $ Wages for 2 secs</td>
<td>15 20 300</td>
<td>Your wage is $300</td>
</tr>
</tbody>
</table>

The desk-checking for the Test Case (2) is in the same way as the above example, except the input values of 15 and 0 for Rate and Hours. The final message is “Your wage is $0”. This can be explained as “no work, no payment”. The value 0 here is called a boundary value as it is at the extreme of a range of acceptable values. It is necessary to test boundary value so that the correct path is actually followed for this value.

Once the value of testing data is beyond the border (e.g. 0) to a negative value there will be totally different test cases, which may or may not be accepted. For example, if the value of Hours is -2 and Rate is 15, the result will be -30. It may be accepted as a deduction from previous overpay. However, if the value of Hours is -2 and Rate is -15, the result will be 30. This would be an unexpected test case.

Debugging

**What is debugging?**

*Debugging is the process of checking through individual commands or blocks to locate and fix errors (known as bugs) in a program. When debugging, you start with a known problem, investigate through the program details, isolate the source of the problem, and then fix it.*

Debugging tools (called debuggers) help identify coding errors at various development stages. In the BYOB programming environment, click on a block or a group of blocks can trigger the execution of this block or blocks. Thus, a program can be executed step-by-step by separating them and clicking on them one-by-one, which helps to isolate and locate the bugs in a long program by comparing to the results in Desk-check Table. We separate all the blocks except the “ask-set-answer” of the Wages program (see Figure 1-11). In order to make it start again, we also add three “set to 0” blocks to set the initial value of 0 to variables Rate, Hours, and Wages. After clicking these “set to 0” blocks, you can see the value of 0 is set to the three
variables. (Make sure the box before these variables in the Block Palette has been ticked so that these variables will be displayed on the Stage.)

The debugging starts from clicking the first “ask-set-answer” blocks. After enter the value of 15 for the pay rate, you will see the variable of Rate has been assigned to a value of 15 on the Stage. Similarly, click on the second “ask-set-answer” blocks to enter the value of 20 for the working hours. You will see the variable of Hours has been assigned to a value of 20 on the Stage. When you click the block “set Wages to Rate*Hours”, you will see the variable of Wages has been assigned to a value of 300 on the Stage. Finally, click the “say” block, a message of “Your wage is $300” is showing on the Stage (see Figure 1-12).

However, if your results are different from those in the Deck-check Table, firstly make sure the syntax of variables and operators are applied correctly. For example, during the above debugging, if the variable of Wages still has a value of 0 on the Stage after clicking the block “set Wages to Rate*Hours”. It may lead by the value of 0 from either Rate or Hours. Thus, you need to check both “ask-set-answer” blocks to ensure that two answers have been assign to variables Rate and Hours. It might have two “set Rate to answer” blocks or two “set Hours to answer” blocks but missing one of them, which leads one of the variables still has its initial value of 0. If the variable of Wages still has a value of 35 on the Stage after clicking the block “set Wages to Rate*Hours”. The operator “+” may be used instead of “*”.

Figure 1-11. The staring of debugging

Figure 1-12. The end of debugging
Secondly, make sure the order or logic of processing data is correct. For example, if the block “set Wages to Rate * Hours” is put before the “ask-set-answer” input blocks, the final output will be still zero even the values of input are correct (see Figure 1-13). In this case, you will find that the block “set Wages to Rate*Hours” must be put after these “ask-set-answer” input blocks and before the “say” block. Otherwise, the output message will be incorrect.

Figure 13. An example of debugging

1.7 Pseudocode

Pseudocode (Pseu means false) is a shortened plain English version of actual computer code. The English-like statements are applied to outline the process of Flowchart or BYOB. It focuses on the steps to solve a problem rather than the details of using a real computer language. After the Pseudocode is completed, it can be easily translated into a real computer language, i.e. VB, C#, etc. The following the Pseudocode for the Wages program.

- Input Rate (Input)
- Input Hours (Input)
- Wages = Rate X Hours (Process)
- Output Wages (Output)

1.8 Summary

New concepts studied in this section:

- History of programming
- Low level and high level computer languages
- Program
- Algorithm
- Basic steps of problem solving
- Three tools for programming in the module.: flowchart, BYOB, Pseudocode
- Variable
- Deskcheck
- Debug
The basic steps of programming:

1) Determine variable (s) for the output
2) Identify variable (s) in the input
3) Set up the relationship between input and output variables.
4) Describe the process or algorithm in the correct order by flowchart
5) Convert from flowchart to BYOB
6) Deskcheck
7) Debug and Test in BYOB
8) Present the program in Pseudocode

1.9 Exercises I

1) You are asked to write a program to display the price including GST (15%) after someone entering the price. For example, if the price is entered as $100, then the price including GST will be $115.

a) What is/are the output variable(s)?
b) What is/are the input variable(s)?
c) What is the relationship between the input and output variables?
d) Use Flowchart to present your design of how you solve this problem.
e) Transfer your flowchart into BYOB to try and test your design.
f) Describe your program in Pseudocode from the BYOB.

2) The order of instructions is very important in the programming. Find out by completing exercise a) from 1: \DipICT L5\PP490\Ex\Ex1A.tif.

3) Change the order BYOB blocks in the above question 1). i.e. drag-and-drop the process block before the input block, etc. Find out the result and explain why.
1. Which of the following is a valid name for a variable?
   (A) Two_One
   (B) Two
   (C) Two-One
   (D) TwoOne
   (E) Two One

2. Which is the correct presentation for the following equation?
   \[ \frac{a + b}{b} \]
   (A) \( y = \frac{a + b}{b / c} - d \)
   (B) \( y = \frac{(a + b)}{(b / c - d)} \)
   (C) \( y = \frac{a + b}{b / c} - d \)
   (D) All of the above
   (E) None of the above

3. What will be displayed when the following lines are executed?
   ```
   x = 3
y = 1
z = x + y * x
x = y
z = x + z
   ```
   Output \( x + y + z \)
   (A) 4
   (B) 9
   (C) 10
   (D) 13
   (E) None of the above

4. What will be displayed when the following lines are executed?
   ```
   a = 10
b = 20
a = b
   ```
   (A) \( a = 10, b = 10 \)
   (B) \( a = 10, b = 20 \)
   (C) \( a = 20, b = 10 \)
   (D) \( a = 20, b = 20 \)
   (E) \( a = 0, b = 0 \)

5. What will be displayed when the following lines are executed?
   ```
a = 5;
b = 3;
c = 7;
a = c;
b = a;
c = b;
   ```
   (A) \( a = 5, b = 3, c = 7 \)
   (B) \( a = 7, b = 3, c = 5 \)
   (C) \( a = 3, b = 3, c = 3 \)
   (D) \( a = 5, b = 5, c = 5 \)
   (E) \( a = 7, b = 7, c = 7 \)

6. Assume that \( x, y, \) and \( temp \) are numeric variables. Which of the following lines of code swaps the values of \( x \) and \( y \)?
   ```
   x = y
   y = temp
   ```
   (A) \( x = y \)
   (B) \( x = temp \)
   (C) \( temp = x \)
   (D) \( x = y \)
   (E) \( temp = x \)
2. Making Decision

In some situation, not all the instructions have to be executed. If the answer to a condition is “Yes” then one group of instructions is executed. If the answer is “No,” then another is executed. The flowchart and Pseudocode of making decision from the value of condition are show in Figure 2-1.

If the condition is ture Then
    Process task(s) 1
Else
    Process task(s) 2
End If

Figure 2-1 The flowchart and Pseudocode of making decision

2.1 Boolean Conditions

A Boolean condition is built up using Boolean Logic operators (see Figure 2-2) in a diamond shape in BYOB. A Boolean condition has a value which is either “true” or “false”. The details of Boolean logical expressions are described in Table 2-1.
### Table 2-1: The Boolean Logical Expressions

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Description</th>
<th>Expression Pseudocode / BYOB</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>less than</td>
<td>$A &lt; B$</td>
<td>When $A = 3$, $B = 5$, $3 &lt; 5$ is True.</td>
</tr>
<tr>
<td><code>=</code></td>
<td>equal to</td>
<td>$A = B$</td>
<td>When $A = 5$, $B = 5$, $5 = 5$ is True.</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>greater than</td>
<td>$A &gt; B$</td>
<td>When $A = 3$, $B = 5$, $3 &gt; 5$ is False</td>
</tr>
<tr>
<td>and</td>
<td><em>Return Yes when both conditions are True</em></td>
<td>$(A &lt; B)$ and $(A &gt; 0)$</td>
<td>When $A = 3$, $B = 5$, the expression is True; When $A = 5$, $B = 3$, the expression is False.</td>
</tr>
<tr>
<td>or</td>
<td><em>Return Yes when either (or both) conditions are True</em></td>
<td>$(A &lt; B)$ or $(A = B)$</td>
<td>When $A = 3$, $B = 5$, the expression is True. When $A = 5$, $B = 3$, the expression is False.</td>
</tr>
<tr>
<td>not</td>
<td>Returns the opposite logical value</td>
<td>not $(A = B)$</td>
<td>When $A = 5$, $B = 5$, since $(5=5)$ is true, not $(5 = 5)$ is False; When $A = 3$, $B = 5$, since $(3=5)$ is No, not $(3 = 5)$ is True;</td>
</tr>
</tbody>
</table>

*In the Table 2-2, the expression “X AND Y” is True when both conditions “X” and “Y” are True. Otherwise, it is No. On the other hand, in the Table 2-3, the expression “X OR Y” is No when both of conditions “X” and “Y” are No. Otherwise, it is True.*

### Table 2-2. True Table for AND

<table>
<thead>
<tr>
<th>AND</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td>X</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

### Table 2-3. True Table for OR

<table>
<thead>
<tr>
<th>OR</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True</td>
</tr>
<tr>
<td>X</td>
<td>True</td>
</tr>
<tr>
<td></td>
<td>False</td>
</tr>
</tbody>
</table>

2.2 The “if” Block
Having seen Boolean conditions, we now look at how to use them in the “if” block to implement selection. There are two “if” blocks in the Block Palette (which can be accessed by clicking the button “Control” from the Menu of Blocks, see Figure 2-3). Both of them have a slot in the diamond shape, which fits in a Boolean condition.

The first “if” block provides a single branch to hold other blocks. When its logical condition is true, the blocks in this branch will be processed; otherwise, the “if” statement simply has no effect. In other words, the first “if” block decides whether to execute the statements within it depending on whether its Boolean condition is True or False.

The second “if” block provides two separate branches. If its condition is true, only the blocks in the first branch will be processed. On the other hand, if the condition is false, only the blocks under “else” will be processed. In other words, the second “if” block decides which of the two branches to execute, depending on whether its Boolean condition is true (first branch) or false (second branch).

**Figure 2-3.** The “if” blocks

**[Example 2-1]** Write a program to input the pay rate and working hours and then output the wages. There are two cases for calculating wages: (1) Normal: When Hours <= 40, Wages = Hours * Rate; (2) Overtime: When hours > 40, Wages = (40 * Rate) + ((Hours - 40) * Rate * 1.5).

The analysis is similar to Example 1-1. The Input variables are Rate and Hours and the output variable is Wages. However, the relationship between the Input and output variables is changed to two different formulas, (1) and (2), under the conditions of working hours. See the sample data and results in Table 2-2. The value of 40 is the boundary value of the condition “Hours <= 40” to use formula one. Otherwise, if “Hours > 40”, e.g. Hours = 41, it will use formula two. The flowchart is shown in Figure 2-4.
To ensure the algorithm works correctly (without logic errors), more test cases of Deskcheck are considered as in Table 2-4, especially a value at the extreme of a range of acceptable values, i.e. Hours = 40 (see Test Case (2)). Because this value is at the point between two different formulas to calculate the Wages, it is also called as boundary value. An example of value past the boundary value (Test Case (3)) is shown in Table 2-5. Another extreme case is discussed as Test Case (4).

**Table 2-4 Test Cases for Wages**

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Input Data</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rate</td>
<td>Hours</td>
</tr>
<tr>
<td>Test Case (1)</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Test Case (2)</td>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>Test Case (3)</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Test Case (4)</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

**Figure 2-4.** The Wages have two formulas

```
Wages = Hours * Rate
Wages = (40 * Rate) + ((Hours - 40) * Rate * 1.5)
```
According to the design in flowchart, it is converted into BYOB program as in Figure 2-5. Through the debugging by the same input data as in the Deskcheck, it confirms that there are no bugs from the original design when testing with these test cases. Finally, a list of Pseudocode of this program is as Figure 2-6.

**Table 2-5 Example of Desk-check for Test Case (3)**

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Variables</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Rate</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Input Hours</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Hours&lt; 40 or Hours = 40?</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Wages = Hours * Rate</td>
<td>10</td>
<td>50</td>
</tr>
<tr>
<td>Output Wages</td>
<td>10</td>
<td>50</td>
</tr>
</tbody>
</table>

**Figure 2-5** Example: Calculating Wages using an If block

Input Rate
Input Hours

If (Hours< 40 or Hours = 40 ?)
Wages = Rate X Hours
Else
Wages = (40 * Rate) + ((Hours - 40) * Rate * 1.5)
End If

Output Wages

**Figure 2-6** Pseudocode of Wages Program with If
2.3 Exercise II

You are required to complete tasks from a) to e) for the following four exercises.

a) Identify the output variable(s), input variable(s), and the relationship between the input and output variables.
b) Use Flowchart to present your design of how you solve this problem.
c) Desk-check of your design.
d) Transfer your flowchart into BYOB. Debug and test it.
e) Describe your BYOB program in Pseudocode.

1) Write a program which allows the user to enter their age. The program should then display either "You are still young" or "You are getting old" depending on whether the person’s age is greater than 70 or not.

2) Write a program in which the user enters two numbers. The program is to divide both numbers and display the result. However, if the divisor is zero, it will display a message of data error.

3) Write a program in which the user enters their wages. The program is to calculate and display the amount of tax payable on the wage. Use the (simple) tax rule: Salaries under $10,000 pay 0% tax, wages of $10,000 or more pay 20% tax.

4) Write a Sale Price Calculator program which inputs a Quantity and UnitPrice, and then computes the SalePrice as follows: (1) Normal: When quantity < 10, SalePrice = Quantity * UnitPrice; (2) Discount for bulk purchase: When quantity >= 10, SalePrice = Quantity * UnitPrice * 0.9

2.4 The Nesting of Selection

Each Selection Plan (i.e. “If” block) can only provide a maximum of two options. For more complex situations, where there might be multiple options, you can combine multiple “If” blocks. This is called “nested selection”, and can provide (N+1) options using N Boolean conditions.

[Example 2-2] Write a program that asks the user to enter an Integer. The program then will identify and display its value range as “Less than 0”, “Between 0 to 49”, “Between 50~100”, and “Greater than 100”.

In this problem, assume the input variable is Number. The output variable is the result for the message of value range. In order to have four options for different messages, there must be three conditions within three “if” blocks. A flowchart (see Figure 2-6) is much clearly described the multiple results from the various conditions.
The first condition from lowest end of the scale is "Number < 0". It has two possible values “True” (or answer is Yes) and “False” (or answer is No). When the answer is Yes, it means Number < 0 and the result is “A) number less than zero 0”. When the first condition is False, the answer is No, it means Number >= 0. In this case we go on to consider the second condition “Number < 50”.

When the second condition is True, it means Number < 50. Meanwhile, because it is also under the False value (Number >= 0) of previous condition, the condition at the level is actually a compound condition of (Number < 50) and (Number >= 0). In other word, the result is “B) number between 0 to 49”.

Figure 2-7 The flowchart of nested IF
When the second condition is False, it means Number $\geq 50$. It follows the third condition “Number $\leq 100$”. When it is True, it means Number $\leq 100$. Since it is under the False value (Number $\geq 50$) of previous condition, the compound condition is actually (Number $\leq 100$) and (Number $\geq 50$). In other word, the result is “C) number between 50 to 100”.

When the third condition is False, it means Number $> 100$. In other word, the result is “D) number greater than 100”. Both BYOB and Pseudocode are in Figure 2-8 and Figure 2-9.

Figure 2-8. The nested “if” blocks

Input Number

\[
\text{If (Number < 0)} \\
\text{Result = “Less than 0”} \\
\text{Else} \\
\text{If (Number < 50)} \\
\hspace{1cm} \text{Result = “Between 0 to 49”} \\
\text{Else} \\
\hspace{1cm} \text{Result = “Between 50~100”} \\
\text{Else} \\
\hspace{1cm} \text{Result = “Greater than 100”}
\]
If (Number < 50)  
    Result = “Between 50 to 100”
Else
    Result = “Greater than 100”
End If
End If
End If

Output Result

Figure 2-9. The Pseudocode of nested “If”

The “if” Blocks
- An “if” block separates some blocks into one or two optional branches.
- According to the block’s Boolean condition, zero or only one branch will be processed.
- In order to provide more options (e.g. N + 1), many (e.g. N) “if” block are nested.

2.5 Exercises III

1) Study the following program to identify how many possible outputs from the program are possible. Fill in the following table with these possible outputs and relevant input value range (e.g. for a score of 0-49 the grade is D). Assume that the input is a whole number (i.e. fractional scores such as 73.8 are not possible). Convert the BYOB program into Pseudocode.

Table 2-6. The Scale of Grade

<table>
<thead>
<tr>
<th>Score Range (Input)</th>
<th>Grade (Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
2) Assume that Today is Thursday and it's raining. What does the following algorithm suggest you do?

```plaintext
If (it is the weekend)
    If (it is raining) Then
        read a book
    Else
        go on a picnic
    End If
Else
    If (it is sunny)
        work outside
    Else
        study Visual Basic inside
    End If
End If
```
1. read a book
2. go on a picnic
3. work outside
4. study Visual Basic inside

3) Study the flowchart in the exercise (a) from I: \DipICT L5\PP490\Ex\Ex1B.tif and answer the questions i), ii), & iii). Convert the flowchart to BYOB and Describe the BYOB program in Pseudocode.

4) Study the flowchart in the exercise 4 in Ex3B from I: \DipICT L5\PP490\Ex\Ex2A_8D.tif and answer the questions (a) and (b). Convert the flowchart to BYOB and Describe the BYOB program in Pseudocode.

Quiz 2 Selection

1. What will be displayed by the following program?
   
   ```
   a = 5
   b = 3
   c = 6
   If (a > c)
      x = 1
   Else
      If (b > c)
         x = 2
      Else
         x = 3
      Output x
   End If
   End If
   ```
   
   (A) 1
   (B) 2
   (C) 3
   (D) None of the above.

2. What will be the output of the following program?
   
   ```
   x = 3
   y = 3
   If (x > y)
      z = x + y
   Else
      z = y - x
   End If
   Output z
   ```
   
   (A) 6
   (B) 3
   (C) 0
   (D) No output
3. Repetition

We have seen one form of processing order, conditional processing, which is described using “if” blocks. We now turn to considering how to specify repetitive processing.

Sometimes a group of blocks will need to be repeatedly processed. This is specified in BYOB using a loop control block. There are four different loop control blocks in BYOB. We only discuss two of them:

- “repeat [10]” — repeatedly process all the blocks inside the loop block a certain given number of times, e.g. 10; and
- “repeat until <condition>” — repeatedly process all the blocks inside the loop block, until the condition is true.

3.1 Repeat for Certain Number of Times

Once we have set up a fixed number of times, i.e. three times, the block “repeat [3]” will repeatedly process all the details which are put inside the “repeat” block. These details are also called loop-body. When the repeat block starts, there is an internal counter (e.g. Times) automatically counting the times of repetition by increasing 1 for each time. For example, the counter variable Times starts from 0. It is false of “equal to 3”, but only for the repetition. After the first repetition, it changes to 1. With three times of repetition, it becomes 3. It is true of “equal to 3” to stop the repetition. However, we normally cannot see the hidden counter variable inside the “repeat” block. The loop is described in flowchart as in Figure 3-1.

A common pattern is to calculate the sum of a sequence of values. The variable that is used to compute the sum is also called “accumulator”. Usually, a repeat block is used to repeatedly update the “accumulator” by one of the sequence values. Before the loop, the “accumulator” usually has an initial value of zero.

[Example 3-1] Write a program to input three values and to display their sum.
Since we are reading a given number of values, namely three, we can use a “repeat [3]” loop. The Output variable is Sum and the Input variable is Number. The Sum must have an initial value of zero before the loop. Each value is input by the “ask”-“set”-“answer” blocks inside the loop. The repeatedly updated “accumulator” by each of the Number values is represented as “Sum = Sum + Number” inside the body of this loop. Each time the loop is executed, the running total of Sum is updated. Finally, the Sum is displayed. See the program in Figure 3-2.

![Figure 3-2. An example of a repeat block](image)

```plaintext
when clicked
set Sum to 0
repeat 3
  ask Enter a value: and wait
  set Number to answer
  set Sum to Sum + Number
say join Sum is Sum
end repeat
output Sum
```

![Figure 3-3. Pseudocode of repeating certain times](image)

### 3.2 Repeat for Uncertain Number of Times

In most applications, it is hard to predict how many times the loop will be repeated. The “repeat until” block is used for the uncertain number of times repetition.  

[Example 3-2] Write a program to repeatedly input a sequence of values and to display their sum. The program should stop inputting when the value read is -1.
The variables and their relationship are similar to the previous example. However, since we do not know how many values to read, we need to use a conditional repeat block which checks for the stopping condition (i.e. the read value being the sentinel value of -1). However, in order to avoid processing the sentinel value in the loop, we need to change the order of input a value and processing this value. Hence, we have one input before the loop to initialise the condition for starting the loop. We also need to have another input inside the loop to update the condition for stopping the loop. See the flowchart and BYOB program in Figure 3-3, 3-4.

**Figure 3-4** The Repeat of fixed number of times

![Flowchart for sum](image)

**Figure 3-4.** The example of sum

```
when clicked
set Sentinel to -1
set Sum to 0
ask join Enter a value, stops when Sentinel and wait
set Number to answer
repeat until Number = Sentinel
set Sum to Sum + Number
ask join Enter a value, stops when Sentinel and wait
set Number to answer
say join Sum is Sum
```
Sentinel = -1  
Sum = 0  
Input Number  
Repeat Until (Number = Sentinel)  
  Sum = Sum + Number  
  Input Number  
End Repeat  
Output Sum

Figure 3-5. Pseudocode of Repeat Until

3.3 Application with both Repetition and Selection

[Example 3-3] Write a program to repeatedly input a sequence of values and to display the count of even numbers from them. The program stops inputting when the value is -1.

This is similar to the previous Example, but we need to only count even numbers. An even number can be identified by the function “mod” which gives the remainder, e.g. 8 mod 3 = 2 because dividing 8 by 3 has a remainder of 2. Specifically, a number is an even number when dividing it by 2 has a remainder of 0. A variable Count is used to count the number of even numbers. It has an initial value from zero, and each even number will increase the count (count = count + 1, or simply “change [count] by [1]”). The flowchart and BYOB program is in Figure 3-6 and 3-7.

Figure 3-6 The Repeat of fixed number of times
Write a program to repeatedly input a sequence of values and to display the largest number input. The program stops inputting when the value is -1.

The Output variable of this program is Max and Input variable is Number. We assume the initial maximum value is a small number, e.g. -99999. Similar to previous two examples, a conditional loop with “sentinel” is used in order to input a sequence values. Two inputs are also used to input value to variable Number, one before the loop and one inside the loop. However, if a new value is greater than the current value of Max, then the new value will become the new maximum value. Therefore, an “if” block is added inside the loop to check whether the new value is the greater than the current maximum value. After inputting all the values, the value of Max is the final maximum value.
3.4 Exercises IV

Study the following THREE questions 1), 2), and 3) and complete the tasks from a) to h).
   a) What are the variables in this program?
   b) Describe the processing these variables in flowchart.
   c) Prepare a test case, including both input and predicted output.
   d) Desk check your design in flowchart
   e) Implement the flowchart in BYOB.
   f) Test and debug BYOB program using the prepared test case.
   g) What does your program do if the very first number given is -1?
   h) Write Pseudocode of the program.

1) Write a program that will read in numbers and output the times that numbers have been inputted. Stop reading when the value -1 is input.

2) Write a program to repeatedly input a sequence of values and to display the Sum of odd numbers from them. The program stops inputting when the value is -1.
   (Note: the number is odd when the reminder of dividing it by 2 is 1. i.e. “number mod 2 = 1”)

3) Write a program to repeatedly input a sequence of values and to display the lowest number read. The program stops inputting when the value is -1.

Quiz 3. Repetition

1. (a) What is the last output of num by the following code segment?
   (b) What is the last value of num?
   ```
   num = 0
   Repeat Until (num = 5)
   Output num
   num = num + 1
   End Repeat
   ```
   (a) (A) 3; (B) 4; (C) 5; (D) 6
   (b) (A) 3; (B) 4; (C) 5; (D) 6

2. What is wrong with the following loop?
   ```
   index = 10
   Repeat Until (index = 5)
   Output "Hello!"
   index = index + 1
   ```

332
End Repeat
(A) The test variable should not be changed inside a loop.
(B) The test condition will never be true.
(C) This is an infinite loop.
(D) Nothing

3. What will be the output of the following program?
   \[
   \text{num} = 10
   \]
   \[
   \text{Repeat Until} \ (\text{num} = 1)
   \]
   \[
   \text{Output num;}
   \]
   \[
   \text{num} = \text{num} - 3
   \]
   \[
   \text{End Repeat}
   \]
   \[
   \text{(A)} \quad 10 \ 7 \ 4
   \]
   \[
   \text{(B)} \quad 10 \ 7 \ 4 \ 1
   \]
   \[
   \text{(C)} \quad 10 \ 7 \ 4 \ 1 \ -2
   \]
   \[
   \text{(D)} \quad \text{This is an infinite loop.}
   \]
   \[
   \text{(E)} \quad \text{No output}
   \]

4. What will be the output of the following program?
   \[
   \text{num} = 10
   \]
   \[
   \text{Repeat Until} \ (\text{num} = 0)
   \]
   \[
   \text{Output num;}
   \]
   \[
   \text{num} = \text{num} - 3
   \]
   \[
   \text{End Repeat}
   \]
   \[
   \text{(A)} \quad 10 \ 7 \ 4
   \]
   \[
   \text{(B)} \quad 10 \ 7 \ 4 \ 1
   \]
   \[
   \text{(C)} \quad 10 \ 7 \ 4 \ 1 \ -2
   \]
   \[
   \text{(D)} \quad \text{This is an infinite loop.}
   \]
   \[
   \text{(E)} \quad \text{No output}
   \]

5. Which two sets of following code produce the same output?

<table>
<thead>
<tr>
<th>A)</th>
<th>B)</th>
<th>C)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{num} = 1 \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{Repeat Until} \ (\text{num} = 5) \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{Output &quot;Hello!&quot;} \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{num} = \text{num} + 1 \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{End Repeat}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{num} = 0 \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{Repeat Until} \ (\text{num} = 5) \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{Output &quot;Hello!&quot;} \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{num} = \text{num} + 1 \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{End Repeat}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{Repeat (5 times)} \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{Output &quot;Hello!&quot;} \</td>
<td></td>
<td></td>
</tr>
<tr>
<td>\text{End Repeat}</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6. Assume that \(i\) and \(last\) are integer variables. Describe precisely the output produced by the following segment for the inputs 4 and -2.

   Input \(last\)
   \[
   \text{i} = 0
   \]
   \[
   \text{Repeat Until} \ (\text{i} = \text{last})
   \]
   \[
   \text{Output i}
   \]
   \[
   \text{i} = \text{i} + 1
   \]
   \[
   \text{End Repeat} \]
7. The following is an infinite loop. Rearrange the statements so that the loop will terminate as intended.

\[
x = 0 \\
\text{Repeat Until (} x > 12 \text{)} \\
\text{Output } x \\
\text{End Repeat} \\
x = x + 2
\]

8. How many times will “Hi” be displayed when the following lines are executed?

\[
c = 12 \\
\text{Repeat Until (} c = 30 \text{)} \\
\quad \text{Output “Hi”} \\
\quad c = c + 3 \\
\text{End Repeat}
\]

(A) 5  
(B) 9  
(C) 6  
(D) 4  
(E) 10

9. What will be displayed by the following code?

\[
um = 7 \\
\text{Repeat Until (} \text{num > 8} \text{)} \\
\quad \text{Output num} \\
\quad \text{num = num + 1} \\
\text{End Repeat}
\]

(A) 7  
(B) 8  
(C) 7 8  
(D) 8 8

10. Which loop construct should be employed when the number of repetitions is known in advance?

(A) Repeat...End Repeat structure.  
(B) Repeat Until...End Repeat structure.
4. Solving Problems with Goal and Plans

In previous study, we have learned the basic instructions or commands in the programming and applied them to simple problems. However, for the more complicated problems, it is hard for beginners to put different pieces of code together. From this section, we are going to introduce a simple effective way to solve complex problems by using BYOB.

Early studies discovered that expert programmers built their program using stereotypical patterns of code. These stereotypical patterns of code are also called programming plans (or plans), code patterns, or templates. Plans are applied to achieve the goals of programming while goals are what must be accomplished to solve a problem. However, it is hard to identify the plans and understand the strategies that experts are using from typical programming textbooks, since they tend to focus on programming language features, rather than on the plans. We now briefly introduce the concepts of goals and plans and use a graphical notation to model them. Furthermore, we will learn a process for programming using goals and plans.

4.1 Goal

What is a goal?
A goal is a certain objective that a program must achieve in order to solve a problem.

We classify goals as being related to input, output, or processing. For example, when a program needs to display the sum of a sequence of values, the goal of displaying the result is classified as an output goal. This goal is associated with a variable such as sum, which holds the result.

However, before the result of sum can be displayed, it also needs to achieve the goal to input this sequence of values and the goal of processing these values to compute the sum. The goal to input the values is identified as input goal, which is associated with the variable Number for accepting values. Meanwhile, the goal for processing the values is called processing goal, which computes the value of the variable Sum, based on the value of Number. In general, a program will typically have at least one input, one output, and one processing goals. Three goal shapes together can build up one common shape of rectangle (see Figure 4-1).

Figure 4-1 Example of target rectangle made by three types of goals

For example, let us analyse the goals for displaying the sum of a sequence of values. In this case, we need to input a sequence of values first; and then to process the data flow of Number for Sum; and finally to achieve the goal of displaying the result of data flow Sum. This is done by the three goals: Input, Sum, and Output (see Figure 4-2). The Input Goal results in a sequence of input values on its port for output to next goal. This port is represented by the shape of multiple files with name “I:o” for the out port of Input Goal. From this out port a data flow Num, which is associate with the variable Number, moves to the in port of Sum Goal (S:i) for processing. After the Sum Goal combined these multiple values of “Num” into a single sum value and sent to its out port (S:o), which is represented by the shape of singular file, it generates a new data flow (Sum) to next goal. Finally, this single value of data flow “Sum” is
received the in port of Output Goal (O:i) for displaying. We also define test cases. For example, when the input values of Number are 1, 2, 3, the predicted result of Sum should be 6.

![Figure 4-2. Example of three basic goals](image)

### 4.2 Plan

**What is a plan?**

A plan is a code segment that accomplishes some programming goal.

Recall that goals are classified as being related to input, processing, or output. Similarly, we classify plans as being input plan, processing plan, or output plan. You can think of plans as being in a network, where data “flows” between the plans. A generic plan is shown as a box with double lines on both sides (see Figure 4-3). The data flows between the plans are represented by a variable, which may have either a singular value or a sequence of values. Data flow into a plan (flow-in) is from the left side and data flow out of the plan (flow-out) is on the right side. When using arrows to link the data flow from one plan to another, the result is a plan network to achieve the goals. Figure 4-4 shows a plan network diagram for the Sum problem in Figure 4-2. It demonstrates that the Input Plan generates an output sequence of values (Num) as a data flow, which becomes an input to the Sum Plan. The Sum Plan receives the data flow with a sequence of values (Num) and generates a single output value (Sum) of a new data flow, which is also the input to the Output Plan. Finally, the Output Plan receives the single value (Sum) of data flow and displays it.

![Figure 4-3. A generic plan](image)

![Figure 4-4. An example of plan network diagram](image)

**[Practice Problem 1]** For the Sum problem in Figure 4-4, assume a sequence of values (e.g. 1, 2, 3) has been input by the Input Plan. Fill in the table 4-1 by predicting the dataflow values through each plan.
Table 4-1. Predict the dataflow through each plan

<table>
<thead>
<tr>
<th>Plan</th>
<th>Port</th>
<th>Data Flow 1</th>
<th>Data Flow 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>out</td>
<td>1, 2, 3</td>
<td>2, 3, 7, 8</td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td>1, 2, 3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3 Plan Blocks

Using BYOB, a plan is developed as a **plan block** like a “command” block in the control block category (see Figure 4-5). However, unlike a normal block, a plan block has a group of blocks inside it. Each plan block has a unique name and parameter(s). The most important parameters are the **plan-ports**, which are used to define the network of plans, i.e. the links between plans.

**What is a parameter in the plan block?**
It is a variable that receives data from outside of the block.

**What is a plan-port?**
A **plan-port** is the interface of the plan to exchange dataflow with other plans.

For example, an Input Plan block (see the third block in Figure 4-5) is used for achieving the goal to input a sequence of values from the keyboard. It only has an out-port with the default name “values1:out” for sending these values to the next plan. The default port name is changeable and used to distinguish it from the output ports used in other Input blocks, e.g. “values2:out”, etc. Its second parameter “sentinel”, is used as an “end of input” marker and has a default value of “-1”. This means that when the user inputs the value “-1” the plan stops reading input and does not flow this value (-1) out to other plans. The value of “sentinel” can also be changed to any other value such as “99999”.

**Figure 4-5. A List of Plan Blocks in the Plan Library**

**What is a Plan Library?**
A group of plan blocks is also called a **Plan Library**.
Subsequently, a Process Plan block is needed for achieving the goal to process values. It generally has both an in-port and an out-port to receive data values from other plans and send results out to other plans after processing. For example, the “Sum Plan” (see the ninth block in Figure 4-5) has an in-port (sum:in) to receive the values and an out-port (sum:out) to send out its processed result (the Sum) to a subsequent plan.

**What is an in-port?**
A plan port for getting data flowing from other plan(s) is called an “in-port”. It can be seen as the left side of the generic plan box in Figure 4-3.

**What is an out-port?**
A plan port for sending data flowing to other plan(s) is called an “out-port”. It can be seen as the right side of the generic plan box in Figure 4-3.

Finally, opposite to the Input Plan block, Output Plan block (see the seventh block in Figure 4-5) only has an in-port to receive data values from another plan block for displaying them. The in-port has a default name “Output1:in”, which is also changeable and used to distinguish to another Output blocks, e.g. “Output2:in”, etc. To start using the plan blocks, a pre-made plan library must be added to the new project.

**The steps for adding the pre-made plan library to a new project:**
1) Start BYOB and click on the “Choose new sprite from file” button to open the New Sprite of “Library” provided (see Figure 4-6). After clicking "OK", a sprite of “fish” comes out.
2) Delete the new sprite of fish and click button “Control” to see a list of plan blocks (see Figure 4-5).

![Figure 4-6. Loading the Plan Library.](image)

[Practice Problem 2] In Figure 4-7, draw lines to link the ports of plan blocks, from out-port to in-port.

![Figure 4-7. Exercise of plan linkage by drawing lines](image)
5 A Programming Process Using Goals and Plans

What steps are used to develop programs using goals and plans?
1) Analysis of Goals: identify the goals to be achieved, and develop a dataflow map like Input-Process-Output.
2) Design Using Plan Blocks:
   a) For each goal, choose a plan block (from the "plan library" provided) that achieves that goal. (For more complex problems the plan blocks may need to be modified or created from scratch.)
   b) Use plan linkage blocks to link these plan blocks by the plan ports according to the data flowing among them.
   c) Run and test the design by comparing to the predicted answer.
3) Expanding Plan Blocks: replace plan blocks with the plan details found within the plan blocks, and test the details by comparing to the results from previous step.
4) Merging Plan Details: combine details of the different plans into one program, and test the merged program by comparing to the results from previous step.
5) Simplify the Program: remove all the scaffolding blocks, and test the final program by comparing to the results from previous step.

The following example is used to demonstrate the process of programming from goals and plans to the final program code.

Write a program that will read in integers and output their total value. Stop reading when the value -1 is input.

5.1 Step1: Devising Test Cases

What do you need to do in ? :
1) identify the output and input and specify inputs of test cases;
2) decide how to compute the output from input; and
3) calculate outputs from specified example inputs of test cases

1) identify the output and input and specify inputs of test cases
First of all, the output of a program can be identified from the requirements of the question. This is also the final goal being to achieved by the program. Next, the input of a program can be identified from the conditions and constraints provided in the question. Moreover, examples of input the test cases of input need to be specified in a format of testable data. For previous example:

Write a program that will read in integers and output their total value. Stop reading when the value -1 is input.

In this example, the output is the “total value” while the inputs are a sequence of integers. The integer “-1” is a sentinel indicating the last input, but is not processed. Inputs of test cases are specified at the beginning. An example of specified inputs for one test case is 1, 2, 3, and -1. Another test case is 3, 4, 5, 6, and -1. The following two examples are more complicated input-process-output cases.
2) Discover how to compute the output from input
The computation process is to calculate the sum of all the input numbers except the sentinel in order to get the output. For example, when the input data is 1, 2, 3, and -1, through the computation process of calculating a sum \((1+2+3=-6)\), the output result of total is 6.

3) calculate outputs from specified example inputs of test cases

**Predict answers: (fill your answer in the table)**

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Test Case(1)</th>
<th>Test Case(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Testing Data:</td>
<td>1, 2, 3, -1</td>
<td>3, 4, 5, 6, -1</td>
</tr>
<tr>
<td>Output Answers</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.2 Step2: Analyse goals and plans

1) What do you need to do in Analysis of Goals and Plans?
   a) Identify goals in the order of dataflow such as Input-Process-Output.
   b) Prepare the test case and predict the expected results.

Previously, we have studied that a program includes at least one of input, output, and processing goals. However, each goal can also be decomposed into several sub-goals that are associated to variables. The relationship between output and input variables determines the direction of dataflow for the processing, which also indicates the order of goals to achieve.

For example, in order to display the Sum of sequence values (see Figure 4-2), we need firstly to input the values by variables Numbers. According to the relationship between Number and Sum, we can then achieve the goal of process and then display the variable Sum. We will discuss the goals decomposition in the later contents. Meanwhile, we also define the test case. For example, when the input values are 1, 2, 3, a predicted result is 6.

[Practice Problem 3] When a program needs to display the count of a sequence of input values, what are the variables related? What are goals to be achieved? Draw a diagram of these goals. According to your goal diagram, map a plan network with ports to link the data flow. Prepare two test cases and predict the expected results.
1. Devise test cases:
   1) Identify the output and input and specify inputs of test cases;
   2) Discover how to compute the output from input;
   3) Calculate outputs from specified inputs.

2. Analyse goals and plans:
   1) Draw goal diagram, identifying goals and joining them by data-flow links;
   2) Map to plan network, identifying ports for each dataflow;
   3) Desk-check plan network.

   Are the results the same as those from previous phase?

3. Encode plan network using BYOB plan blocks:
   1) Apply plan blocks from library or build new ones;
   2) Link plan blocks using ports for each dataflow;
   3) Test.

   Are the results the same as those from previous phase?

4. Expand plan blocks and change parameters:
   1) Replace plan blocks by plan code details found inside each plan block;
   2) Replace every parameter of each plan with its plan port;
   3) If the same plan is used multiple times, the same variables in the different occurrences of the plan are renamed differently, e.g. `<name1>&<name2>`;
   4) Test.

   Are the results the same as those from previous phase?

5. Merge plan code details:
   1) Collect all the initialization blocks.
   2) Combine loops that share the same dataflow.
   3) Remove loop control if driven by a dataflow including only a single value;
   4) Test.

   Are the results the same as those from previous phase?

6. Simplify the Merged Details:
   1) If two variables share a dataflow to/from the same port, rename the second variable to be the same as the first one;
   2) Remove all the scaffolding blocks;
   3) Test.

   Are the results the same as those from previous phase?

Y: Continue
N: End
Exercises of Analysing Goals and Plans

1) Draw Goal Diagram: identifying goals and joining them by data flow.

![Figure 1. Example of three basic goals](image)

2) Map to Plan Network: identifying ports for each data flow.

![Figure 2. An example of plan network diagram](image)

3) Desk-check Plan Network. (Fill your answer in the table)

<table>
<thead>
<tr>
<th>Test Cases:</th>
<th>1, 3, 5</th>
<th>2, 4, 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Answers:</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>Port</th>
<th>Data Flow 1</th>
<th>Data Flow 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Are the results the same as predicted answers?

If your results are the same as the predicted answers, you have completed Step 1 of analysis. Before you move the next step, complete columns “Predicted Answers” and “Results from Analysis (Step 1)” in the following table. Complete the rest of columns when you work at the following steps. When filling this table, ensure the results from each step are the same as the previous step.
Table 2. Test schedule

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Predicted Answers</th>
<th>Results from Analysis (Step 1)</th>
<th>Results from Design by plan blocks (Step 2)</th>
<th>Results from expand plan details (Step 3)</th>
<th>Results from merged details (Step 4)</th>
<th>Results from final program (Step 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 3, 5, -1</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2, 4, 6, -1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.3 Step 3: Encode plan network using BYOB plan blocks

2) What do you need to do in encoding plan network using BYOB plan blocks?
   a) Identify plan blocks and develop a plan network diagram according to the order of goals.
   b) Map the plan network into the linkage by the plan-ports.
   c) Run and test the design.

According to the goal analysis, three plans are identified to achieve these three goals. These plans are 1) “Input Plan” for entering in a sequence of values, 2) “Sum Plan” for computing the total, and 3) “Output Plan” for displaying the result. According to the order of achieving these goals in Figure 4-2, a plan network diagram is developed as in Figure 4-4. By using BYOB, the plan network is implemented by a set of plan blocks (see Figure 5-1).

What is a scaffolding block?
A block is called scaffolding block, which helps either in the construction of plan blocks or in the linkage between the plan blocks, but it cannot be executed in a general environment without the support of plan library and must eventually be removed from the final program.

However, these plans blocks must be linked in order to transfer dataflow from one plan block to another. That is to say the plan network needs to be mapped into a collection of link statements. The scaffolding blocks have been created to do so, which can be found in the bottom part from the variable panel after clicking button “Variables” in BYOB (see Figure 5-2). We will firstly discuss the scaffolding blocks for linking plan blocks and then (in the next section) describe the scaffolding blocks for constructing new plan blocks.
Linking plan blocks is done by using “Link” blocks, which appear between a “Begin Links” and an “End Links” block. The “Link” block sets up the linkage between one plan and another plan through their ports. It presents a data flow between two plans. There are two parameters in this block. The first parameter is the out-port of a plan; the second parameter is the in-port of the next plan.

The plans blocks must be linked in the order of transferring dataflow from one plan block to another. The sequence of their linkage (see Figure 5-3) is: a “Begin Links” block, followed by one “Link” block to link from the out-port of “Input Plan” to the in-port of “Sum Plan” and another “Link” block to link from the out-port of “Sum Plan” to the in-port of “Output Plan”, followed by a single “End Links” block.

Figure 5-3. Mapping the plan network into the linkage

After setting the linkage between the plans, the plan network blocks are put after the linkage. Both parts together are the design of the program (see Figure 5-4).

Practice Problem 4] Use the plan library to implement the design of Figure 5-4 in BYOB and test it with the values 1, 3, 5, and -1. Answer the following questions.

a) Before starting, save the loaded plan library from Figure 4-6 named as PP4_Design in the folder with your name.
b) What is the value of Sum you expect to see from the plan network?
c) Run the plan network by clicking the flag. Does the testing result match your predicted answer a)? If not, why?
d) Fill in the following Table 5-1 by the dataflow. Compare the results to the predicted dataflow in Table 4-1. Find out whether or not you have got the same results at the ports “sum:in” and “output:in” in both Table 4-1 and Table 5-1.
Table 5-1. Trace the dataflow through plan blocks

<table>
<thead>
<tr>
<th>Test Cases:</th>
<th>1, 3, 5, -1</th>
<th>2, 4, 6, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Answers:</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>Port</th>
<th>Data Flow 1</th>
<th>Data Flow 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>out</td>
<td>(e.g.) 1, 3, 5</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>in</td>
<td>out</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Are the results the same as previous step?

*If the results are the same as those from previous step, “Save” your program as “Step 2” first and then “Save as” it to “Step 3” for next step.*

e) Save the completed program.

5.4 Step 4: Expanding Plan Blocks

3) **What do you need to do in Expanding Plan Blocks?**

*Replace each plan block by duplicating the plan details found inside the plan block.*

1) Replace plan blocks by plan code details found inside each plan block.

The expanding is quite simple and mechanical: we simply replace each plan block with the details inside it. The replacement order follows the plan block process order. For example, in Figure 5-4, the replacement starts from the Input Plan block by duplicating the Input Plan block details (see Figure 5-4 and Figure 5-6) from the “Block Editor” to replace the Input Plan block in the above plan network.

2) Replace every parameter of each plan by its plan port.

Since structural blocks (“GET DATA”, “SEND DATA”, and “NO MORE DATA?”) inside every plan block contains local parameters “in-port” and “out-port” to refer to ports of their hosting plan block, after expanding from a plan block, these parameters must be changed to the plan port name in order to work with the linkage by plan ports.

In other words, we need to replace variables “out-port” and “in-port” in the scaffolding blocks (see Figure 5-7) by entering the default value or coping the relevant port name from the plan block. In this example,

- Replace THREE “sentinel” by -1;
- For the details duplicated from “Input Plan”, replace ONE “out-port” by “values1:out” (through typing in or copying-and-pasting from the port of “Input Plan” plan block).
- For the details duplicated from “Sum Plan”, replace TWO “in-port” by “Sum:in” (through typing in or copying-and-pasting from the in port (left) of “Sum Plan” plan block); replace ONE “out-port” by “Sum:out” (through typing in or copying-and-pasting from the in port (right) of “Sum Plan” plan block);
- For the details duplicated from “Output Plan”, replace TWO “in-port” by “Output1:in” (through typing in or copying-and-pasting from the port of “Output Plan” plan block.)
Practice Problem 5] Continue the above Practice Problem 4 in Figure 5-4 to complete the following tasks:

a) Before starting, save the previous program named as PPS_Expanding.

b) Expand the plan blocks, “Sum Plan” and “Output Plan”.

c) Replace variables, “out-port” and “in-port”, by the port name of relevant plan block.

d) Test the expanded plan details by the same data as in the Practice Problem 4 to find out whether or not the results are also the same as those from the Practice Problem 4.

e) The result of expanded plan details is also executable and testable. Table 2 is continuously used to fill testing results in the column of Step 3.

f) Are the results the same as previous step?________________

g) If the results are the same as those from previous step, “Save” your program of Step 3 first and then “Save as” it to “Step 4” for next step.
5.5 Step 5: Merging Plan Details

4) What do you need to do in Merging Plan Details?
Combine the plan details of the different plans into one program.

From the expanded plan details, we can see that the commands/blocks which are dealing with the same dataflow come from different plan blocks. These blocks were designed for individual plan blocks rather than focusing on the whole dataflow of a program. To avoid duplicated blocks from different plan blocks, we need to remove the redundant blocks by combining or merging these details.
Practice Problem 6: Complete the following tasks:

a) Before starting, save the previous program named as PP6_Merging.

b) Draw arrows showing which blocks in Figure 5-8 correspond to which Figure 5-9 blocks.

c) Merge the program you have saved as Ex5_Expanding and compare your result to the solution in the program save as Fig 5_9.

de) Test the merged plan details (Figure 5-9) by the same data as in the Practice Problem 5 to find out whether or not the results are the same as those from Practice Problem 5.

e) Trace the dataflow and visualisation of variables as in question d) of Practice Problem 5. Describe the similarities and difference. Explain why.

f) Save the completed program.
Figure 5-8. The expanded plan details

Figure 5-9. The merged plan details

Please watch Video 4_Merging Details on the I: Drive

Figure 5-10. The Final Program
5.6 Step 6: Simplifying the Merged Details

5) What do you need to do in **Simplifying the Merged Details**?

   *Remove all the scaffolding blocks and the blocks containing the scaffolding blocks.*

Finally, we simplify the program by removing the scaffolding (grey) blocks from the “Begin of Link” to “End of Link”, as well as “SEND DATA”, “GET DATA” and “NO MORE DATA?”. The simplified program from the previously merged plan details is represented in Figure 2-10. Once again, the testing results from the simplified blocks are the same as those from the previous merged blocks. If the simplification is invalid, the feedback can be seen from the visualisation as well.
5.7 Summary

- Generally, a program has three types of goals to achieve. These goals are basically managed in a pattern of Input-Process-Output.
- In this tutorial, three types of plan blocks are applied to implement each of these goals in BYOB. Therefore, a plan network is a group of plan blocks to achieve of goals for a program.
- According to the dataflow, the plan network is linked by the scaffolding blocks through the plan ports. The plan network is also executable and testable to ensure the design is correct.
- A program is generated by expanding the plan blocks, merging the plan details, and simplifying the program.
- The process from problem to program is illustrated in Figure 5-11, and is described in Table 5-2.

![Diagram](image-url)
<table>
<thead>
<tr>
<th>Steps</th>
<th>Process</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Step 1: Devising Test Cases</strong></td>
<td>Predict answers from test cases without using computer program.</td>
<td>Predicted results</td>
</tr>
<tr>
<td><strong>Step 2: Analysis of Goals and Plans</strong></td>
<td>1) According to the data flow from Input, Process, to Output, analyse the goals and sub-goals and their relationships, and then draw a Goal Diagram. 2) Map the goal diagram to a Plan Network, adding ports for each plan, which link plans by data flow. 3) Desk-check Plan Network. Ensure the results are the same as the predicted ones.</td>
<td>A goal diagram, a plan network, and a desk check table.</td>
</tr>
<tr>
<td><strong>Step 3: Encode Plan Network Using BYOB Plan Blocks.</strong></td>
<td>According to the Plan Network, 1) Select relevant plan blocks from library or by building new ones to achieve every plan. 2) Use Link block to link every two plan blocks by using port names of starting and ending for each data flow. 3) Test and debug. Ensure the results are the same as the previous step.</td>
<td>An executable and testable plan network to achieve the diagram of goals.</td>
</tr>
<tr>
<td><strong>Step 4: Expanding Plan Blocks</strong></td>
<td>1) Replace plan blocks by plan code details found inside each plan block. 2) Replace every parameter of each plan (such as “in-port”, “out-port”, and “sentinel”) with its real name in the plan block (e.g. sum:in, sum:out, -1, etc.) and port. 3) If the same plan is used twice, the variable in each plan is renamed as &lt;name1&gt; and &lt;name2&gt; (e.g. when using two Input Plans, rename variable “Number” to “Number1” and “Number2” 4) Test and debug.</td>
<td>An executable and testable program which uses scaffolding blocks</td>
</tr>
<tr>
<td><strong>Step 5: Merging the Plan Details</strong></td>
<td>1) Put all the blocks relating to initialisation together after the “End of Link” block. 2) Put the loop bodies, which send or receive the same data, together based on their original plan order. 3) Remove the loop from the plan details if there is only one item sent from the previous plan. 4) Test the merged plan details and make sure the results are the same as those from the previous step.</td>
<td>An executable and testable program which contains but does not use scaffolding blocks</td>
</tr>
<tr>
<td><strong>Step 6: Simplifying the Merged Details</strong></td>
<td>1) If two variables share a data flow to/from the same port, rename the second variable to be the same as the first one. 2) Remove all the grey coloured scaffold blocks (including the blocks containing “scaffold” block) from the merged plan details. 3) Test the simplified program and make sure the results are the same as those from the previous step.</td>
<td>A final program without any scaffolding blocks.</td>
</tr>
</tbody>
</table>
Exercise V

1) Study the following question and complete the tasks.
Write a program that will read in integers and output the sums of both even and odd numbers. Stop reading when the value -1 is input.

   a) **Fill in variables in the following goal analysis, and draw a plan network diagram to achieve the goals.**

   ![Figure 5-12. The Analysis of Goals for the Sums of Even and Odd Numbers](image)

   b) **Start a new program to load the plan library from Figure 4-6 and save it and named as Ex5_7_1b. Select relevant plan blocks to implement the plan network in BYOB and save it again.**

c) **Map the plan network into the linkage by the plan-ports and test the initial solution by a set of test data. For example, when input 4, 2, 3, 1, 5, 2, 7, 6, and -1, it displays the sum of even numbers is 14 (=4+2+2+6) and sum of odd numbers is 16 (=3+1+5+7) as -1 is the sentinel. Save it again if there is any change from the above.**

d) **Save the previous result named as Ex5_7_1d_Expanding. Expand the plan blocks and use two different variables for the two applications of the same plan block, e.g. using variables “Sum1” and “Sum2” for the “Sum Plan” block in two sections. Test the expanded details by the same test data in the previous step to ensure the expanding is correct. Save it again.**

e) **Save the previous result named as Ex5_7_1e_Merging. Merge the expanded details and test it by the same test data again to ensure the merging is correct. Save it again.**

f) **Save the previous result named as Ex5_7_1f_Simplyfying. Remove all the scaffolding blocks to simplifying the merged program and test it again for the final program. Save it again.**

2) Complete the five steps of following question.

Write a program to repeatedly input a sequence of values and to display the ratio of Sum of odd numbers to Sum of even numbers. The program stops inputting when the value is -1.

(Note: The number is odd when the reminder of dividing by 2 is 1. i.e. “number mod 2 = 1”, e.g. 1, 3, 5, 7, etc. The number is even when the reminder of dividing by 2 is 0. i.e. “number mod 2 = 0”, e.g. 2, 4, 6, 8, etc.)
1. Predicting answers: *(Fill in the table by your predicated answers)*

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Predicted Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sum of Odd Numbers</td>
</tr>
<tr>
<td>1, 2, 3, 4, 5, 6, -1</td>
<td>9 (= 1 + 3 + 5)</td>
</tr>
<tr>
<td>1, 2, 3, 4, 5, 6, 7, 8, -1</td>
<td>16 (= 1 + 3 + 5 + 7)</td>
</tr>
</tbody>
</table>

2 Analysing Goals and Plans

1) Draw Goal Diagram: identifying goals and joining them by data flow.

2) Map to Plan Network: identifying ports for each data flow.


3) Desk-check Plan Network.
Table 1 Desk-check for dataflow in plan ports

<table>
<thead>
<tr>
<th>Test Cases:</th>
<th>1, 2, 3, 4, 5, 6, -1</th>
<th>1, 2, 3, 4, 5, 6, 7, 8, -1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predicted Answers:</td>
<td>0.75</td>
<td>0.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Plan</th>
<th>Port</th>
<th>Data Flow 1</th>
<th>Data Flow 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Plan</td>
<td>out</td>
<td>(e.g.) 1, 2, 3, 4, 5, 6</td>
<td></td>
</tr>
<tr>
<td>Odd Num Plan</td>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum1 Plan</td>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Even Plan</td>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum2 Plan</td>
<td>in</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Division Plan</td>
<td>in1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>out</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output Plan</td>
<td>in</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2 Testing Schedule

<table>
<thead>
<tr>
<th>Test Cases</th>
<th>Predicted Answers</th>
<th>Results from Analysis (Step 1)</th>
<th>Results from Design by plan blocks (Step 2)</th>
<th>Results from expand plan details (Step 3)</th>
<th>Results from merged details (Step 4)</th>
<th>Results from final program (Step 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4, 5, 6, -1</td>
<td>0.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1, 2, 3, 4, 5, 6, 7, 8, -1</td>
<td>0.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Are the results the same as predicted answers?

3 Designing a Plan Network by Plan Blocks
1) Apply plan blocks from library. (see the competition questions in BYOB)
2) Link plan blocks by ports for each data flow.
3) Test

Are the results the same as previous step?

If the results are the same as those from previous step, “Save” your program as “Odd Num Step 3” first and then “Save as” it to “Odd Num Step 4” for next step.

4. Expand Plan Blocks:

1) Replace plan blocks by plan code details found inside each plan block.

2) Replace every parameter of each plan by its plan port.

3) Test and debug

Are the results the same as previous step?

If the results are the same as those from previous step, “Save” your program of Step 4 first and then “Save as” it to “Odd Num Step 5” for next step.

5. Merge Plan Code Details:

1) Collect all the initialization blocks.

2) Combine loops which share the same data flow.
(Note: merge the loops for both Odd Number Plan and Sum Plan first, and then merge loops for Input Plan and the new merged loop.)

3) Remove the loop control if it is driven by a dataflow including only a single value

4) Test and debug

Are the results the same as previous step?

If the results are the same as those from previous step, “Save” your program of Step 5 first and then “Save as” it to “Odd Num Step 6” for next step.

6. Simplify the Merged Details:

1) If two variables share a data flow to/from the same port, rename the second variable as the first one.

2) Remove all the scaffolding and structural blocks.

3) Test and debug

Are the results the same as previous step?

If the results are the same as those from previous step, “Save” your program of Step 6 and Finish the Exercise.

2) Write a program that will read in integers and output their sum and count values. Stop reading when the value -1 is input. For example, after input 1, 2, 3, and -1, it displays Sum as 6(=1+2+3), and Count as 3 as -1 is the sentinel. Save every intermediate result as Ex_5_7_2_Design, Ex_5_7_2_Expanding, Ex_5_7_2_Merging, and Ex_5_7_2_Simplyfing.
3) Write a program to calculate and output wages for each member of a group of people. Input the pay rates and working hours for each person. The formula for wages is wages = rates * hours. It stops when the rates or hours is -1. Save every intermediate result as Ex_5_7_3_Design, Ex_5_7_3_Expanding, Ex_5_7_3_Merging, and Ex_5_7_3_Simplyfing.

4) Write a sales program that will input the price and amount of a serial of sales. The formula of sale for each item is: sales = price * amount. Stop input wages when the value -1 is input. The program outputs the sales for each item and sum of sales for all the items. Save every intermediate result as Ex_5_7_4_Design, Ex_5_7_4_Expanding, Ex_5_7_4_Merging, and Ex_5_7_4_Simplyfing.

6. BYOP (Build Your Own Plans)

You have studied how to achieve the goals of a program using existing plan blocks in BYOB. In this part, you will learn how to build up your own plan blocks in order to deal with situations where the plan library is not sufficient. Since the Input and Output plan blocks can be used in most situations, the following contents only describe how to build Process Plan blocks.

6.1 Overview of Building Process Plan Blocks

What does an interface of a Process Plan block look like?
The interface includes both in-port and out-port, which have default values of “<plan name>:in” and “<plan name>:out”.

A Process Plan block needs an interface and contents. Under certain patterns, the contents are defined by using BYOB control blocks and the scaffolding blocks.

What kinds of blocks are needed inside a Process Plan block?
At least five of the following blocks are needed.

6.2 Scaffolding Blocks for Building Plans

There are three scaffolding blocks that support building plan blocks (see Figure 6-1). They are “GET DATA”, “NO MORE DATA?”, and “SEND DATA” blocks. The “SEND DATA” block sends data to a given output port, “GET DATA” gets data from an input port, and “NO MORE DATA?” is used to check whether there is any more data on an input port.

Figure 6-1. The Scaffolding Blocks for Building Plan Blocks
6.3 Defining New Plan Blocks Using a Process Pattern

The internal structure of a Process Plan block is defined as a certain typical pattern such as Table 6-1, which consists of a sequence of regular tasks.

Table 6-1. The Pattern of Process Plan to Produce a Single Data-Flow-Out

<table>
<thead>
<tr>
<th>Plan Name (in-port = plan:in), (out-port = plan:out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialise variable</td>
</tr>
<tr>
<td>repeat until &lt; NO MORE DATA? (in-port) &gt;</td>
</tr>
<tr>
<td>set (variable) to (GET DATA (in-port))</td>
</tr>
<tr>
<td>Process variable according to the logic</td>
</tr>
<tr>
<td>SEND DATA (variable) (out-port)</td>
</tr>
</tbody>
</table>

The following example is shown how to build the Count Plan block (see the first block in Figure 4-5). The Count Plan has two ports, “count:in” and “count:out”. The plan details (see Figure 6-2) show the pattern of how the data is repeatedly gotten from the “in-port” and then counted. At the end, the processed result is sent to the “out-port”.

The steps of creating a Count Plan are: (if the “Count Plan” block already exists in the “Control” block palette as in Figure 4-5, remove it by selecting “delete block definition” item when right-clicking on it.)

1) Click the command button “Variable” in the BYOB (see Figure 6-3) to see the panel of making a variable and blocks. Click button “Make a block” to see the Window “Make a block”. Then select “Control” under the “category” and see “command” type is highlighted. Type the plan name, “Count Plan”, and then click “OK” to see a new Window “Block Editor”.

2) Inside the “Block Editor”, click the plus sign “+” after the plan name “Count Plan” to see a new Window “?” (see Figure 6-4). Click the triangle sign after the “Input name” to expand the Window “?” as Figure 6-5. Under “Create input name”, type “in-port” for the first parameter of the plan block. Also select “Text” as its “Input type”. Then type “count:in” as its “Default value”. Click “OK” to complete the setting of in-port of the plan block.

3) Similarly, click the plus sign “+” at the end of the plan title to set up the out-port. Repeat step 2) to type “out-port” for the second parameter with the default value “count:out”.

4) If the variable “Count” does not exist in your BYOB, create it by clicking the button “Make a variable” in the “Variables” panel and then type the variable name “Count” and click “OK”.

5) Drag and drop the blocks into the “Block Editor” as Figure 6-2. And then click “OK”. Now the “Count Plan” block is ready for further application.
6.4 Other Patterns of Process Plan Blocks

When a Process Plan block produces a *sequence* of values to its out-port and each of the output value is directly related to every value from the in-port, the process of building the plan is similar to the above process, but the last step must be put inside the loop body (see Table 6-2). For example, to produce even numbers from a sequence of values, the Even Number Plan block is created as Figure 6-6.

<table>
<thead>
<tr>
<th>Plan Name (in-port = plan:in), (out-port = plan:out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialise variable</td>
</tr>
<tr>
<td><strong>repeat until</strong> &lt; NO MORE DATA? (in-port) &gt;</td>
</tr>
<tr>
<td><strong>set</strong> (Number) <strong>to</strong> (GET DATA (in-port))</td>
</tr>
<tr>
<td>Process (Number) to (variable) according to the logic</td>
</tr>
<tr>
<td><strong>SEND DATA</strong> (variable) (out-port)</td>
</tr>
</tbody>
</table>

However, some Process Plans have multiple in-ports to receive data from different plans. For example, For example, the Multiplying Plan block (see Figure 6-7) has two in-ports to receive two sequences of items, item1 and item2. It multiplies every two items, item1 * item2, and sends a sequence of results to its out-port. Its plan detail is shown in Figure 6-8. A plan produces a sequence of values to its out-port and each of the values is directly related to the
value from its multiple in-ports, the structure of building its plan block by multiple “set” and “get” is shown as Table 6-3.

**Figure 6-7. The Multiplying Plan block**

**Figure 6-8. The plan details of Multiplying Plan**

**Table 6-3. The Pattern of Process Plan to Produce Multiple Dataflow-Out from Multiple in-ports**

<table>
<thead>
<tr>
<th>Plan Name (in-port1 = plan:in.item1) (in-port2 = plan:in.item2) (in-port3=...) (out-port = plan:out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialise variables</td>
</tr>
<tr>
<td>repeat until &lt; NO MORE DATA? (in-port1) &gt; or &lt; NO MORE DATA? (in-port2) &gt;or &lt;...&gt;</td>
</tr>
<tr>
<td>set (variable1) to (GET DATA (in-port1))</td>
</tr>
<tr>
<td>set (variable2) to (GET DATA (in-port2))</td>
</tr>
<tr>
<td>set ....</td>
</tr>
<tr>
<td>Process variables according to the algorithm</td>
</tr>
<tr>
<td>variable = f(variable1, variable2, ...)</td>
</tr>
<tr>
<td>SEND DATA (variable) (out-port)</td>
</tr>
</tbody>
</table>

6.5 Summary

- A new plan block can be built by using the three types of “scaffold” blocks.
- The Process Plan block needs at least two parameters as its ports.
- The ports are normally named as “<plan name>:in” and “<plan name>:out”.
- A plan block always utilises a loop to continuously get a sequence of values from its in-port(s) after initialising the applied variables. It then follows the general pattern of input-process-output to organise the blocks in the loop body.
- The step of sending values to the out-port can be after the loop or inside the loop, which depends on the singular or multiple values of the dataflow to the out-port.

6.6 Exercise VI

1) According to the following table (see Table 6-4), create a Times Plan block to multiply each element of a sequence of values from its In-Port. For example, when the values in the in-port are 5, 2, 3, the processing is $5 \times 2 \times 3 = 30$. Finally, the processed result (30) is sent to the out-port.
Table 6-4. The structure of Times Plan block

<table>
<thead>
<tr>
<th>Times Plan (in-port = plan:in), (out-port = plan:out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\underline{Result} = 1</td>
</tr>
<tr>
<td>repeat until &lt; NO MORE DATA? \textcolor{red}{(in-port)} &gt;</td>
</tr>
<tr>
<td>set \textcolor{red}{(Number)} to ( \textcolor{blue}{GET DATA (in-port))}</td>
</tr>
<tr>
<td>\underline{Result} = Result * \textcolor{red}{Number}</td>
</tr>
<tr>
<td>\textcolor{blue}{SEND DATA (Result) (out-port)}</td>
</tr>
</tbody>
</table>

Create a plan network to test this new plan block by using Input and Output plan blocks. Expand, merge, and simplify the initial solution to a final program. Save every intermediate result as Ex6_1_Design, Ex6_1_Expanding, Ex6_1_Merging, and Ex6_1_Simplyfing.

2) Create a plan to generate a sequence of positive integers (greater than zero) to its out-port after retrieving a sequence of integers from its in-port. For example, when this plan retrieving data, 2, -5, 3, from its in-port, it will send values of 2, 3 to its out-port.

Table 6-5. The structure of Positive Number Plan block

<table>
<thead>
<tr>
<th>Positive Number Plan (in-port = plan:in), (out-port = plan:out)</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat until &lt; NO MORE DATA? \textcolor{red}{(in-port)} &gt;</td>
</tr>
<tr>
<td>set \textcolor{red}{(Number)} to ( \textcolor{blue}{GET DATA (in-port))}</td>
</tr>
<tr>
<td>If \textcolor{red}{Number} &gt; 0</td>
</tr>
<tr>
<td>\underline{Result} = \textcolor{red}{Number}</td>
</tr>
<tr>
<td>\textcolor{blue}{SEND DATA (Result) (out-port)}</td>
</tr>
</tbody>
</table>

Write a program to test this new plan and output the sum of the positive integers.

3) Create a plan to calculate the square value for each of the value from its In-Port and send results to its Out-Port. For example, when this plan retrieving data, 2, 5, 3, from its in-port, it will send values of 4, 25, 9 to its out-port. Write a program to test this new plan.

4) Write a program that will read in integers and output their maximum, minimum and Sum values. Stop reading when the value -1 is input.
6.7 Have Fun with Goals and Plans

1) Write a program to let user guess a secrete number created by computer. When the input number is less than the secret number, it asks the user to enter a larger number (e.g. “Your guess is too low”). Otherwise, it asks the user to enter a lower number (e.g. “Your guess is too high”).

   a) Create a Number Plan to generate the secrete number by a random number. This sends a random number within a certain range (e.g. from 1 to 10) by its out port.

   ![Number Plan Diagram]

   b) Create a Guess Plan to ask user input a sequence of numbers until it matches the number at its in port. In this plan, the loop continues to ask for the guess number. It stops only if the guess number is the same as the secrete number or the user gives up by entering -1. Finally, the plan sends the secrete number out. See the structure:

   ![Guess Plan Diagram]
c) Test both Number Plan and Guess Plan by the following program.

```
when clicked
Begin of Link
Link number:out to guess:in
Link guess:out to output1:in
End of Link
Number Plan from 1 to 10 number:out
Guess Plan guess:in guess:out sentinel -1
Output Plan output1:in
```

d) Modify the Guess Plan Block so that when the input number is less than the secret number, it shows “Your guess is too low”. Otherwise, it shows “Your guess is too high”.

```
Guess Plan2 guess:in guess:out -1
```

```
set Result to GET DATA in port
ask join Enter a number, stop when number is sentinel and wait
set Number to answer
repeat until Number = Result or Number = sentinel
    if Number < Result
        say Too low! for 2 secs
    else
        say Too high! for 2 secs
    ask join Enter a number, stop when number is sentinel and wait
    set Number to answer
SEND DATA Result out-port
```

e) Test the modified Guess Plan2 block by replacing the Guess Plan block in the program c).
2) Write a “Math Teacher Program” to teach pupil adding. It generates two random numbers (e.g. 5 and 7). Ask user to answer the sum of these two numbers (e.g. 12 (=5 + 7)).
   a) Create an Add Plan to sum two numbers.
   
   b) Use the new plan blocks created in the above exercises to complete the program as follows:

   c) Go through the process of Expanding, Merging, and Simplifying to get the final BYOB program and test them.