
THE UNIVERSITY of ADELAIDE

# Codification Pedagogy for Introductory Procedural Programming Courses

Rita Alicia GARCIA

A thesis submitted for the degree of
DOCTOR OF PHILOSOPHY
The University of Adelaide

March 5, 2020

# Contents

# List of Figures

# List of Tables

University of Adelaide

# *Abstract*

**Codification Pedagogy for Introductory Procedural Programming Courses**

by Rita Alicia GARCIA

Generally, students in introductory programming courses (CS1) do not devote time to designing solutions to their programming problems, even though it is a necessary part of the problem-solving process. Without the design process to reflect on a problem, students might haphazardly solve them, but with incomplete solutions. Students might skip the design process because they have limited design knowledge and lack the skills to help them identify goals and create a plan for solving a problem. Students might also ignore problem-solving information provided to them and instead rely on past problem-solving approaches, which keeps them from learning both new problem-solving strategies and new programming concepts.

This research explores a pedagogical approach for procedural programming assignments facilitated within an online learning environment that encourages CS1 students to incorporate the design process into their problem-solving process. This thesis refers to the pedagogical approach as the *Codification Pedagogy*, a teaching approach for ordering rules corresponding to a plan. The pedagogy is designed to help students identify goals and create plans for solving problems. The pedagogy is comprised of three learning activities:

1. A scaffolded assignment presentation designed to help students better understand the programming problem. The assignment presentation helps students identify the problem's goals and provides additional support for struggling students. This research produces a framework that educators can use to develop scaffolded presentations for CS1 programming assignments.

2. A questioning activity that encourages students to engage their internal knowledge to solve the current problem. This research produces a questioning framework. The framework contains instructional questions mapped to the Bloom's Taxonomy cognitive levels. The framework can help educators construct learning activities through questioning to help elevate students' cognitive level appropriate for their learning.

3. A Parsons problems activity designed to help students organise an implementation plan. Parsons problems is a learning tool that has students arrange code fragments to form a working program. The research demonstrates that Parsons problems can be used to help students organise plans to solve programming problems.

The Codification Pedagogy is integrated into CS1 programming assignments. Studies were conducted for three semesters in an introductory programming course offered at the University of Adelaide. The research comprises quantitative studies using interactive analytics and variable-oriented analysis, along with qualitative studies using mixed methods that include pre-post tests, think-alouds, and interview sessions.

The pedagogy is designed to help students better understand the programming problem and support their learning of problem-solving strategies for practical programming assignments. The results from this thesis demonstrates the pedagogy can support students during the design process. The studies presented in this thesis shows the pedagogy supporting students' use of problem-solving strategies that help them to identify goals for the problems and enable them to validate their programming solutions. The results also show the learning activities encouraging students to analyse the assignment, promoting self-reflection that reduce misconceptions. Through its design-based support, the pedagogy can support students to successfully complete programming assignments.

# Declaration of Authorship

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I acknowledge that copyright of published works contained within this thesis resides with the copyright holder(s) of those works.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library Search and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

I acknowledge the support I have received for my research through the provision of an Australian Government Research Training Program Scholarship.

Rita Alicia Garcia

November 2019

# *Acknowledgements*

I want to thank my partner Tony Horstman for being so supportive and patient throughout this journey. Thank you to my wordsmith Jon Hertzig who tirelessly answered my grammar questions. I learned so much from his talents. Thanks to all my friends and family, especially to Ron Howie and Melinda Jennings, who listened to my ramblings about the research.

Thank you to Alicia Zakarevicius and Wanru Gao who administered the course with my research. I am grateful for Alicia setting up the online courses and finding inconsistencies in my assignment designs. Working with Alicia really elevated the quality of the research. Thank you to the team who developed the js-parsons library: Ville Karavirta, Petri Ihantola, and Juha Helminen. The comprehensive documentation and interface made it very easy to apply the Parsons problems in a novel way needed for this research. Thanks to the University of Adelaide Computer Science Education Research (CSER) group for the discussions and information sharing. The conversations have been an inspiration.

Thank you to my manager at Method Studios, Vincent Dedun, and my co-workers, especially Eva Lu and Rebecca Bever, who were supportive through my academic journey and giving me the opportunity in continuing to contribute to the film industry during this journey.

Thank you to my co-advisor, Rebecca Vivian, whose guidance and expertise in data analysis really helped my skills in this area. Lastly, thank you to my supervisor Katrina Falkner. Her experience and guidance has made me a better educator and researcher. I will carry the skills and wisdom from both advisors in my future endeavors.

# Chapter 1

# Introduction

Designing a solution to a programming problem is a necessary part of the CS problem-solving process (Hoadley and Cox, 2009) However, many students in introductory programming courses (CS1) do not always devote time to designing solutions to their programming problems (Pintrich, Berger, and Stemmer, 1987; Rist, 1995). The design process is a heuristic composed of task analysis, planning, and evaluating, which requires practice and repetition from the learner to acquire (Ginat, 2008). Students new to CS have not yet acquired skills used in the design process that can help them take a systematic and methodical approach to successfully solve problems (Bishop-Clark, 1995). Instead, novices engage in haphazard approaches to the software development process, choosing to skip the design process and not consider their design decisions when making changes to their programming problems (Chmiel and Loui, 2003).

With limited or fragile knowledge about the design process, novices have been observed using Poor Learning Tendencies (PLTs) (Baird and Northfield, 1995) when previous problem-solving approaches fail, leading to frustration (Kinnunen and Simon, 2010). Poor Learning Tendencies are behaviours that promote impulsive actions that lack reflective thinking and instead encourage students to focus on surface aspects of the problem, such as using lexical analysis instead of trying to understand the problem at the semantic level (Adelson, 1984). Problem solving using Poor Learning Tendencies can result in misunderstandings when reading the problem description (Collins and Stevens, 1983).

To help students avoid using Poor Learning Tendencies, and instead build their design knowledge and skills, educators can model problem-solving strategies that can help them become expert programmers (Mead et al., 2006) and progress further in their CS studies (Robins, Rountree, and Rountree, 2003). Teaching design-based problem-solving approaches can help students identify goals and create plans for solving problems, reducing the surface-level approaches to their learning (Joughin, 2010) and encouraging a deeper understanding of the programming problem (Spoher and Soloway, 1989). Without a deeper understanding of the problem, students might form flawed mental representations of the solution, making it difficult for them to develop correct learning concepts (diSessa, 2014). Teaching design-based problem-solving approaches to students can help them integrate the design process into their software development processes. When students integrate the design process into their software development process, they take a more systematic approach to solving problems and consider their changes prior to implementation (Bishop-Clark, 1995). Taking a more considered approach to problem solving and reflecting on design decisions are traits found in students who perform well in CS1 (Bergin, Reilly, and Traynor, 2005).

This thesis focuses on helping students adopt design-based problem-solving approaches early in their CS1 studies, to discourage Poor Learning Tendencies. This

thesis has students model experts' problem solving through a pedagogical approach. The pedagogical approach supports CS1 students practising problem-solving approaches pertaining to planning and organising a programming solution, and is embedded within programming assignments. The pedagogy is designed to help students better understand the problem and support their learning of problem-solving strategies for practical procedural programming assignments, an activity with minimal supervision that focuses on the procedural programming paradigm. In this thesis, the term *understanding* is defined as 'the ability to follow instructions successfully and readily' (Marcus, Cooper, and Sweller, 1996). The pedagogy is administered in a CS1 course at the University of Adelaide, where the cohort size is between 100-250 students and supported by one lecturer and two tutors.

This thesis presents the development and evaluation of the *Codification Pedagogy*, a pedagogical approach designed to help students identify goals and organise into a corresponding plan. *Codification* is a legal term for arranging laws according to a plan (Villiger, 1985) and is applicable to Software Development. Codification can describe the processes that occurs during the design phase, where a person orders program requirements into an organised plan prior to implementing a solution. This thesis explores the extent to which the pedagogy can help improve CS1 students' program comprehension, critical thinking, and design knowledge skills. The pedagogy is grounded in *Cognitive Apprenticeship*, a model of instruction designed to bring reasoning and strategies to the forefront during the problem-solving process (Collins, Brown, and Holum, 1991). Cognitive Apprenticeship is based on the practice of apprenticeship that demonstrates to the learner how to perform the task and assists them through the process. The Codification Pedagogy models experts' behaviour, to promote higher learning gains and encourage the adoption of this behavior.

This thesis investigates methods used by learners during the problem-solving process. Methods include *Self-Regulated Learning (SRL)* strategies, strategies that can help students 'plan, set goals, organise, self-monitor, and self-evaluate' (Zimmerman, 1989). The pedagogical approach is designed to guide the students through the problem-solving process (Mayer, Richard, 1983), to help them form effective problem-solving approaches, through the use of SRL strategies. The pedagogy encourages the learning of problem-solving approaches by providing students with different activities that promote the use of SRL (Zimmerman, 1989) and design-based strategies (Détienne, 2002), strategies used during the problem-solving process for planning and setting goals (Greeno and Simon, 1988).

One part of the pedagogy is the assignment presentation, designed to help improve students' understanding of the instructional materials. The understanding of the problem description is part of the concept known as *Program Comprehension*, the process of forming a mental representation of a solution by assimilating the text-based representation of the problem to associate with their current knowledgebase using cognitive strategies (Schulte et al., 2010). This thesis investigates the presentation of the programming problem to help students better understand the problem. The assignment presentations designed in this thesis are contained in a learning environment that promotes better understanding of the problems, to promote success in identifying programming tasks. As the student progresses in their learning and builds experience, the support within the learning environment is slowly removed to encourage the independent practice of problem-solving skills (Ghefaili, 2003).

Based on a review of research literature in critical thinking, this pedagogy supports the use of critical thinking skills through a questioning activity that contains questions related to the problem's tasks (Roll et al., 2007). *Critical thinking* is thinking

that attempts to form correct judgement about the problem (Le and Huse, 2016). Interacting with the learning activity can encourage a deeper analysis of the problem, which can reduce students' misconceptions about the problem (Weusijana, Reisbeck, and Jr, 2004).

The pedagogy is also designed to help build students' *design knowledge*, knowledge that helps students plan and set goals (Greeno and Simon, 1988). To support students' design knowledge, pedagogy integrates Parsons problems (Parsons and Haden, 2006) to demonstrate how to construct an organised plan. Parsons problems is a learning tool that has students arranging code fragments to form a working problem. Rather than presenting code fragments, the Parsons problems used in this thesis presents design plans for students to organise.

## 1.1 Areas of Research

This section describes the areas of research explored by this thesis. These areas are program comprehension (Section 1.1.1), critical thinking skills (Section 1.1.2), and design knowledge (Section 1.1.3).

### 1.1.1 Program Comprehension

This thesis investigates an approach for improving students' *program comprehension*, the process of forming a better mental representation of the solution when assimilating the problem description with their internal knowledge (Schulte et al., 2010). The pedagogical approach attempts to improve students' program comprehension when solving procedural programming problems. The study focusing on program comprehension evaluates how the presentation for the programming assignments can help students better understand the problem. Through a better understanding of the problem description, students might be able to successfully develop and complete the assignment. When assignments are presented with ill-defined goals, more mental effort is required from the student to understand the problem (Reitman, 1965). The thesis examines approaches to minimising ill-defined goals, by identifying design treatments that can help construct well-formed programming assignments that positively impact students' understanding (Schulte et al., 2010). The well-formed assignments can help students complete the programming problem (Feldman and Zelenski, 1996). This thesis looks at how constructing well-formed assignments with these treatments can help improve students' program comprehension.

### 1.1.2 Critical Thinking Skills

This thesis integrates critical thinking skills into the pedagogical approach. *Critical thinking skills* promote 'thinking explicitly aimed at well-formed judgment, utilising appropriate evaluative standards in an attempt to determine the true worth, merit, or value of something' (Paul and Elder, 2007) and engage students' internal knowledge to have them reflect on the problem prior to solving a problem (Bloom, 1956). This thesis explores the use of a questioning activity as an intervention for CS1 procedural programming assignments. The purpose of the questioning activity is to encourage critical thinking skills. Questioning is a learning activity that requires thinking processes to answer the question (Bloom, 1956). Effective questions can promote good learning behaviours that encourage self-reflection during the design process and support the students' comprehension of programming assignments, helping them deconstruct problems and develop plans (Boyer et al., 2010).

The questioning activity is designed to encourage self reflection and the use of Self-Regulated Learning (SRL) strategies (Zimmerman, 1989), such as seeking out information needed to solve the problem. To construct the questioning activity, this study uses instructional questions previously applied in a CS learning environment (Boyer et al., 2010). These 23 instructional question types form a framework that identifies the appropriate question for the students' learning abilities. The framework was formed by mapping the instructional questions to cognitive levels within Bloom's Taxonomy, a classification of critical thinking skills required for cognition (Bloom, 1956).

### 1.1.3 Design Knowledge

Novice programmers either do not devote enough time (Pintrich, Berger, and Stemmer, 1987) or completely neglect (Rist, 1995) the design process. Novices might avoid the design process because of their fragile *design knowledge*, which makes it difficult for them to identify plans and goals (Robins, Rountree, and Rountree, 2003). Research (Prather et al., 2019) has shown that when students are provided with a design-based intervention, they have a higher degree of completing programming assignments. Prior research (Détienne, 2002) has shown CS1 students misapplying design strategies, components in the cognitive activity used during the problem-solving process for planning and setting goals (Greeno and Simon, 1988). Design-based interventions can help students apply correctly to help them build their design knowledge.

This thesis looks at helping students develop a decomposition problem-solving strategy (Gick, 1986) that is invoked by the problem schema. The decomposition approach has the student deconstruct requirements from problem's specifications and map them into programming language constructs. The pedagogical approach developed in this thesis adopts the problem-decomposition problem-solving strategy to demonstrate to students how to solve programming problems. When using the problem decomposition problem-solving strategy, the software implementation order is determined by the dependencies between goals and tasks.

This thesis supports novices during the design process by helping them analyse and decompose problems. This thesis supports the learning of design strategies by using Parsons problems as an intervention within procedural programming assignments.

## 1.2 Research Questions

This thesis focuses on helping CS1 students develop design-based problem-solving skills through a pedagogical approach. The pedagogical approach demonstrates how to apply these skills in practical programming assignment by modeling problem-solving approaches. The pedagogy is comprised of three learning activities that work together to investigate problem-solving skills used during the design process. These learning activities are a scaffold assignment presentation designed to improve students' program comprehension, a questioning activity that promotes the use of critical thinking skills, and a Parsons problem used to support the learning of design knowledge.

Five research questions have been posed to evaluate the pedagogical goals. These five questions are divided into the three areas of research shown in Table 1.1. The research questions are addressed in Chapters 5 through 10. Table 1.1 also provides the

| Research Questions | Hypothesis | Data Source |
|---|---|---|
| **Program Comprehension**<br>**RQ1.1:** How does scaffolding the assignment presentation influence the student's ability to identify goals and subgoals necessary to complete a procedural programming assignment?<br><br>**RQ1.2:** What presentation treatments within the programming assignments support students in their understanding of the problem? | **H1.1:** Scaffolding the assignment presentation will guide students in identifying the problem's goals and subgoals, providing them support in identifying a starting point for developing a programming solution.<br>**H1.2:** Itemising goals and subgoals as a treatment in the assignment presentation will help students identify and apply the goals and subgoals during the programming process. | • Interviews<br>• Questionnaires<br>• Surveys<br>• Overall programming solutions grades<br>• Generated coded solutions |
| **Critical Thinking Skills**<br>**RQ2.1:** Does encouraging questions in an online CS1 learning environment promote the expected cognitive levels from students when answering the questions? | **H2.1:** Providing a questioning activity as an intervention to a programming assignment will help students internally reflect on and apply their knowledge when solving the current problem. | • Answers to questions<br>• Survey responses |
| **Design Knowledge**<br>**RQ3.1:** How do students use Parsons problems during the design process for solving CS1 procedural programming assignments?<br><br>**RQ3.2:** What Self-Regulated Learning (SRL) strategies are supported by Parsons problems used as a design-based intervention for programming assignments. | **H3.1:** Parsons problems will promote internal reflection, enabling the student to form a mental model on how the problem's plans interact with each other prior to developing a programming solution.<br>**H3.2:** The Parsons problem will promote organisation and planning SRL strategies that are necessary to solve the programming problem. | • Parsons problem solutions<br>• Audio-visual material<br>• Interview<br>• Metacognitive Awareness Inventory Instrument<br>• Questionnaire |

TABLE 1.1. Overview of Thesis Questions, Hypotheses, and Data Sources

hypothesis for each question and a brief description of the data collected to help answer the questions. The table shows a blend of quantitative and qualitative studies used to collect the data.

The research was conducted over three semesters using different quantitative approaches, such as interactive analytics (Turkay et al., 2017) and variable-oriented analysis (Bergman and Trost, 2006). Metacognitive Awareness Inventory (MAI) (Schraw and Dennison, 1994) was another quantitative instrument used in the study to measure skill gains through the use of the pedagogy. Metacognitive Awareness Inventory has been shown in the field of Education to reliably measure skill growth (Akin

and Abaci, 2007).

The research incorporated qualitative studies with volunteers from the introductory programming course offered at the University of Adelaide. The qualitative studies used a mixed-methods approach of pre-tests, surveys, questionnaires, think-alouds, and interview sessions, and were conducted in a lab environment, where students solved worked examples, completed practical programming assignments, and received assistance from tutors.

## 1.3 Summary of Contributions

This section provides a summary of contributions made by this thesis. Table 1.2 provides an overview of these contributions to the field of Computer Science Education. The table presents the contributions in three areas of research: program comprehension, critical thinking skills, and design knowledge.

The first contribution is towards improving program comprehension through the assignment design for procedural programming assignments. This thesis looks at presentation treatments that can help students better understand the programming problem and develop mental representations of the solution. The outcome from this study is the creation of a CS1 assignment design framework designed to help educators develop assignments that promote better understanding of the problem and encourage students to use problem-solving skills appropriate for their learning abilities. Another contribution in the area of program comprehension is the identification

| **Contribution** | **Description** |
|---|---|
| Program Comprehension | • Provides a CS1 assignment design presentation framework to assist in the scaffolding of assignment descriptions.<br>• Finds that presenting goals in list format helps students better identify the problem requirements and evaluate solving the problem at higher cognitive levels. |
| Critical Thinking Skills | • A framework to help educators construct questioning activities that will help students practise critical thinking skills appropriate to their cognitive levels.<br>• Demonstrates that the combination of lower and higher-order questions in a learning activity can help students generate answers that are appropriate to their cognitive level. |
| Design Knowledge | • Identifies three problem-solving approaches students take when organising plans in a design strategy activity. These approaches are: experimenting with the plans order, ordering plans from the top down, and addressing easier plans first.<br>• Shows that the presence of a design-based Parsons problem as an intervention to a programming assignment can support students' use of SRL strategies to solve programming problems. This study finds that the Parsons problems support the use of design and planning SRL strategies. |

TABLE 1.2. List of Thesis Contributions

of design treatments within the assignment presentation to help students better understand the problem. The thesis identifies design treatments that help students use deeper reasoning and Self-Regulated Learning strategies to better understand the problem and validate their work upon coding completion.

The third contribution relates to critical thinking skills through a questioning intervention. A questioning framework was developed to identify the appropriate question to ask students based on their abilities for the desired task. The framework was formed by classifying instructional question types to the original Bloom's Taxonomy (Bloom, 1956) cognitive levels. The resulting framework can help educators build questioning activities that can also help their students improve skills to analyse programming problems.

Another contribution relating to critical thinking skills is in identifying the question types that encourage CS1 students to generate answers appropriate to their current cognitive levels. The thesis findings suggest a blend of low and high-order thinking questions help students to better understand the problem. When combining low and high-order questions, students analyse the problem deeper, and help reduce misconceptions related the problem (Weusijana, Reisbeck, and Jr, 2004).

The fifth contribution relates to using Parsons problems learning tool as a design-based intervention. The thesis shows students using Parsons problems to engage Self-Regulated Learning (SRL) strategies, active strategies that students take when taking control over their learning (Zimmerman, 1989). The study also identifies different behaviours adopted by the students when using Parsons problems during the design process, which can help identify struggling students. By identifying the struggling students during the design process, corrective feedback could be provided to help adjust their understanding of the problem, to help them successfully complete the assignment.

## 1.4 Thesis Overview

The remainder of this thesis evaluates the Codification Pedagogy. The discussion of the pedagogy is divided into three learning activities for deeper analysis: program comprehension, critical thinking skills, and design knowledge. Chapter 2 presents the motivation and background for developing the pedagogy, along with providing a description of each of the pedagogy's learning activities. Chapter 3 presents the related work for this thesis. Chapter 4 outlines each of the studies performed in this thesis.

Chapters 5, 6, and 7 present the studies performed on the pedagogy's learning activity focusing on improving students' program comprehension through the assignment presentation. Chapter 8 discusses the support of critical thinking skills in the pedagogy through questioning activities. Chapter 9 presents a design strategy activity as an intervention for programming assignments to support design knowledge. Chapter 10 brings together the three learning activities to evaluate the entire pedagogy.

Chapter 11 is the final chapter, presenting the pedagogy's contribution to the field of Computer Science Education and future research opportunities. The thesis concludes with an Appendix containing the practical programming assignments used in the research. Appendix A contains the assignments that include the Codification Pedagogy, and Appendix B presents the instruments used in the quantitative and qualitative studies for the data collection.

# Chapter 2

# Pedagogy Design and Background

This chapter presents the motivation, background, and design for the Codification Pedagogy. The chapter is organised as follows. Section 2.1 presents the overview for the Codification Pedagogy, which introduces the pedagogical goals. Section 2.2 presents the background for the pedagogical goals, and introduces the learning skills and strategies used in the pedagogy. Sections 2.3 and 2.4 present the learning skills and strategies that support the pedagogical goals. Section 2.5 presents the design of the pedagogy. Section 2.6 presents the summary.

## 2.1 Overview

Introductory programming (CS1) students sometimes avoid the software design process because they have limited design knowledge (Robins, Rountree, and Rountree, 2003). With limited design knowledge to perform inquiry skills, students might not fully comprehend the problem (Roll et al., 2007), resulting in the development of misconceptions or difficulty in understanding the problem (Qian and Lehman, 2017). When CS1 students have difficulty understanding the problem, they might adopt Poor Learning Tendencies (PLTs) to solve the problem (Carbone et al., 2001; Ricken, 2005).

Poor Learning Tendencies (PLTs) are habits used by students to compensate for their lack of understanding (Baird and Northfield, 1995). Poor Learning Tendencies were defined by the Project to Enhance Effective Learning (PEEL), originating at Bellarine Secondary College in 1993. The PEEL project identified habits that negatively impact students' approaches to learning, and provided teaching techniques to practitioners that promote Good Learning Behaviours (GLBs). Good Learning Behaviours promote active learning (Baird and Northfield, 1995), which can improve long-term retention and comprehension of the instructional material by students (Briggs, 2005). Examples include:

- *Impulsive and superficial attention*: The student skims over instructional materials, causing them to overlook the core concepts. As a result, the student focuses on other aspects of the problem, leading to incorrect or incompleted programming solutions (Carbone et al., 2000; Carbone et al., 2001).
- *Lack of reflective thinking*: The student does not engage prior experiences to solve the problem, and views the problem in isolation without drawing on past solved problems for guidance (Carbone et al., 2000; Carbone et al., 2001).
- *Inappropriate application*: The student applies a concept without considering whether it is the best approach for the situation (Baird and Northfield, 1995).

This thesis is motivated to minimise students' adoption of Poor Learning Tendencies and help them adopt Good Learning Behaviours. Within the context of CS1 students solving programming problems, Good Learning Behaviours identified by the PEEL project are:

- *Checking comprehension of instructional materials*: Related to the 'Self-evaluation' (Zimmerman, 1989) SRL strategy, this behaviour encourages the student to review the instructional materials to validate their understanding (Baird and Northfield, 1995).
- *Organising a strategy before starting the learning activity*: Related to the 'Goal-setting and planning' (Zimmerman, 1989) SRL strategy, this behaviour demonstrates the student is thinking of ways to solve a problem prior to implementing the solution (Baird and Northfield, 1995).
- *Seeking additional information*: Related to the 'Seeking information' (Zimmerman, 1989) SRL strategy, this behaviour shows the student pursuing additional information if they feel they do not understand the problem (Baird and Northfield, 1995).

When examining Good Learning Behaviours within the context of Computer Science, students take an active role in their learning to better understand and plan programming problems. This thesis seeks to encourage students in taking an active role in their learning by providing them support to improve their program comprehension and promote the use of Self-Regulated Learning (SRL) strategies. Improved program comprehension and use of SRL strategies are the pedagogical goals for this thesis. Section 2.2 introduces the background literature for the pedagogical goals, explaining the program comprehension process and usage of Self-Regulated Learning (SRL) strategies.

To achieve these pedagogical goals, the Codification Pedagogy incorporates learning skills and strategies designed to encourage critical thinking skills and promote the use of design strategies. Critical thinking skills are the learning skills that encourage students to use Good Learning Behaviours by *checking their comprehension on instructional materials*, and *seeking additional information* to better understand the problem. Critical thinking skills are presented in Section 2.3. Design strategies are the learning strategies that encourage students to use the Good Learning Behaviours for *organising a strategy before starting the learning activity*. Design strategies are described in Section 2.4.

This thesis considers the pedagogical design while developing the Codification Pedagogy. Pedagogical design includes the structure of the pedagogy and learning objectives, to guide CS1 students through the software design process. The presentation of the pedagogy gives students the opportunity to replicate the design process after experts' behaviours. The pedagogy is designed to model an expert's behaviour by applying a problem-solving process model (Gick, 1986). The overall design of the pedagogy, discussing the problem-solving process model, is discussed in Section 2.5. The model of the problem-solving process is contained within a scaffolded learning environment, to provide students the opportunity to practise the expert's problem-solving process. Scaffolding is an instructional strategy that provides students a path to successfully complete a task without giving the learner the answer (Wood, Bruner, and Gail, 1976), and is a core teaching method for the the Cognitive Apprenticeship teaching method (Collins, Brown, and Holum, 1991). The Codification Pedagogy adopts Cognitive Apprenticeship for the core teaching method. Cognitive Apprenticeship is presented in Section 2.5.1.

## 2.2 Pedagogical Goals

This section presents the background literature for the pedagogical goals. The pedagogical goals are designed to help students improve their program comprehension, and support their use of Self-Regulated Learning (SRL) strategies. Section 2.2.1 presents the background literature on program comprehension, while Section 2.2.2 presents the background literature on Self-Regulated Learning (SRL) strategies.

### 2.2.1 Program Comprehension

This section provides the background literature for the first pedagogical goal, program comprehension. *Program Comprehension* occurs when a learner uses cognitive strategies to form a mental representation of a solution through the assimilation of the text-based representation and their current knowledgebase (Schulte et al., 2010). Program comprehension is an action performed by the learner with the purpose to better understand the problem (Détienne, 2002). Through the program comprehension process, the learner can form a better mental representation of what they are trying to solve, which makes program comprehension an important part of the problem-solving process (Sonnentag, 1998). Figure 2.1 provides a visual representation of program comprehension defined by Schulte et al. (Schulte et al., 2010). Schulte et al. developed this representation through the synthesis of previous program comprehension models. Figure 2.1 identifies common components within established program comprehension models.

The generalised program comprehension model, shown in Figure 2.1, displays two key components: *World* and *Programmer*. The *World* component is the external representation of the problem. In the *World* component are external representations that contributes to the learners' understanding of the problem, such as the instructional material that describes the programming problem. The second component in program comprehension model is the *Programmer* component, representing elements within the learner used in the program comprehension process. The *Programmer* component contains subcomponents. One subcomponent is the *cognitive structure*, which is the combination of the learner's internal knowledgebase and their mental model of the problem. The other subcomponent is the *assimilation process*. In the *assimilation process*, the learner forms the mental model, or cognitive schema, for solving the problem by applying their internal knowledgebase to the external representations.

The cognitive schema is activated by the learner when they determine how to solve the problem. Some cognitive schemas novices take are trial and error (Chi, Glaser, and Rees, 1982) and means-ends analysis (Sweller and Levine, 1982). Means-ends analysis is a goal-based strategy that is performed in the cognitive space by the learner, where they focus on the goal state based on their current problem state. 'Means-ends analysis does not foster the learning of problem states and associated moved that will result in a new state' (Gick, 1986, p. 109). To shift the learner from using goal-based strategies, such as means-ends analysis, worked examples can be incorporated into the instructional materials (Sweller, 1988) to help them develop their cognitive schema formation. Worked examples provide the learner with a problem that includes the problem description and guidance on how to solve the problem (Morrison, Margulieux, and Guzdial, 2015). The worked examples can help the learner identify design (Kuittinen and Sajaniemi, 2004) and algorithmic (Muller, 2005) patterns, which can help develop their cognitive schema formation.

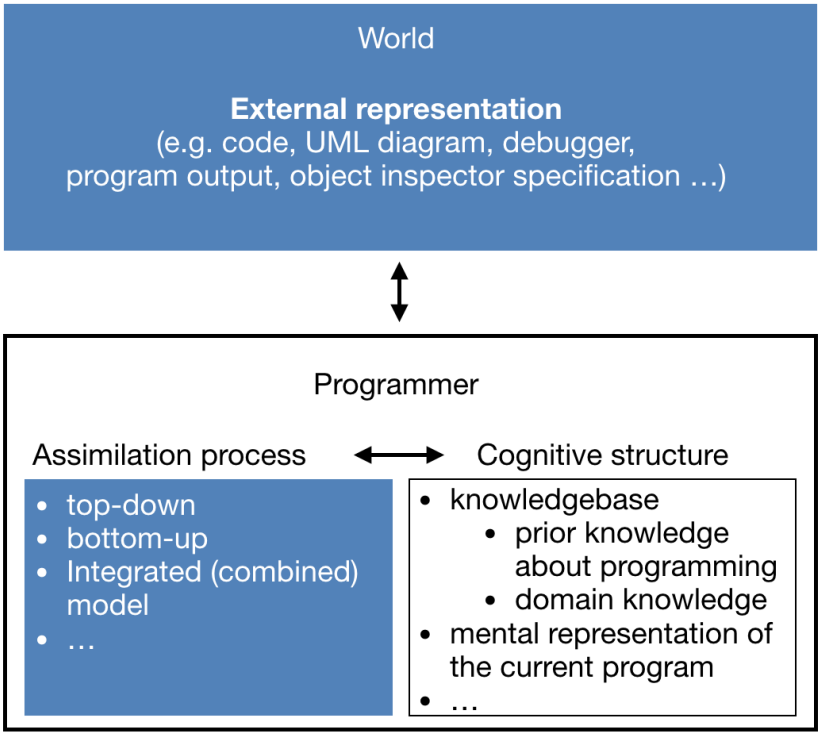



FIGURE 2.1. Program Comprehension Model (Schulte et al., 2010, p. 66)

The program comprehension process is unique to an individual because it involves elements contained within the individual, such as the learner's internal knowledgebase and the method in which they form a cognitive schema. The program comprehension process also evolves as the learner gains more internal knowledge that can be applied in the process. The program comprehension process differs between novices and experts due to how CS1 students draw from their internal knowledge to help them understand the problem (Schulte et al., 2010). With limited internal knowledge, a novice leans more towards discourse comprehension. Discourse comprehension forms a mental model that relies on word identification and sentence parsing in the problem description. Using discourse comprehension can sometimes result in flawed mental models, which demonstrates the learner does not fully understand the problem (Perrig and Kintsch, 1985). A flawed mental model can cause misconceptions about the problem, where the learner's belief about the solution does not produce a viable theory for the solution (Smith III, diSessa, and Roschelle, 1993). Flawed mental models might form due to learner's internal knowledge, where the internal knowledge is a component within the program comprehension process. The learner's fragile internal knowledge used in the program comprehension process might reflect fragile understanding of CS concepts, which could influence their ability to form correct concepts (diSessa, 2014).

Previous research (Rajlich and Wilde, 2002) has identified the different program comprehension processes made by novices and experts. When novices form a mental model, they adopt a bottom-up approach (Pillay, 2003; Schulte et al., 2010). The bottom-up approach 'involves putting the system together to formulate sub-systems: from lowest details to the high abstract level' (Popovic and Kraal, 2010). The bottom-up approach involves the novice mapping their knowledge about the problem to microstructures within the programming solution before having an overall understanding of how these microstructures work together to form a solution. Experts use

a top-down approach (Schulte et al., 2010; Gerdes, Juering, and Heeren, 2012), incorporating higher-level abstractions when grouping code statements. Experts begin with simpler elements across all the requirements, progressing to complicated aspects of the requirements until the solution is completed. Experts have been shown to take time, 'to evaluate the task and the data prior to engaging in the solution process' (Schoenfeld, 1992). How a novice evolves from the bottom-up model to integrating a top-down approach involves the learner building their internal knowledge through experience and the learning process (Détienne, 2002).

As novices acquire more knowledge and skills, they advance through the levels of programming competence. Identifying the learner's level of programming competence can help educators better understand the learner's behaviours and abilities. The Dreyfus Model (Dreyfus and Dreyfus, 1986) identifies the progression of programming competence, which includes the learner's behaviours. The Dreyfus Model is a taxonomy containing five developmental stages: *Novice*, *Advanced Beginner*, *Competent*, *Proficient*, and *Expert*. A *Novice* elevates to an *Advanced Beginner* when they begin to apply prior experiences to the problem-solving process. The behaviours provided by the Dreyfus Model relate to Good Learner Behaviours (Baird and Northfield, 1995) previously introduced in Section 2.1. For example, a *Competent* programmer uses skills that help them develop plans designed to achieve their programming goals, and they become aware of their actions to reach these goals, which relates to the Good Learning Behaviour *organising a strategy before starting the learning activity*. The Dreyfus Model can be helpful in identifying Good Learner Behaviours for learners at a certain level of competence, ensuring they are using these behaviours for long-term retention and comprehension of the instructional materials (Briggs, 2005).

This section presented the process novices engage when understanding a problem, comparing the process to an expert's approach. These comparisons can help guide novices towards using the expert's problem-solving approaches. This thesis focuses on the helping improve students' program comprehension by focusing on the external representation of the problem. Another method previously examined to improve students' program comprehension is another pedagogy, *comprehension-first* pedagogy (Nelson, 2017), that focuses on teaching programming semantics before teaching novices how to code. This pedagogy has been applied to program tracing in CS1 (Nelson, Xie, and Ko, 2017), where the program tracing is performed during the software implementation and debugging phases.

This section also identified novices' behaviours when elevating their skills towards becoming expert programmers. Identifying these behaviours can help with assessing their progress using learning activities, and can help develop instructional materials appropriate for their programming competence.

### 2.2.2 Self-Regulated Learning Strategies

This section provides the background literature for the second pedagogical goal, the use of Self-Regulated Learning (SRL) strategies. SRL strategies help students 'plan, set goals, organise, self-monitor, and self-evaluate', which can improve the learning process (Zimmerman, 1989). The development of SRL strategies by students is a complex issue, associated with the perceived purpose of engagement with the activity, their self-perception of their ability, and the situated context of the activity—these three factors impact upon the self-regulation strategies that the student then considers relevant for application (Paris and Turner, 1994).

The SRL strategies were developed and categorised by Barry Zimmerman and Manuel Martinez-Pons through the use of interviews with students, discussing strategies they use when studying (Zimmerman and Pons, 1986). Zimmerman and Martinez-Pons devised a 14 category taxonomy for SRL strategies, which has since been used as a framework for researchers to investigate SRL strategies. In Computer Science Education, SRL continues to be researched due to proven gains in students' understanding of learning materials when strategies are applied (Bergin, Reilly, and Traynor, 2005), resulting in higher grades and motivation (Kizilcec, Pérez-Sanagustín, and Maldonado, 2017), along with higher student confidence to obtain their goals from constructed plans (Alexiou and Paraskeva, 2010). Students performing well in programming tasks have been shown to frequently use SRL strategies, showing an increase in understanding of the learning materials (Bergin, Reilly, and Traynor, 2005). When students are encouraged to use SRL strategies, there is an increase in their motivation, resulting in higher grades (Kizilcec, Pérez-Sanagustín, and Maldonado, 2017).

From the 14 identified SRL strategies, Zimmerman (2000) went on to develop a model that describes a better approach to SRL. Zimmerman's triadic model (Zimmerman, 2000) is comprised of three stages. The first stage in the model, *forethought*, gives the student the opportunity to analyse the problem's goals, identify tasks, and make a plan to carry out these tasks. The second phase, *performance control*, is when the student will perform the task and manage how they are processing in accomplishing it. During the performance control phase, the student engages other skills and strategies to stay on task and uses these skills and strategies to remain motivated to accomplish all tasks. In the final phase, *self-reflection*, the student will evaluate their work on the tasks, judging whether they have successfully accomplished them. This phase involves *metacognition*, the student's ability to reflect and understand the problem, which has them taking control of their learning (Schraw and Dennison, 1994). With this last, self-reflection phase, as part of the problem-solving process the student will likely spend more time in the design process (Bishop-Clark, 1995).

The triadic model brings in the essential SRL and places achievement goals as one of four processes that students utilise in the initial, *forethought* phase of self-regulation. For example, a student who adopts a mastery-approach goal would view the purpose of activities as focused on developing learning and understanding. In this example, the student would then proceed to employ strategies positively associated with this purpose (Pintrich, 2000). Nicholls (1992) expands upon this to identify that achievement goals identify not only what it means to succeed but also how this success will be achieved. In this view, a student's adoption of SRL strategies is not simply associated with those that are available to them, that is, those that they have developed skill in, but is associated with those that they consider to be relevant to the situation and purpose.

This section presented a type of learning strategy that has been shown (Bergin, Reilly, and Traynor, 2005) to help students better understand the instructional materials. Promoting the use of SRL skills can help students elevate their skill set towards becoming expert programmers (Buckley and Exton, 2003).

## 2.3 Critical Thinking Skills

This section presents the background literature on critical thinking skills, which is one of the learning skills incorporated into the Codification Pedagogy. Critical thinking skills encourage 'thinking explicitly aimed at well-founded judgment, utilising appropriate evaluative standards in an attempt to determine the true worth, merit, or value of something' (Paul and Elder, 2007). Critical thinking skills can promote reflective thinking, an approach that can help students better understand the problem and improve their program comprehension.

Critical thinking skills are comprised of two types of thinking skills: lower and higher-order thinking skills. Lower-order thinking skills help students to recall, memorise, and understanding learning materials, to promote reflection on previously learned language constructs (Tikhonova and Kudinova, 2015). Lower-order thinking skills engages the learners internal knowledge for recalling the information, and can help demonstrate the learner's pre-existing knowledge. Higher-order thinking skills promote the application of programming knowledge to the assignment's problem domain, encouraging internal reflection through deep reasoning (Bloom, 1956). Higher-order thinking skills require the learner to make certain judgements prior to answering, which requires the learner to bring together other skills to form the judgement, such as reasoning, comprehending, and analysing skills (Chi et al., 1994).

In the CS problem space, when students use both lower and higher-order thinking skills, they are more likely to develop a cross-referencing comprehension strategy (Pennington, 1987). Cross-referencing comprehension involves two layers for understanding: the program and problem domain. The program layer is where the student gains understanding by evaluating the program code. The problem domain layer involves the understanding the problem at the abstract level, typically represented in CS as the problem description. Understanding the problem at both the program and problem domain layer results in a higher level of understanding about the problem (Pennington, 1987), which can contribute to the successful completion of the programmed solution.

The cognitive processes required for lower and higher-order critical thinking skills have been defined in educational objective taxonomies, such as Structure of the Observed Learning Outcome (SOLO) (Biggs and Collis, 1982) and Bloom's Taxonomy (Bloom, 1956). SOLO is an educational objective taxonomy that examines students' solutions to measure their mastery of the domain. The SOLO taxonomy is comprised of five levels of comprehension, and can be used to help educators with instructional materials, such as setting learning objectives and outcomes. Prior work using SOLO in CS has classified reading problems (Lister et al., 2006) and exam questions (Petersen, Craig, and Zingaro, 2011; Whalley et al., 2006).

Bloom's Taxonomy, sometimes referred to as simply *Bloom's*, is an educational objective taxonomy that addresses the student's depth of learning, and categorises cognitive structures (Bloom, 1956). Table 2.1 shows the six cognitive levels that comprise Bloom's Taxonomy. The table provides definitions for each of the levels and definitions, ordered from lower to higher-order cognitive levels. The cognitive levels start with the lower-order *Recall* level that involves recollection and memorisation skills by the student. The highest order is the *Evaluation* level that involves deep reasoning to form judgements on the problem space. Bloom's has been used to develop CS classroom exercises (Sanders and Mueller, 2000), analyse assessments (Howard, Carver, and Lane, 1996), develop assessment requirements (Thompson et al., 2008), and identify various cognitive levels for testing (Scott, 2003). Frameworks

using Bloom's have been developed for programming knowledge (Buckley and Exton, 2003) and higher-order problem-solving questions (Zohar and David, 2008).

By engaging both lower and higher-order critical thinking skills, the learner has different outlets to reflect on the problem. These outlets give the learner different ways to access and apply their internal knowledge to the existing problem, providing them with more opportunities to think about and better understand the programming problem.

| **Level** | **Definition** |
|---|---|
| Recall | "Involves the recall of specifics and universals, the recall of methods and processes, or the recall of a pattern, structure, or setting." |
| Comprehension | "Refers to a type of understanding or apprehension such that the individual knows what is being communicated and can make use of the material or idea being communicated without necessarily relating it to other material or seeing its fullest implications." |
| Application | "Use of abstractions in particular and concrete situations." |
| Analysis | "Breakdown of a communication into its constituent elements or parts such that the relative hierarchy of ideas is made clear and/or the relations between ideas expressed are made explicit." |
| Synthesis | "Putting together of elements and parts so as to form a whole." |
| Evaluation | "Judgments about the value of material and methods for given purposes." |

TABLE 2.1. Bloom's Taxonomy Cognitive Levels (Bloom, 1956)

## 2.4 Design Strategies

This section presents related work on design strategies that are supported within the Codification Pedagogy. Design strategies are actions the student takes to find answers when designing a programmed solution (Hoadley and Cox, 2009). An example design strategy is the action taken during the problem-solving process to plan and set goals for the programming problem (Greeno and Simon, 1988). Novices have been shown to use general problem-solving strategies over specific strategies, such as goal setting and planning (Winslow, 1996). Providing students with CS-specific design strategies can support building their knowledge to coordinate the goals and plans of a program (Spoher and Soloway, 1989).

Applying design strategies during the problem-solving process is a trait of an effective novice, enabling them to progress further in their learning with minimal assistance (Robins, Rountree, and Rountree, 2003). The process of using design strategies can help students build their design knowledge establishing metaknowledge containing methods for finding solutions to problems (Hoadley and Cox, 2009). As students gain more experience developing software and build their design knowledge, they are able to identify more ambiguous aspects of the problem and its requirements (Fincher et al., 2004). More design experience and knowledge can help improve the quality in the student's software designs (Atman et al., 1999), and help

them 'perform abstract thinking and to exhibit abstraction skills' (Kramer, 2007). Improvements in CS1 students' design process have been observed between their first and final year at university (Falkner et al., 2015). Novices were shown in this study to design after or during the coding process, while final-year students refactor their designs using Self-Regulated Learning (SRL) design and demonstrating the use of planning strategies.

When novices have limited design knowledge, they will spend little time designing their programming solutions (Pintrich, Berger, and Stemmer, 1987), or completely neglect the design process (Rist, 1995). Instead of integrating a design phase into the problem-solving process, novices will approach solving the problem one line at a time, instead of designing a solution using structures (Winslow, 1996). With fragile design knowledge, novices do not take the time to plan, and have difficulties identifying plans and goals (Robins, Rountree, and Rountree, 2003). Novices sometimes attempt to solve the problem with their existing internal knowledge and generalised problem-solving strategies, rather than using the existing information in the instructional material to help them form new approaches and knowledge (Roll et al., 2007). The instructional materials can contain additional guidelines for the student to elevate their skills (Vihavainen, Paksula, and Luukkainen, 2011a), but they might not see value if the guidelines do not relate to their existing problem-solving strategies.

One design strategy used in CS is the decomposition of programming goals, a process of decomposing problems to identify programming goals and schemas (Guzdial et al., 1998). Prior research (Jeffries, Polson, and Atwood, 1981) has shown that novices and experts approach this design strategy differently. Novices use a depth-first decomposition approach, focusing on details for one task before addressing others (Ormerod and Ridgway, 1999). Depth-first decomposition approach has the novice fully developing the subgoal before evaluating or partially developing other subgoals (Détienne, 2002). This approach can work for novel problems (Newell and Simon, 1972), but other approaches are required for more complex problems, especially when the solution for the completed subgoal does not work for the other problem's subgoals (Ormerod, 2004). Experts integrate a breadth-first decomposition approach that systematically explores the problem before focusing on task details (Robillard, 1999). Experts will develop solutions for all the subgoals at a higher level prior to solving them at a more detailed level. This approach gives the expert more opportunities to explore the design (Ormerod, 2004).

The decomposition of programming goals can be performed at the task (Winslow, 1996) and plan (Soloway, 1986) levels. Task-level design strategies focus on algorithm and data structures, while at the plan level, the design strategies identify the software components and interrelationships for the programmed solution (Rombach, 1990). During the design process, tasks are deconstructed, a process where the learner analyses the problem then designs a solution structure (Fincher et al., 2004). When novices deconstruct tasks, they construct more textual decomposition tasks, while experts draw on more graphical notations (Fincher et al., 2004). Plan-level design involves 'breaking the problem into subproblems and finding a suitable order for completion of the subproblems' (Greeno, 1980). With plan-level design, 'a plan corresponds to a fragment of code that performs actions to achieve a goal' (Hu, Winikoff, and Cranefield, 2012). Planning helps the student identify the subproblems in the textural description of the problem, and outline the purpose of the problem (Bergin, Reilly, and Traynor, 2005). The planning process generates core

components that have been abstracted and represented explicitly (Bateson, Alexander, and Murphy, 1987). During the planning process, novices have difficulty scoping the problem, to identify plans for solving it (Sutcliffe and Maiden, 1992). Experts draw on their internal design knowledge, using known design strategies to help them identify plans for solving the problem (Soloway et al., 1982).

This section presented design strategies and are considerations for the Codification Pedagogy to help students with the skills to decompose a problem into subgoals. Another theory designed to help students develop skills to decompose subgoals is *automation*, where high-level task procedures influence the students' programming behaviours (Merriënboer and Paas, 1990). Automation promotes the use of worked examples that allow students to practice and better understand low-level, task-specific procedures. As the novice gains more experience and understanding, they can build on the previously learned low-level, task-specific procedures, which allows them to construct more complex task-specific procedures like experts during their problem-solving process (Anderson, 2013).

This section described the benefits to CS1 students using design strategies, and the methods they use when starting to integrate design strategies into their problem-solving process. This section also described the experts' approach to design strategies, which novices can progress towards developing their programming solutions. The background literature presented in this section can help better understand the novices' progression in design strategy usage, and can help to assess their design skills. This literature introduces different best practices used by experts during the design process so that instructional materials can be developed to encourage novices to adopt the same best practices.

## 2.5 Pedagogical Design

This section provides a high-level view of the pedagogy's design, including introducing the pedagogy's teaching methods. Cognitive Apprenticeship is the teaching method applied to the Codification Pedagogy, to present the learning skills and strategies. Section 2.5.1 provides the background literature for the Cognitive Apprenticeship, presenting the six teaching methods within the Cognitive Apprenticeship theory. Section 2.5.2 describes the pedagogy's instructional approach, layered on the Cognitive Apprenticeship teaching methods.

### 2.5.1 Cognitive Apprenticeship

The design of the Codification Pedagogy is grounded in Cognitive Apprenticeship, a teaching method that helps students practice difficult tasks through task demonstration modeled after experts' behaviours (Caspersen and Bennedsen, 2007). Cognitive Apprenticeship provides instructional and organisational guidance in the instructional design and learning activities. The following is a list of teaching methods within Cognitive Apprenticeship (Collins, Brown, and Holum, 1991):

- *Modeling*: A teaching method that allows the student to observe how an expert performs a given task. Through demonstration, the teaching method is making the process visible, encouraging the learner to replicate the behaviour to complete their tasks.
- *Coaching*: A teaching method that makes the expert responsible for guiding the learner through the problem-solving process. The expert supervises the student's work through the entire learning experience.

- *Scaffolding*: An instructional strategy that provides the learner a path to successfully completing a task without giving them the answer (Wood, Bruner, and Gail, 1976). Scaffolding has been show to control the student's frustration and focus the learner on the task (Chalk, 2001).
- *Articulation*: A teaching method that promotes the novice to self-reflect and focus on the problem, modeling an expert's behaviour to engage their metacognitive skills and problem-solving strategies for solving the problem.
- *Reflection*: A teaching method that encourages the student to compare their work with others. Like the *Articulation* teaching method, *Reflection* also has the student engaging their metacognitive skills, modeling problem-solving strategies after an expert's approach when encountering a problem.
- *Exploration*: A teaching method that encourages the learner to generate their own problems and determine ways to solve them. *Exploration* attempts to help the learner transfer their skills outside the scaffolding learning environment.

Cognitive Apprenticeship uses guided learning environments to support students' acquisition of knowledge and skills. Support within the learning environment is reduced so that students can apply their acquired knowledge and skills as independent learners (Ghefaili, 2003). A pedagogical approach using Cognitive Apprenticeship can raise students' awareness of the processes used to solve a problem, and teach them skills that enable them to apply those processes in other problem-solving situations (Collins, Brown, and Holum, 1991).

Understanding the different teaching methods supported by the cognitive apprenticeship can help identify methods that better support CS1 students when they have little experience with the design process. For example, *Exploration* would require a learner to have more programming experience to create and solve problems. This teaching method could be reserved for learners having more design experience, encouraging them to demonstrate best practices previously learned.

### 2.5.2 Proposed Pedagogical Design

This section describes the design of the Codification Pedagogy. As stated in Section 2.1, the development of this pedagogy is motivated to encourage CS1 students to use Good Learning Behaviours when solving procedural programming assignments. In the context of CS, these behaviours include having the student reflect on the problem for better understanding, and organise a plan to solve the problem. To encourage Good Learning Behaviours during the design process, the Codification Pedagogy uses a problem-solving process model (Gick, 1986) that guides the student through the problem-solving process. The Codification Pedagogy layers the learning activities on the problem-solving states within the model. The learning activities are designed to include skills and strategies that encourage students to use Good Learning Behaviours when solving CS1 programming problems.

Figure 2.2 is the problem-solving process model used within the Codification Pedagogy that presents the best practices in the problem-solving strategy. The figure shows four states in the model: *Construct a Representation*, *Search for a Solution*, *Implement Solution*, and *Stop*. Within the model are transitions that represent the relationship between the different problem-solving states. Figure 2.2 shows two highlighted states, to signify the states supporting the pedagogical goals. The *Construct a Representation* state involves the program comprehension process, where the student

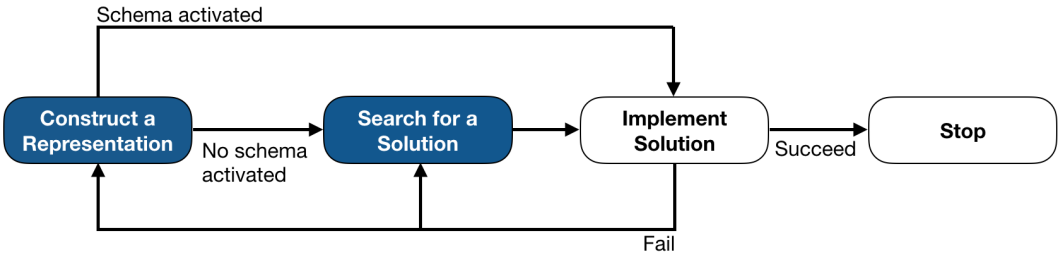



FIGURE 2.2. Problem-Solving Process Model (Gick, 1986, p. 101)

forms a mental model of the solution. This process is initiated through the reading of the problem description. The pedagogy is designed to help students through this state by presenting them with learning activities that support the construction of a better mental model of the solution. The *Search for a Solution* state involves the student organising a plan for solving the problem. The pedagogy is designed to provide learning activities that support students' use of SRL skills that encourage them to organise a plan prior to software implementation. The pedagogy is designed to support students in this state with interventions that encourage critical thinking skills and use of SRL strategies.

Figure 2.2 shows the problem-solving process model with state transitions that enable the student to return to prior problem-solving states. The pedagogy adopts these transitions, allowing students to make the same transition to adjust their mental model of the problem, and form a more accurate plan for solving the problem.

From the Cognitive Apprenticeship teaching methods presented in Section 2.5.1, the pedagogy incorporates *Modeling* and *Scaffolding* core teaching methods for promoting effective learning skills and strategies. Figure 2.3 shows how the pedagogical goals with its learning skills and strategies are applied to the problem-solving process model. The figure shows the pedagogy's components supported within the problem-solving states. Program comprehension, critical thinking skills, and support of Self-Regulated Learning strategies help students form a representation of the solution by helping students form a more accurate mental model of the problem.

The *Scaffolding* teaching method is applied in the Codification Pedagogy by providing students a highly scaffolded learning environment in early assignments. In assignments containing the pedagogy early in the semester, more instructional support and guidance are provided. As students practice programming and gain more design experience, the guidance and support is reduced, to allow them to practice their design skills and strategies as independent learners.

The application of the pedagogical goals is performed through three key components in the Codification Pedagogy. The first learning activity is the *Assignment Presentation* activity, presenting the practical programming assignment in a scaffolded learning environment, designed to help the student better understand the problem.

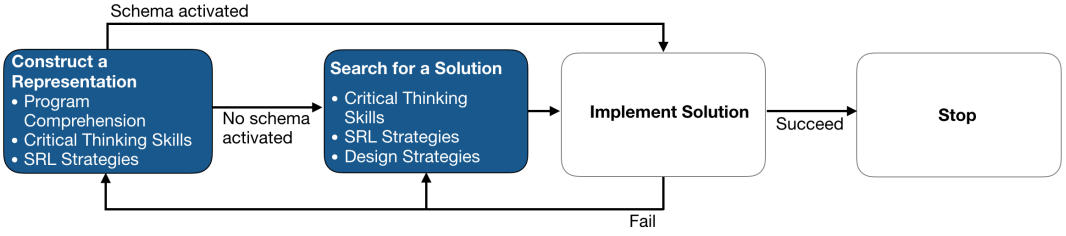



FIGURE 2.3. Codification Pedagogy Workflow

The second and third components, *Questioning* and *Design Strategy* activities, are designed as interventions, to help students through the design process. The interventions are designed to encourage students to further think about the problem, to identify the problem's goals, and to organise a plan that will help them solve the problem. More information on these learning activities are presented in Chapter 4.

## 2.6 Summary

This chapter presented the motivation and background for the Codification Pedagogy. The thesis is motivated to encourage CS1 students to use Good Learning Behaviours (GLBs) during the design process through the application of a design-based pedagogy. Theories of cognitive and metacognitive strategies emerged from the motivation for this thesis that form the pedagogical goals. The pedagogical goals are improving students' program comprehension, and encouraging students to use Self-Related Learning (SRL) strategies. To support the pedagogical goals, learning skills and strategies were identified. SRL, Critical thinking skills, and design strategies were presented to show how these skills and strategies achieve these pedagogical goals.

This chapter presented the design of the Codification Pedagogy, grounded in Cognitive Apprenticeship theory. The background literature on Cognitive Apprenticeship presented the teaching methods that can be used to support the pedagogy's instructional strategy. This chapter described how the pedagogy layers the problem-solving process model on the Cognitive Apprenticeship's *Modeling* and *Scaffolding* core teaching methods, to demonstrate to novices how an expert would approach the design process. The description of the proposed pedagogical design also described how the learner is supported during their design process.

This chapter presented the background literature for theories that informed the design and development of the Codification Pedagogy. The next chapter continues investigating the design of the Codification Pedagogy, by surveying CS literature related to the theories presented in this chapter. Surveying the literature can help identify how best to apply the theories in the pedagogy and identify how this thesis can build on work previously undertaken in the field of Computer Science Education.

# Chapter 3

# Related Work

This chapter surveys previous work using theories explored in this thesis. The chapter is organised as follows. Section 3.1 provides an overview of this chapter. Sections 3.3, 3.4, and 3.5 present the surveyed work using the theories that form the Codification Pedagogy. Finally, Section 3.6 presents the summary.

## 3.1 Overview

In Chapter 2, background literature was presented that described the theories used in the design and development of the Codification Pedagogy. This chapter is exploring the Computer Science Education landscape more specifically to see what work has been done previously to improve students' program comprehension, and support their use of Self-Regulated Learning (SRL) strategies. This chapter explores related work in CS on program comprehension, SRL strategies, critical thinking skills, and design strategies. The related work has influenced the design and development of the Codification Pedagogy, providing guidance on presenting the learning activities and the applied teaching method.

This chapter organises the surveyed related work in a similar way as the presentation of the background literature in Chapter 2. Section 3.2 presents the related work using the theories that support the pedagogical goals. Sections 3.3 and 3.4 present prior research using the learning skills and strategies. Finally, Section 3.5 presents related work using the Cognitive Apprenticeship teaching method.

## 3.2 Pedagogical Goals

Section 3.2.1 presents the related work in CS using theories in program comprehension. Section 3.2.2 presents the related work incorporating Self-Regulated Learning (SRL) strategies into CS learning.

### 3.2.1 Program Comprehension

This section presents related work using program comprehension. The background literature for program comprehension was discussed in Section 2.2.1. The background literature presented a generic program comprehension model that identified key components that explain the program comprehension process. This section presents different program comprehension models, with some previously applied in the classroom. These models focus on different areas of program comprehension, helping students through the problem-solving process.

Different program comprehension models have been developed to describe theories on the programmer's cognitive processes as they attempt to better understand the program. The first program comprehension model presented in this section is

the Pennington Model (Pennington, 1987). The Pennington Model brings to the forefront textual descriptions as influencers to the students' mental model. The Pennington Model builds on the text comprehension theory (van Dijk and Kintsch, 1983), focusing on text-based understanding through the assimilation of the textual information with the individual's internal knowledgebase. Textual comprehension is prominent in the Pennington Model, where one of the two model layers focuses on the textual representation. The first model layer, the program model, relates to the textual representation and how the student recalls the problem. The second layer is the domain model, where the mental model is formed by the student when interpreting the textual representation. The Pennington Model was developed by observing professional software developers refactor code, and was developed by observing the behaviours of software developers within the procedural language paradigm. The study observed the participants evaluating the code for the first time, using strategies to help them better understand. Think-alouds were used in this study to analyse the developers' refactoring processes. The study concluded that participants with higher comprehension during the refactoring process could relate to the problem both at the abstract problem domain and at the coding level.

Bonar and Soloway (1985) developed a program comprehension model by observing novices during the problem-solving process. The purpose for developing the model was to better understand the reasons for novices introducing bugs in their programming solutions. The model is the first approach to explaining to novices why bugs are present in their software. The model was developed through observing novices solve programming problems and interviews conducted with them. The results from the developing this model showed novices' pre-programming knowledge — the learner's internal knowledge prior to programming (Spohrer, 1992) — is vital in their ability to debug and create patches for programming problems.

Caspersen and Bennedsen (2007) developed a program comprehension model based on cognitive architecture and three learning theories: cognitive load theory, cognitive apprenticeship and worked examples. Cognitive Load Theory is an instructional theory 'that was explicitly developed as a theory of instructional design based on our knowledge of human cognitive architecture. Cognitive load theory consists of aspects of human cognitive architecture that are relevant to instruction along with the instructional consequences that flow from the architecture' (Sweller, Ayres, and Kalyuga, 2011, p. v). Cognitive apprenticeship is a teaching approach that helps students practice difficult tasks through task demonstration modeled after experts' behaviours (Caspersen and Bennedsen, 2007). Worked examples are used to promote cognitive skill acquisition in the program comprehension model. The program comprehension model was developed by researching the theories in cognitive sciences and educational psychology to form the model and learning activities, designed to apply in the classroom. This program comprehension model and its learning activities were later applied to a learning environment, which resulted in a reduction in failing students (Knobelsdorf, Kreitz, and Böhne, 2014).

This section presented different program comprehension models developed to represent how experts and novices understand a problem within the problem-solving process. Researchers developed these models to help novices improve their program comprehension, either through awareness of their current actions, such as introduction of bugs in programming problems, or leveraging established instructional design and educational theories for instructional design. The studies presented in this section demonstrate to novices the best practices used by experts to better understand a problem, which resulted in higher learning gains by the novices.

### 3.2.2 Self-Regulated Learning Strategies

This section presents the related work about Self-Regulated Learning (SRL) strategies in the CS problem space. The background literature SRL strategies was previously presented in Section 2.2.2. In the background literature, the original Zimmerman and Martinez-Pons SRL taxonomy (Zimmerman, 1989) was presented. The taxonomy contains 14 SRL categories with an additional category initiated by outside influencers (Zimmerman, 1989). Table 3.1 presents the 14 categories and definitions defined by Zimmerman (1989) in a classroom environment. The table contains a third column that provides adaptions for CS students when using these strategies within e-learning tools (Garcia, Falkner, and Vivian, 2018). The adaptation for CS usage evaluated the 14 SRL categories and devised a framework for how the strategy could be applied in online setting with e-learning tools. The table demonstrates the application of the SRL strategies in an online learning environment.

Since the original study (Zimmerman, 1989), the Computer Science Education community has continued to research SRL within the CS problem space. Prior work (Bergin, Reilly, and Traynor, 2005) focused on identifying a relationship between SRL and programming performance in introductory programming courses. The study was conducted in an introductory object-oriented programming course with 35 participants taking a questionnaire on motivational and learning strategies, and evaluating the participants' assessments throughout the course, such as examinations. The results from the study showed students performing well with programming tasks when they frequently used SRL strategies. Strategies related to cognitive, metacognitive, and resource management played a significant (45%) role in participants' programming performance.

Violet (1991) explored the co-operative development of metacognitive strategies for a CS1 course that involved students developing metacognitive strategies through the use of model behaviours and coaching procedures designed to guide them through the strategy. The research involved an experiment and control groups studied for 13 weeks. The study examined the groups' development of metacognitive strategies related to programming. Students receiving instructional method to guide the metacognitive strategies showed high learning outcomes.

Another study (Alexiou and Paraskeva, 2010) used the development of an e-portfolio to encourage students' use of SRL strategies. This study was conducted with students enrolled in a CS professional development course. The 41 participants took part in a pre-test to measure their pre-existing knowledge on learning strategies and creating e-portfolio. The study applied the Zimmerman's triadic model (Zimmerman, 2000) of self-regulation, where students were guided through the three SRL phases during the development of the e-portfolio. For example, the e-portfolio system administrator guided students towards the execution phase of the model by messaging with participants. Results from this study and developing the e-portfolio showed students having higher confidence in their ability to apply SRL strategies to construct plans to obtain their goals, and increased their motivation throughout the construction of the e-portfolio.

The last study examined for this section (Kizilcec, Pérez-Sanagustín, and Maldonado, 2017) investigated the encouragement of SRL usage in Massive Open Online Courses (MOOCs), a learning environment with a low level of support and guidance. The study involved training participants to use SRL strategies in MOOCs, with the MOOC environment prompting the participants to use the SRL strategies. When students were encouraged through prompts to use SRL strategies within the

MOOCs, the results from the study showed an increase in participants' motivation and grades when they engaged SRL strategies.

| **Categories** | **Definitions** | **CS Students' Usage** |
|---|---|---|
| 1. Self-evaluation | Student-initiated evaluations of the quality or progress of their work. | Student-initiated self-assessments to validate programming exercises. |
| 2. Organising & transforming | Student-initiated overt or covert rearrangement of instructional materials to improve learning. | Student-initiated development of design plans prior to programming coding assignments. |
| 3. Goal-setting & planning | Statements indicating student setting of educational goals or subgoals and planning for sequencing, time, and completing activities related to those goals. | Student setting programming goals and time aside when developing programming assignments or studying for a tests. |
| 4. Seeking information | Student-initiated efforts to secure further task information from nonsocial sources when undertaking an assignment. | Student-initiated use of online knowledgebases to assist in further understanding of programming objectives. |
| 5. Keeping records & monitoring | Student-initiated efforts to record events or results. | Student-initiated effort to save or link to learning materials. |
| 6. Environmental structuring | Student-initiated efforts to select or arrange the physical setting to make learning easier. | Student-initiated arrangement of windows for digital environment more conducive to learning. |
| 7. Self-consequences | Statements indicating students arrangement or imagination of rewards or punishment for success or failure. | Students arrangement or imagination of rewards or punishment for success, failure, or achievement of awards during the learning process. |
| 8. Rehearsing & memorising | Student-initiated efforts to memorise material by overt or covert practice. | Student-initiated efforts to memorise programming objectives by overt practice. |
| 9-11. Seeking social assistance | Statements indicating student-initiated efforts to solicit help from *peers* (9), *teachers* (10), and *adults* (11). | Student-initiated efforts to solicit or discover answers to questions from *peers* (9), *teachers* (10), and *adults* (11) through social knowledgebases. |
| 12-14. Reviewing records | Student-initiated efforts to reread *tests* (12), *notes* (13), or *textbooks* (14) to prepare for class or further testing. | Student-initiated efforts to reread *notes, programming logs* (12), *tests* (13), or *learning materials* (14) to prepare for class or further testing. |

| | | |
|---|---|---|
| 15. Other | Statements indicating behavior that is initiated by other persons such as teachers or parents and is unclear verbal responses. | Learning behavior prompted by outside influencers, such as teacher, parents, or digital agents. |

TABLE 3.1. SRL Strategies Categorised within a CS e-Learning Context (Garcia, Falkner, and Vivian, 2018)

This section presented related work in the field of Computer Science Education, encouraging the use of SRL strategies. The related work encouraged the learning of SRL strategies in learning environments, modeling strategy use through active learning. The related work demonstrated higher motivation and grades from students applying SRL strategies during their learning. The related work also showed that additional guidance through questioning increase students' use of SRL strategies, even within online learning environments. The use of SRL strategies within online learning environments can help students take an proactive approach to their learning within a learning environment with a low level of guidance.

## 3.3 Critical Thinking Skills

This section describes the related work using critical thinking skills. Section 2.3 presented the background literature for critical thinking skills, promoting 'thinking explicitly aimed at well-formed judgment, utilising appropriate evaluative standards in an attempt to determine the true worth, merit, or value of something' (Paul and Elder, 2007, pp.71-72). Questioning is a teaching method that encourages critical thinking skills that engages students' cognitive processes to reflect on a problem prior to answering (Bloom, 1956), to help students reduce their misconceptions (Collins, 1985; Weusijana, Reisbeck, and Jr, 2004). Questioning helps students overcome certain aspects of their fragile knowledge by helping them fill in holes in their partial knowledge and getting them to use skills to seek out information (Perkins and Martin, 1985).

Critical thinking skills are classified into lower and higher-order thinking skills. Short-answer questions engage lower-order thinking skills and are used by educators in the classroom to assess students' domain knowledge (Dillon, 1984). Higher-order thinking skills promote internal reflection through deep-reasoning (Bloom, 1956) and knowledge-inference questions (Chi et al., 1994), yet few questions (4%) generated by educators engage higher-order thinking skills (Kerry, 1987). Higher-order thinking-skills questions require the educator to triage a student's knowledge deficit (Collins, 1985), which can be difficult to apply in a large classroom.

The literature presented in this section promotes questioning as an activity to encourage critical thinking skills, but there are other methods to teach critical thinking skills, such as project-focused teaching (Kaasbøll, 1998). Project-focused teaching gives students the opportunity to re-evaluate their work when developing and organising their projects over the course duration. However, project-focused teaching requires changing the teaching approach in the classroom, and not the focus on this research.

Though questioning has been examined in other disciplines (Conati and Vanlehn, 2000; Hausmann and Chi, 2002; Weusijana, Reisbeck, and Jr, 2004; Yang, Newby, and Bill, 2005) to support the teaching of critical thinking skills. The literature in this

section focused on evaluating questioning research in the CS space, researching different question types, such as self-explanation, Socratic, and instructional question types. These different questioning types and the research related to the CS space are discussed further in the following three sections.

### 3.3.1 Self-Explanation Questions

Self-explanation is a metacognitive skill that encourages students to further reflect on the material by having them generate an explanation of what they are trying to comprehend (Conati and Vanlehn, 2000; Vihavainen, Miller, and Settle, 2015). A learning tool that supports self-explanation is AtoL (Yoo et al., 2006), a CS tutoring system administered in a lab environment that integrates self-explanation questioning. AtoL provides CS1 and CS2 students with a list of questions to help them develop programming skills. Teachers can distribute instructions in the lab using AtoL, allowing them to tailor instructional materials for individuals' needs. Results showed students using AtoL had greater knowledge in programming concepts than non-participants.

ProPL is a tutoring system for introductory programming courses, encouraging students to write responses to questions related to the programming assignment (Lane and VanLehn, 2005). Results showed improvements in students' ability to deconstruct problems, especially during the debugging phase. Though ProPL generated positive results with participants, this tutoring system required extensive time from the teacher to prepare instructional materials.

Another study (Vihavainen, Miller, and Settle, 2015) applied self-explanation in a CS1 classroom. In this study, students were asked to explain code segments to their peers, dividing the students into two groups: one receiving questions prompting self-explanation, and the other group performing self-explanation without prompts. The results from the study showed students receiving the self-explanation questions were able to transfer this skill, such as better explanation of code provided in exams. The questions helped the students focus their explanations by having them reflect on the relevant segments of code.

### 3.3.2 Socratic Questioning

The Socratic questioning method is a form of questioning that encourages critical thinking through rational arguments (El-Zakhem, 2016). Socratic questioning assists students in making important observations to improve their programming knowledge (Lane and VanLehn, 2005). A chatbot (Le and Huse, 2016) applied the Socratic Method (Nelson, 1970) in an online learning environment, having students traverse the three Socratic phases. The three Socratic phases are: searching for examples, searching for attributes, and generalising the attributes. Using the chatbot was shown to effectively guide students through the Socratic Method in a group discussion. Socratic questioning has been applied to CS tutor-student dialogues during the debugging process, helping students correct their misconceptions and teach them skills to become more self-reliant during the this process (Wilson, 1987). The Why System (Stevens and Collins, 1977) is a tutoring system that integrated a student-teacher Socratic questioning approach into a script-based tool. The Why System guided students through tutorial sessions, asking probing questions related to a problem. The tool was useful in helping identify missing requirements for solving the problem within the students' dialogues. However, the tool was limited by technology to parse and interpret the natural language responses. When the tool

was developed in 1977, the technology did not have the ability to support natural language comprehension. Through this work, the authors were able to characterise the goal structure of Socratic dialogues for future Socratic questioning tools, providing guidelines to better implement the goal structure of the dialogues.

### 3.3.3 Instructional Question Types

Instructional Question Types (IQTs) are classifications of questions that can be used to ascertain students' understanding of learning materials. IQTs can help raise the awareness of misunderstandings (Graesser and Person, 1994). IQTs have been used in the CS problem space, with IQTs being classified using CS questions (Boyer et al., 2010). The IQT classification involved analysing the dialogues between 78 upper-division CS students and 17 tutors in a classroom environment. The questions from these dialogues were collated to assist CS teachers in asking effective questions in the classroom, and to encourage students to use both lower and higher-order thinking skills. Table 3.2 displays the 23 IQTs with example questions generated from the study.

| **Category** | **Types** |
|---|---|
| Backchannel* | Right? |
| Focus* | See where the array is declared? |
| Hints* | We didn't declare it; should we do it now? |
| Definition | What does that mean? |
| Knowledge* | Have you ever learned about arrays? |
| Calculation | What is 13 % 10? |
| Casual Consequence | What if the digit is 10? |
| Clarification* | What do you mean? |
| Confirmation* | Does that make sense? |
| Enablement | How are the digits represented as bar code? |
| Procedural | How do we get the $i^{th}$ element? |
| Quantification | How many times will this loop repeat? |
| Free Creation | What shall we call it? |
| Goal Orientation | Did you intend to declare a variable there? |
| Casual Antecedent | Why are we getting that error? |
| Feature/Concept Completion | What do we want to put in digits[0]? |
| Free Option | Should the array be in this method or should it be declared up with the other private variables? |
| Judgment | Would you prefer to use math or strings? |
| Justification | Why are we getting the error? |
| Plan | What should we do next? |
| Improvement | Can you see what we could do to fix that? |
| Assessment* | Do you think we're done? |
| Status* | Do you have any questions? |

TABLE 3.2. Instructional Question Types for CS (* denotes emerging question category (Boyer et al., 2010))

## 3.4 Design Strategies

This section presents related work focusing on the students' learning and usage of design strategies during the problem-solving process. Background literature on design strategies was presented in Section 2.4, while this section presents related work on the application of design strategies in prior research.

There are a number of approaches that have been taken to support CS students in the development of design strategies through the use of Intelligent Tutoring System (ITS). *Coached Program Planning* (CPP) is an Intelligent Tutoring System that has students identify the problem's goal and generate steps to achieve the goal (Lane and VanLehn, 2003). The tutoring system has students designing a solution to a programming problem by using a notation to describe the solution. The notation is called pseudocode that is a cross between program code and natural language, which is designed to encourage students to reflect on the problem. The tutoring system uses a dialogue approach to encourage the design process, asking students to identify goals and describe how they would achieve those goals. The results from the study showed students using CPP making fewer mistakes in the structure of their programs, and their software development process was less erratic.

Another tutoring system, Program Planner (ProPL), uses questioning during the design process to encourage CS1 students to deconstruct problems into subgoals (Lane and VanLehn, 2005). Like *Coached Program Planning* , ProPL uses dialogues to guide the students through the design process, and encourages them to use pseudocode to design a solution. ProPL is a Java-based tutoring system available via a web browser. This tool allows the students to view their design nodes and pseudocode while developing their programming solution. The results from using ProPL showed improvements with participants solving composition problems. The study suggests the improvements were due to the participants reflecting on the problem at a higher, abstract level.

Both *Coached Program Planning* and ProPL promoted pseudocode within the tutoring systems. Another study (Garner, 2007) applied pseudocode in a classroom environment. The study was motivated to use pseudocode to encourage students to think more on the design process. When pseudocode was provided in a classroom learning environment, CS1 students felt they were learning another programming language, giving them the impression they were learning two languages. The perception of learning two programming languages influenced students' feedback, where they felt pseudocode was unhelpful.

Another tutoring system (Hu, Winikoff, and Cranefield, 2012) integrated the goals/plan approach (Soloway, 1986), to guide students through the development process. The tool extended the work on GPCeditor (Guzdial et al., 1998), for Build Your Own Blocks (BYOB) within a Visual Programming Environment (VPE). Results from using the tool showed improvements in students' learning of programming skills, but does not draw any conclusions about students' usage of strategies during the process.

Design-based exercises have been applied in the design space (Falkner, Vivian, and Falkner, 2014). The purpose of these exercises were to give students the opportunity to practise SRL design and planning strategies (Falkner et al., 2015). For the study, students were given two compulsory exercises that encouraged students to describe their software development processes. Students involved in this study had 1-2 prior programming courses. The results from analysing the answers to these exercises showed that more scaffolding assistance is needed to support students' mastery of the SRL strategies as they progress through their studies.

A framework, *Multi-Faceted SOLO Taxonomy* (Castro and Fisler, 2017), was developed to track the design skills of novices over the duration of a course. The framework aligns with functional programming courses and mirrors the progression of the SOLO taxonomy (Biggs and Collis, 1982). The development of the framework included a methodology for educators to develop and apply assessments using the framework. Another framework was developed with problems for 'cross-linguistic studies of plan compositions' (Fisler, Krishnamurthi, and Siegmund, 2016, p. 215). This framework was developed to assist plan-composition studies, focusing on ordering tasks to form a programming solution. The framework included exercises to refine students' plan-composition skills.

This section presented related work that supports the learning and use of design strategies. The related work demonstrated different approaches used to support design strategies, through learning tools, guidance from educators, and activities.

## 3.5 Cognitive Apprenticeship

This section presents related work in Computer Science that uses Cognitive Apprenticeship, the teaching method used in the Codification Pedagogy that helps students practice difficult tasks through demonstrations modeled after experts' behaviours (Caspersen and Bennedsen, 2007). The background literature on Cognitive Apprenticeship was presented in Section 2.5.1, providing an overview of the core teaching methods within the Cognitive Apprenticeship. Some of the related work showed Cognitive Apprenticeship applied in classroom and online learning environments. Classroom application includes Extreme Apprenticeship (Vihavainen, Paksula, and Luukkainen, 2011b), a method of Cognitive Apprenticeship that adopts aspects of Extreme Programming (Beck and Andres, 2004) and the teaching method. The related work presented in this section focuses on prior work applying Cognitive Apprenticeship to tools used in the digital space. Focusing on tools within the digital space can provide guidance in the design of the Codification Pedagogy, which is applied in online learning environment.

Within the CS problem space, Cognitive Apprenticeship has been applied to Intelligent Tutoring Systems (ITS). Cognitive Apprenticeship was used within an Intelligent Tutoring Systems developed by Aleven et al. (2016) that provides a 'step-by-step guidance during (moderately) complex problem solving' (Aleven et al., 2005). Cognitive Tutor Authoring Tool (CTAT)/TutorShop (Aleven et al., 2016) is another ITS, applying the Cognitive Apprenticeship teaching method to the edX online teaching environment. The purpose of this research is to resolve some of limitations using stand-alone Intelligent Tutoring Systems, such as platform dependencies, by leveraging the technologies provided by Massive Open Online Courses (MOOCs). Providing the Intelligent Tutoring System within a MOOC makes the tool available to a wider audience. CTAT provides step-by-step support for CS assignments, and is the first cognitive tutor to be integrated into MOOCs, using the feedback features from the online learning environment. The study provides guidelines for using CTAT, providing a case study, but no results from applying CTAT to the the course.

QBasic (Dadic, Stankov, and Rosic, 2008) is another example of an Intelligent Tutoring System using Cognitive Apprenticeship. QBasic models the teaching method by comparing novice and expert solutions. This approach helped students learn programming semantics and problem deconstruction. QBasic was found to be successful in teaching students skills, and received satisfactory feedback from the students.

Jin is a CS1 pre-programming analysis tutor (Jin, 2008), containing a cognitive model that provides students a solution path to help them through solving a task. Jin provides corrective feedback when students deviate from the path for solving the task. Though Jin has a higher completion rate of programming problems, the development of the cognitive model is time-consuming for educators, due to the number of programming problems, learning objectives, and the size of the cohort.

The tools presented in this section use the Cognitive Apprenticeship *Modeling* and *Scaffolding* core teaching methods. A potential reason for using the *Coaching*, *Articulation*, *Reflection*, and *Exploration* teaching methods might be how they require students to use problem-solving skills. For example, *Articulation* requires metacognitive skill and self-reflection to compare work with others. It is possible for these other core teaching methods to be integrated into the upper division Cognitive Apprenticeship tools after the basic skills are established through use of *Modeling* and *Scaffolding*. This related work has highlighted teaching methods appropriate for a pedagogy supporting students early in their learning of CS problem-solving skills.

## 3.6 Summary

This chapter presented the related work for this thesis. It showed how the theories, teaching methods, and learning skills and strategies have been applied to previous studies. Some of the related work presented in this section builds on the students' prior knowledge to help them learn new skills and strategies. For example, three works presented in Section 3.4 encouraged students to use pseudocode, a notation that is closely aligns with the natural language, to best describe how they plan on solving the problem. Using pseudocode allows the students to articulate the design process with a familiar notation without learning another concept to perform the design process; however, students do not realise the benefits of using pseudocode to bridge the textual description of the problem to the programmed solution. Rather, students view the notation as another programming language they are required to learn (Garner, 2007). Another example of the related work building on students' existing knowledge is prompting self-reflection through the use of questioning. Questioning allows the student to think of what they currently know and how they can apply that to the existing problem space. Investigating how the related work builds on the students' internal knowledge helps with the development of the Codification Pedagogy, ensuring that the learning activities developed within the pedagogy attempt to build on what the student knows for greater success and understanding.

The related work showed how theories in program comprehension and SRL strategies were applied using active learning to support students in building skills to better understand problems. The related works also showed learning activities modeling expert behaviour, to demonstrate best practices in problem solving. The related work presented in this section target certain design process skills, such as comprehending, analysing, and reflecting. The Codification Pedagogy investigates bringing more of these skills together to encourage students to perform the workflow within the design process, over exercising design skills in isolation.

The Codification Pedagogy is influenced by the teaching principles in these works, encouraging active learning and modeling best practices. The next chapter, Chapter 4, presents how the theories are applied, forming the learning activities within the Codification Pedagogy. Chapter 4 describes a high-level view of the pedagogy's design, development, and evaluation, building on the findings from the related work presented in this chapter.

# Chapter 4

# Pedagogy Studies

This chapter presents an overview of the pedagogical design, and introduces the studies performed in this thesis. This chapter is organised as follows. Section 4.1 is the overview of the chapter. Section 4.2 provides a high-level introduction to the pedagogy. Section 4.3 presents the overarching context for the studies. Section 4.4 provides an overview for the pedagogical design. Section 4.5 presents commonalities for the students participating in the study. Section 4.6 presents the summary.

## 4.1 Overview

This thesis attempts to encourage CS1 students to use Good Learning Behaviours (GLBs) when solving procedural programming assignments. To help students use Good Learning Behaviours, a design process pedagogy is explored as a teaching method, supporting pedagogical goals to improve students' program comprehension and encourage their use of Self-Regulated Learning (SRL) strategies. In Chapter 2, critical thinking skills and design strategies are identified as being important to helping to achieve these pedagogical goals. Chapter 2 presents the Cognitive Apprenticeship theory as the teaching method for introducing the instructional content and demonstrating to CS1 students how an expert would approach solving a programming problem. Chapter 2 also describes the layering of the pedagogical design over a problem-solving process model (Gick, 1986), shown in Figure 2.2. Layering the pedagogy over the model gives students the opportunity to practice the problem-solving process strategy in the scaffolded learning environment, using the Cognitive Apprenticeship teaching method.

This chapter provides an overview of the studies involved in designing and examining the Codification Pedagogy. The studies describe the design and creation of learning activities that compose the Codification Pedagogy. Separate studies are composed to design and evaluate the learning activities, along with evaluating the pedagogy in its entirety.

## 4.2 Pedagogy Introduction

Based on the the research presented in previous chapters, good learning design can engage students' program comprehension, Self-Regulated Learning (SRL) strategies, critical thinking skills, and design strategies. This thesis proposes an instructional approach grounded in Cognitive Apprenticeship that aims to guide students through the design process for CS1 procedural programming assignments. The *Codification Pedagogy* provides CS1 students a scaffolded learning environment to build effective skills and strategies for the software design process, by modeling an expert's problem-solving approach through this process. The Codification Pedagogy

> Write a program that repeatedly reads in a positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average.

FIGURE 4.1. Original *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983)

is comprised of three key learning activities, designed to help students through the design process: an *Assignment Presentation* that contains the programming problem, a *Questioning Activity* that helps the students reflect on the problem, and a *Design Strategy Activity* that supports the students in organising a plan for implementing a programming solution.

## 4.3 Context

This section describes the overarching context for the thesis studies. The information presented in this section comprises commonalities for all studies. Any study introducing custom attributes to the study's context is described within its study chapter.

The research into the Codification Pedagogy was conducted over three semesters in a 12-week Introductory Programming course offered at the University of Adelaide. Students enrolled in the course were new to software development. The Introductory Programming course was a blended learning environment, composed of a classroom lecture and a supervised computer lab that met once a week. This course has a large cohort, ranging from 100 to 250 students each semester. During the lab environment, two tutors were available to provide students guidance and assistance when they seem assistance. This course used Canvas, a Learning Management System (LMS), administering the course's reading materials, instructional videos, assessments, practical programming assignments, and the instructional instruments for this research. The procedural programming language used in this course is Processing.js (Fry and Raes, 2018), a JavaScript variant of Processing that enabled students to quickly develop graphic-based outcomes with minimal programming experience (Godwin-Jones, 2010).

The Codification Pedagogy is designed for practical procedural programming assignments that are available a web page presented within the LMS. The Codification Pedagogy is also available on the same web page presenting these practical procedural programming assignments. The teacher administering the course is responsible for making the assignments available before the assignment's start date, and it remains accessible to the students throughout the semester.

The studies performed in this thesis use Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983) as the problem's context. This is a well-known problem, originating in a 1983 publication. The problem is commonly used in CS1 research (de Raadt, Toleman, and Watson, 2004; Fisler, 2014; Lakanen, Lappalainen, and Isomöttönen, 2015; Simon, 2013) and originally describes four programming tasks in two sentences, shown in Figure 4.1.

1. Counting the number of positive integers entered by the user,
2. Summing those positive integers,
3. Stopping the the sentinel (99999) is encountered, and
4. Averaging the values entered.

| Assignment Name | Week Released | Semester | |
|---|---|---|---|
| | | Feb 2018 & 2019 | Aug 2018 |
| A4.1 | 5 | Appendix A.1.1 | Appendix A.2.1 |
| A4.2 | 5 | Appendix A.1.1 | Appendix A.2.1 |
| A5.1 | 10 | Appendix A.1.2 | Appendix A.2.2 |
| A5.2 | 10 | Appendix A.1.2 | Appendix A.2.2 |
| A6.1 | 11 | Appendix A.1.3 | Appendix A.2.3 |
| A6.2 | 11 | Appendix A.1.3 | Appendix A.2.3 |
| A7.1 | 12 | Appendix A.1.4 | Appendix A.2.4 |
| A7.2 | 12 | Appendix A.1.4 | Appendix A.2.4 |

TABLE 4.1. List of Assignments Used in the Research

The prior studies with the *Rainfall Problem* inform this research, contributing to the design of some of the study methods. Any adjustments to the *Rainfall Problem* are explained in the study chapters.

The introductory course has eight practical programming assignments containing between two and four programming exercises. The assignments focus on procedural programming because the introductory course teaching CS concepts in the procedural programming paradigm with Processing.js.

The interventions for the Codification Pedagogy is introduced in the fourth practical programming assignment, Assignment 4, released at week 5 of the semester. The presentation of the programming assignments, discussed in Chapters 5, 6, and 7, is for all the assignments, starting with Assignment 1, released at week 2 of the semester. A selection of programming exercises within the practical programming assignments contain the Codification Pedagogy. Table 4.1 presents the exercises containing the Codification Pedagogy, with the name of the exercise and the week the assignment was made available to students. The table also contains references to the assignments within Appendix A, listing the assignments administered over the three semesters, where the students enrolled in the first (February 2018) and third (February 2019) semesters received the identical assignment, while the second semester (August 2018) received an isomorphic version of the programming assignments. The isomorphic problems presents the learning objectives in the same way, so that the learners develop the solutions in the same problem-solving paths. For the remainder of this thesis, the exercises are referenced as 'Assignment'; for example, 'Assignment A4.2' denotes the second exercise within the fourth practical programming assignment.

## 4.4 Pedagogy Design

This section introduces the learning activities that comprise the Codification Pedagogy. It is described at a high level, providing a description of the activity, its learning objectives, and its interaction with other learning activities. The roles and interactions of the learning activities form a problem-solving approach modeled after an expert's process model (Gick, 1986).

To model an expert's problem-solving approach, the pedagogy is layered on a problem-solving process model, shown in Figure 4.2. This model is described in Section 2.5.2, providing background on the model and describing the states of the model in greater detail. The pedagogical design has the learning activities layered over the model for the purpose of supporting students' problem-solving processes in

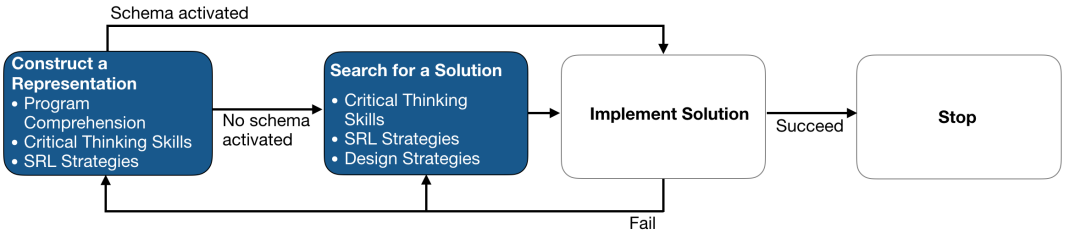



FIGURE 4.2. Placement of Pedagogical Goals and Strategies in Problem-Solving Process

a scaffolded learning environment. Figure 4.2 highlights the problem-solving states and learning activities that are driven by the pedagogy. The figure contains a list of pedagogical goals within each state, which include learning goals and strategies supported in the state. The two problem-solving process states supporting the Codification Pedagogy are *Construct a Representation* and *Search for a Solution*. *Construct a Representation* is the first state that takes into account the student's view of the problem, where the student develops a mental model of a solution. The state supports the pedagogical goal to improve students' program comprehension. *Search for a Solution* is the second state, where the student determines the best approach to solve the problem. This state represents the design process that can involve critical thinking skills and use of SRL strategies to solve the problem. The pedagogy is designed to support students in this state by giving them learning activities to support the design process.

Figure 4.2 shows the transition between states for the problem-solving process model. The transitions allow students to return to prior states during the problem-solving process. Like the model, the pedagogy is designed to allow students to transition between the problem-solving process states within the pedagogy's scaffolded learning environment. For example, the learning activities work together to help the student identify the problem's goals in the *Construct a Representation* state, and help them identify a plan to solving the problem from these goals within the *Search for a Solution* state.

The design of the Codification Pedagogy is grounded in Cognitive Apprenticeship. Cognitive Apprenticeship was introduced in Section 2.5.1, presenting its teaching methods: *Modeling*, *Coaching*, *Scaffolding*, *Articulation*, *Reflection*, and *Exploration*. The purpose of these teaching methods is to support students' acquisition of knowledge and skills. The Codification Pedagogy applies the *Scaffolding* teaching method to provide the learner guidance, specifically *metacognitive scaffolding*, a type of scaffolding that does not focus on the problem content, but guides the learning approach for solving the task (Roll et al., 2007). Metacognitive scaffolding has been shown to have a positive influence on the learner's problem-solving process 'by helping them set goals and deadlines, engage in research, organize their ideas and thoughts, correct misunderstandings, revise ineffective plans or strategies, avoid procrastination, use time effectively, and monitor and evaluate their progress' (An and Cao, 2014).

Figure 4.3 is a workflow of the Codification Pedagogy, layered on a problem-solving process model. The figure includes the corresponding research methods and learning activities used to support the problem-solving state. The three learning activities are an *Assignment Presentation*, a *Questioning Activity*, and a *Design Strategy Activity*. The figure shows transitions between the learning activities, to demonstrate the workflow. The workflow shows that the learning activities allow the student to return or skip activities, giving the student the opportunity to form their own

problem-solving process based off an expert's model, or use the problem-solving process suggested by the pedagogy. More details on the research studies and learning activities are presented in the remainder of this section.

### 4.4.1 Assignment Presentation

The *Assignment Presentation* is presented as the first learning activity in the pedagogy. This learning activity contains the programming assignment within a scaffolded learning environment. The activity is designed to help students improve their program comprehension, helping them form a more accurate model for solving the problem. The design and development of this activity is grounded in **Program Comprehension**, described in Section 2.2.1.

### 4.4.2 Questioning Activity

The second learning activity is a design-based intervention that uses instructional questions to encourage self-reflection, a behaviour present in high-performing students (Hausmann and Chi, 2002). The *Questioning Activity* is designed to engage the students' lower and higher-order **Critical Thinking Skills**, described in Section 2.3. The learning activity gives students different views into their existing internal knowledge to help them plan and organise a solution. The *Questioning Activity* is the first intervention that follows the *Assignment Presentation*. The organisation of the learning activities within the pedagogy allows students to reference the problem description when answering the questions.

### 4.4.3 Design Strategy Activity

The final learning activity is a *Design Strategy Activity* that supports students to build their **Design Knowledge**. The activity uses Parsons problems, a learning tool that helps reduce students' cognitive load when creating working programs, by arranging code fragments in the right order (Morrison, Margulieux, and Guzdial, 2015). Reasons for using Parsons problems as the learning activity are presented with this study. This activity resides after the *Questioning Activity*, where students use the *Assignment Presentation* and their answers to the *Questioning Activity* as references for this intervention.

## 4.5 Pedagogy Studies

This section describes the studies performed on the Codification Pedagogy. Six studies are performed to measure how the pedagogy achieves its pedagogical goals.

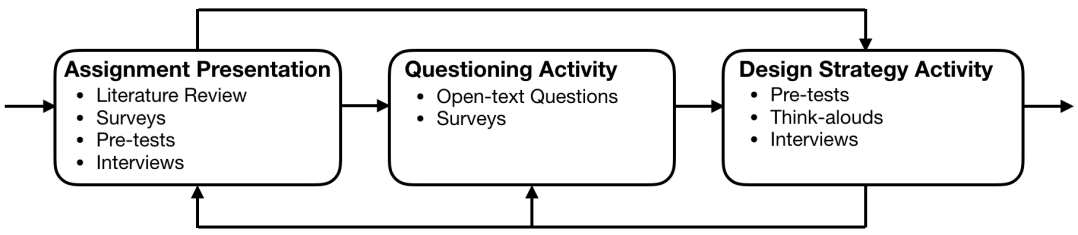



FIGURE 4.3. Research Studies for Learning Activities

Qualitative studies using a mixed-methods approach of pre-tests, surveys, questionnaires, think-alouds, and interviews are conducted to collect students' perceptions on using the pedagogy, to gain insight into how they perceive the pedagogy helps them with their understanding of the problem. The qualitative studies underwent ethics approval at the University of Adelaide. Quantitative analysis is performed using different analysis approaches, such as comparative analysis, to confirm that the data aligns with the hypothesis that the pedagogy will improve students' understanding of the problem, and support their use of SRL strategies.

The six studies are presented as separate chapters. The studies are organised in the order the learning activities are presented to students within the pedagogy. The first five studies examine the learning activities, while the final study evaluates the pedagogy in its entirety. The rest of this section describes each study.

### 4.5.1 Assignment Design Study

Chapter 5 describes the study performed to identify best practices in CS1 assignment design. The study involves a survey of literature seeking evidence-based design treatments proven to help students' understanding of programming assignments. The study constructs a framework with the design treatments from peer-review publications. This study shows the framework being used to construct assignment presentations for procedural programming problems. This study contributes towards answering the research question:

- ***RQ 1.2**: What presentation treatments within the programming assignments support students in their understanding of the problem?*

From the study, an instructional instrument was constructed using the framework, and used in the next study for further examination of the assignment presentation.

### 4.5.2 Assignment Comparative Study

Chapter 6 describes a study that uses a comparative study method to examine the assignment presentation formed in the first study. The assignment presentation is a highly scaffolded version of Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983) that allows for comparison with other *Rainfall Problem* studies. The study compares the completion rate of programming tasks, to determine whether the design treatments help students complete tasks. The study addresses the following research questions:

1. ***RQ1.1**: How does scaffolding the assignment presentation influence the student's ability to identify goals and subgoals necessary to complete a procedural programming assignment?*
2. ***RQ1.2**: What presentation treatments within the programming assignments support students in their understanding of the programming problem?*

The results from this study provide insight into helpful design treatments; however, this study is performed with just one programming problem. The next study continues to evaluate the *Assignment Presentation* learning activity by using a qualitative study with another programming problem.

### 4.5.3 Assignment Design Interview Study

Chapter 7 presents a qualitative study using a questionnaire and interviews with CS1 students to evaluate the design treatments within a programming assignment. This study involves collecting student volunteers' feedback on how the presentation impacts their understanding of the problem and their use of SRL strategies. The interviews provide insight into how students are using the design treatments to help them solve the problem. The study contributes towards answering the following research questions:

- ***RQ1.1**: How does scaffolding the assignment presentation influence the student's ability to identify goals and subgoals necessary to complete a procedural programming assignment?*
- ***RQ1.2**: What presentation treatments within the programming assignments support students in their understanding of the programming problem?*

The study strengthens the findings from the previous study on assignment presentation. The results help to identify design treatments that can help students better understand the problem and support their use of SRL strategies.

### 4.5.4 Questioning Activity Study

Chapter 8 examines the first intervention in the Codification Pedagogy, the *Questioning Activity*. The activity builds on the *Assignment Presentation* by encouraging students through questioning to use critical thinking skills to reflect on the programming problem. The *Assignment Presentation* activity is designed to help students form a better mental model from the assignment description, while this learning activity helps the student apply their internal knowledge to solving this mental model. The study presents the development of a questioning framework to construct questioning activities. The framework maps 23 instructional question types to Bloom's Taxonomy. The Bloom's Taxonomy is an analysis tool to determine if the questioning helps students to support students in engaging with the design process through questioning. This study is designed to answer the following research question:

- ***RQ2.1**: Does encouraging questions in an online CS1 learning environment promote the expected cognitive levels from students when answering the questions?*

This study seeks to identify the cognitive levels used by students when answering the *Questioning Activity*, supporting the decomposition of programming goals to help them better solve the problem. Results demonstrate students reflecting on the problem and using Good Learning Behaviours (GLBs) to plan a solution for the programming problem.

### 4.5.5 Design Strategy Activity

Chapter 9 presents a study on the second intervention in the pedagogy, a *Design Strategy Activity*. The learning activity supports students' use of design strategies by having them arrange plans in the order of implementation. The learning activity incorporates Parsons problems, a learning tool that enables students to construct working programs by arranging code fragments in a scaffold environment (Parsons and Haden, 2006). In this study, students arrange plans in the Parsons problems, to order the plans for implementation.

The study uses qualitative and quantitative study methods to show how students use the learning activity during the design process. This study is designed to answer the following research questions:

- ***RQ3.1**: How do students use Parsons problems during the design process for solving CS1 procedural programming assignments?*
- ***RQ3.2**: What Self-Regulated Learning (SRL) strategies are support by Parsons problems used as a design-based intervention for programming assignments?*

Results from the study show SRL strategies being supported by the learning activity, and students using the activity throughout the software development process.

### 4.5.6 Entire Pedagogy Study

Chapter 10 focuses on the entire pedagogy, evaluating how the entire pedagogy might have influenced students' awareness of skill usage and their academic success. This study applies quantitative methods to compare the influence of the entire pedagogy with the *Questioning* and *Design Strategy* activities, to students that did not receive the activity. Results from the study show students exposed to the pedagogy using Good Learning Behaviours (GLBs) to plan and understand the problem, along with having higher completion rates for programming assignments.

## 4.6 Summary

This chapter presented a high-level view of the pedagogy design, and the studies examining the pedagogy. The pedagogy design outlines the learning activities used within the pedagogy, and identifies the pedagogical goals each learning activity supports.

The chapter described the pedagogical design layered over a problem-solving process model. The pedagogical design allows students to transition between the learning activities, similar to transitioning between problem-solving process states. The pedagogy is designed to support students to transition through the problem-solving process states, helping them readjust their understanding of the problem and supporting their use of problem-solving strategies within a scaffolded learning environment.

The evidence of the studies demonstrate that the pedagogy encourages students to use SRL strategies and helps improve their program comprehension. The studies also confirm that the Codification Pedagogy has a positive impact, revealing students' adjustments in their understanding of the problem through self-reflection, their perception of their understanding of the instructional materials, and their use of SRL strategies.

The study chapters identify which part of the pedagogy each study examines and how that study's findings relate to the other studies. Each chapter also provides the study's methodology, the experiment, the results from the experiment, and a discussion summarising the study's findings.

# Chapter 5

# Assignment Design Study

This is the first study on the Codification Pedagogy, focusing on the *Assignment Presentation* learning activity. Section 5.1 provides an overview of the study. Sections 5.2, 5.3, and 5.4 present the study method, analysis, and results. Finally, Section 5.5 presents the summary.

## 5.1 Overview

As stated in the background literature, Chapter 2, this thesis is motivated to encourage students to adopt Good Learning Behaviours (GLBs) so that they would avoid using Poor Learning Tendencies (PLTs). This study evaluates the assignment presentation as a means of promoting Good Learning Behaviours and improving students' program comprehension (Schulte et al., 2010), which could lead to higher assignment completion rates (Bergin, Reilly, and Traynor, 2005). Practitioners have the opportunity through the assignment design to discourage the adoption of Poor Learning Tendencies, such as focusing on surface aspects (Adelson, 1984) or keywords (Ginat, 2003) within the assignment description that could lead to misunderstandings. Prior assignment design research has focused on the inconsistencies in the presentation of assessment tasks (Thompson et al., 2008) and assignments for software engineering courses (Hashim and Khairuddin, 2009; Khairuddin and Hashim, 2008), but this research focuses on the assignment design and how it can be used to present CS1 programming assignments for better understanding, helping students to follow instructions with success and with more ease (Marcus, Cooper, and Sweller, 1996).

Assignment design can influence students' learning, where inappropriate forms of presentations, such as overly complicated assignment descriptions, can lead to the adoption of surface-level learning approaches (Joughin, 2010). Poorly constructed assignments require more understanding (Reitman, 1965), which could 'impede students from acquiring new concepts in computer programming' (Veerasamy, D'Souza, and Laakso, 2016, p. 51) and affect how they perceive their programming abilities, potentially leading to lower assessment scores (Ramalingam, LaBelle, and Wiedenbeck, 2004). Well-formed assignments can help CS1 students complete assignments (Feldman and Zelenski, 1996) and develop problem-solving skills (Fee and Holland-Minkley, 2010).

This first study builds on the background literature, by seeking evidence-based support in text-based representations of problems to improve students' program comprehension. The study presented in this chapter helps towards answering the following research question:

- *RQ1.2: What presentation treatments within the programming assignments support students in their understanding of the problem?*

| Inclusion Criteria | Exclusion Criteria |
|---|---|
| • Peer-reviewed papers published in journals and conferences between 2000-2017.<br>• Papers on programming assignment design for CS1 students to solve independently.<br>• Assignments developed for classroom and online learning environments.<br>• Papers that identify guidelines that the authors feel helped students better understand the programming problem. | • Studies focused on assignments and learning strategies for students beyond their first year at university.<br>• Purely theoretical papers.<br>• Research examining changes to curriculum for higher learning gains.<br>• Literature providing guidelines for either lab- or team-based instructions.<br>• Research focusing on a single programming concept, where guidelines cannot be generalised nor applied to other contexts. |

TABLE 5.1 Literature Review Selection Criteria

To address this research question, this study surveys existing peer-reviewed publications that identify design treatments for instructional materials that positively contribute to students' understanding of the programming problem, or give students support in using Self-Regulated Learning (SRL) strategies to help them better understand.

The study method collates design treatments found in peer-review publications into a framework. The framework is designed to help construct the presentation of CS1 programming assignments. This chapter presents an example assignment presentation, formed by using the framework.

## 5.2 Methods

This section describes the study method used to find design treatments that can improve students' program comprehension through a text-based representation of the programming problem. A thorough and systematic literature review (Booth, Papaioannou, and Sutton, 2012) is conducted to find design treatments in the existing literature that can help improve students' mental representation of the solution. A systematic literature review identifies and collects research based on a defined criteria, where the collected data is then analysed and presented. This section presents the literature review processed used in this study. Section 5.2.1 describes the search criteria for finding the design treatments. Section 5.2.2 describes the data collection performed.

### 5.2.1 Search Criteria

The systematic literature review began with a search using a meta-search engine provided through the university, which queries the ACM Digital Library, EBSCOhost for Computers and Applied Science, IEEE Xplore Digital Library, Springer, and SAGE Publications libraries. The search criteria, shown in Table 5.1, focus on publications in peer-reviewed journals and conferences between 2000 and 2017 that discuss design treatments for individual take-home CS1 programming assignments, promoting improved program comprehension. As well as being the focus of the

search results, design treatments also include authors' observations made from performing the study's experiment, case study, or survey.

The applied selection criteria use the following search string: (*Homework* OR *Exercises* OR *'Problem Description'* OR *Assignment* OR *Comprehension*) AND (*CS1* OR *'Computer Science'* OR *'Introductory Programming'*), where the meta-search engine applied the query string for all search fields, such as *Title*, *Subject*, and *User* tags. The keywords used in the search criteria were selected from terms used in publications (Kinnunen and Simon, 2011; Köppe and Pruijt, 2014) focusing on assignment design.

### 5.2.2 Data Collection

This section describes the process of selecting publications from the study's search criteria, and collating them using the multi-level assignment educational design pattern to form the assignment presentation framework. Part of the data collection process is applying exclusion criteria from the search criteria described in Section 5.2.1. The exclusion criteria, presented in Table 5.1, contain five rules defining the exclusion requirements.

To illustrate the selection process with the exclusion criteria, a couple of example publications are presented. A peer-review journal paper (Buckley et al., 2004) appeared in the initial search results, where the study discussed socially-relevant content for assignments using three exemplar projects. However, the paper does not meet the selection criteria because the example projects are designed for senior-level undergraduate courses, and not CS1. Another example (Sharmin et al., 2019) investigates using open-ended assignment presentations for the purpose of improving CS1 students' self-efficacy. This paper did not meet the selection process because the study uses design treatments to improve self-efficacy.

After the inclusion and exclusion criteria are applied, a quality evaluation step (Booth, Papaioannou, and Sutton, 2012) is performed, to ensure the appropriate publications are selected. Abstracts from the publications are reviewed to determine if the design treatments influence students' program comprehension.

## 5.3 Analysis

This section describes the analysis process for this study. Section 5.2.2 presents the selection process for the design treatments. Section 5.3.1 describes the method of organising the data collected from the search criteria to form a framework, incorporating the design treatment into the assignment design framework.

### 5.3.1 Framework Design

A framework was chosen to organise and classify the publications collected from the systematic literature review. This section describes how the design treatments are classified to form an assignment design framework that can be used to construct programming assignments. The publications identified through the systematic literature review are classified using a theoretical multi-level assignment educational design pattern that incorporates 'a variety of educational objectives into a single assignment by including the concepts on multiple knowledge and process levels' (Köppe and Pruijt, 2014). The design pattern is a variant pedagogical design pattern (Alexander, Ishikawa, and Silverstein, 1977) for educational activities that can improve applications for teaching and learning (Fioravanti and Barbosa, 2016).

Reliability of the literature review was considered by using strategies applied to systematic literature reviews that mitigate bias and increase reliability (Haddaway et al., 2015). Multiple databases were used in the search to avoid any bias that might arise from finding literature within a single database. Another strategy used was designing 'search strings with appropriate synonyms and combinations of search terms that the relevance of all search results be determined based on consistent criteria' (Haddaway et al., 2015).

This design pattern was selected to classify the design treatments, because it contains layers of scaffolding. *Scaffolding* is a teaching method used within Cognitive Apprenticeship, designed to help support students' learning by providing students a learning environment to successfully complete tasks (Wood, Bruner, and Gail, 1976). Background on Cognitive Apprenticeship was presented in Section 2.5.1. The design pattern's ability to support scaffolding aligns with the instructional strategy used in the Codification Pedagogy.

The multi-level assignment educational design pattern is comprised of the following sections:

1. *Context*: Describes the conditions in which the design pattern is used that can help facilitate deeper learning.
2. *Problem*: Attempts to support the learning experience for the student by focusing on the presentation of materials, and presents examples and requirements to help them for higher conceptual understanding.
3. *Forces*: Clarifies the problem by describing tactics used by students to complete the assignment.
4. *Solution*: Describes how to solve the pattern's *Problem* by encouraging students to realise the concepts by integrating them into the assignment domain.
5. *Implementation*: Provides three examples on how to apply the design pattern.
6. *Consequences*: Discusses the risks that may arise from using the design pattern.

This study uses the *Implementation* Section as the template for the assignment presentation structure. The *Implementation* Section identifies three categories to help organise the design treatments:

- *Context*: Provides the basis and explanation for the programming problem (Guzdial, 2010) and displays required concepts for completing the assignment. The concepts are presented early in the assignment description to encourage reflection on the concepts.
- *Problem Description*: Describes the problem needing to be solved using the concepts identified in the *Context* Section.
- *Hints*: Depicts concepts in other knowledge levels to help the student better understand the problem and concepts.

To determine the best category for a design treatment, the purpose of the treatment is examined. The purpose is then mapped to one of the three categories in the design pattern: *Context*, *Program Description*, or *Hints*. If the initial examination suggests the treatment can be mapped to more than one category, the publication introducing the treatment is further examined, providing additional background for selecting the appropriate category.

The classification of the design treatments within the design pattern forms an assignment presentation framework. This study demonstrates framework usage by applying the framework to a programming problem.

## 5.4 Results

The study's search criteria generated 2169 results, but was reduced to 92 papers when the meta-search engine's *Peer-reviewed* filter was applied. The *Peer-reviewed* filter was previously applied to another search criteria outside the scope of this study, which produced in an accurate search result. The exclusion criteria from Table 5.1 was applied to the 92 papers' abstracts and titles. The exclusion criteria reduced the 92 papers to 11 papers. The citations from the 11 papers were evaluated to determine if any of the citations were relevant for this study. The citations from the 11 papers brought the final literature result to 14 publications.

Table 5.2 presents the 14 publications that meet the search criteria, listing the publications in alphabetical order by title, identifying the design treatments from the publication. The 14 publications were then divided into three categories, *Context*, *Program Description*, and *Hints*. These categories emerged from the multi-level assignment educational design pattern (Köppe and Pruijt, 2014), presented in Section 5.3.1 and used as the assignment presentation template. The remainder of this section presents the design treatments organised by the three presentation categories, followed by the developed framework and an exemplar assignment constructed by the framework.

### 5.4.1 Context

The search criteria identify three publications supporting context. The assignment design pattern suggests including the learning objectives and concepts required to solve the problem, promoting internal reflection on previously learned materials. The design pattern also suggests placing key concepts in bold to focus the student and discourage them from concentrating on secondary or superficial concepts (de Raadt, Watson, and Toleman, 2009).

Two of the publications focus on approaches to minimise Poor Learning Tendencies (PLTs) (Carbone et al., 2000; Carbone et al., 2001). The first study targets Poor Learning Tendencies related to superficial attention, impulse attention, and staying stuck, while the second study focuses on Poor Learning Tendencies related to non-retrieval and lack of internal and external reflective thinking. The second study (Kussmaul, 2008) presents lessons learned from scaffolding CS1 and CS2 Java-based assignments, presenting best practices that helped with their understanding of the programming problems. Design treatments from the literature include:

- Emphasising key points of the problem by highlighting the tasks to help reduce students making hasty problem-solving decisions (Carbone et al., 2000).
- Avoiding the presentation of numerous unfamiliar concepts in a single assignment (Carbone et al., 2000). The tutors observed students could only focus on one question when encountering a lot of new concepts.
- Containing familiar tasks, such as prior learning materials, which reinforces learning (Carbone et al., 2001).

| Title of Publications | Suggested Design Treatments |
|---|---|
| *A code snippet library for CS1* (Lorenzen et al., 2012) | Includes external code in the assignment to help students learn more complex concepts sooner, and allows practitioners to build interesting problems. |
| *Characteristics of programming exercises that lead to poor learning tendencies: Part II* (Carbone et al., 2001) | Includes tasks that students are familiar with to promote learning reinforcement; Present high-level justification to link previously learned concepts. |
| *Developing real-world programming assignments for CS1* (Stevenson and Wagner, 2006) | Incorporates layers in the assignment to challenge students and make the problem interesting and fun for them to complete. |
| *Experiences in threading UML throughout a computer science program* (Ruocco, 2001) | Includes use cases, class diagrams, and associations within the assignment design to provide better examples of how control structures can be applied to the problem. |
| *Exploring factors that include computer science introductory course students to persist in the major* (Barker, McDowell, and Kalahar, 2009) | Provides concepts with context that is interesting to students; provide instructional material related to students' background knowledge, for better retention. |
| *Extreme apprenticeship method in teaching programming for beginners* (Vihavainen, Paksula, and Luukkainen, 2011a) | Provides smaller goals and relevant examples with clean guidelines to help students find the starting point to solve the problem. |
| *Interesting basic problems for CS1* (Gal-Ezer, Lanzberg, and Shahak, 2004) | Provides realistic problems to help increase motivation. |
| *Note to self: Make assignments meaningful* (Layman, Williams, and Slaten, 2007) | Provides socially-relevant material that students find interesting to keep them more engaged in solving the problem. |
| *Novice programmers and the problem description effect* (Bouvier et al., 2016) | Includes contextualisation that can help with motivation and engagement. |
| *Principles for designing programming exercises to minimise poor learning behaviours in students* (Carbone et al., 2000) | Emphasises the key point of the problem by using bold text; minimises overloading the student with unfamiliar concepts, so they can remain focused. |
| *Scaffolding for multiple assignment projects for CS1 & CS2* (Kussmaul, 2008) | Provides design elements from lessons learned in the classroom. |
| *Research directions for teaching programming online* (Settle, Vihavainen, and Miller, 2014) | Presents published materials to construct acceptable practices to teach in online learning environments. |
| *Visualisations in preparing for programming exercise sessions* (Ahoniemi and Lahtinen, 2007) | Includes visualisations into the problem that can help students with no programming experience better understand the problem. |
| *Why the rhetoric of CS programming assignments matter* (Wolfe, 2004) | Provides real-world context for the problem to keep students motivated. |

TABLE 5.2 Publications Meeting Literature Review Selection Criteria

- Providing a broad view of the assignment to give it purpose and to help students associate the concepts taught in the course with the problem (Carbone et al., 2001).
- Providing an outline on how to approach solving the problem to help students stay on the right problem-solving path (Carbone et al., 2000).
- Providing the students with the requirements (Kussmaul, 2008). The early assignments provide more detailed requirements and fade the support over the duration of the course.
- Interleave assignments to provide a variety to the programming problems, while giving students the opportunity to return to provide work and incorporate the previous feedback by the educator (Kussmaul, 2008).

### 5.4.2 Program Description

The results from the study's search criteria contain seven publications focusing on the assignment's purpose. Three of the publications (Bouvier et al., 2016; Settle, Vihavainen, and Miller, 2014; Vihavainen, Paksula, and Luukkainen, 2011a) discuss providing students with contextualised problem descriptions so they can develop a cognitive schema by using their internal knowledgebases. The familiarity with the contextualised content can free space in the students' working memories so that they can develop more complex solutions (Sweller, Merriënboer, and Paas, 1998). A study (Bouvier et al., 2016) from the literature review performed a study on the impact of contextualised content on students' grades. Though the contextualised content did not have an impact on the overall grades, it did have an impact on the participants' motivation and interest in the assignment. Contextualisation has also been studied from the perspective of students' motivation to persist with the problem-solving process. Another study studied eight factors that predicted students' persistence in CS (Barker, McDowell, and Kalahar, 2009). One factor was related to contextualised content, showing students persisting in solving programming assignments, and also enabling them to retain more of the information.

Another study (Gal-Ezer, Lanzberg, and Shahak, 2004) found using problems with real-world context can help demonstrate complex concepts in CS1 because of students' familiarity with the context. A study (Layman, Williams, and Slaten, 2007) studied motivation by administering a survey at North Carolina State University with 200 CS1 students, asking for testimonials on practical and socially-relevant assignments. The results showed that students preferred practical and socially relevant assignments, a preference that could increase their interest in the assignment. Another publication (Wolfe, 2004) conducted a survey with 81 students evaluating different contextualised instructional materials. The results from the survey showed a majority (54%) of the students prefer real-world assignments, since they felt the assignments were easier to understand, and the real-world context motivated them to complete the assignment. Another publication (Stevenson and Wagner, 2006) performed a test using two programming assignments with real-world contexts in a classroom environment. The students participating in the study found solving these assignments fun and interesting because they appreciated seeing their work applied to real external websites.

A study performed at the University of Helsinki within a CS1 learning environment and Java using Extreme Apprenticeship, a teaching approach based on principles that guide students in solving programming problems, providing feedback

during the problem-solving process (Vihavainen, Paksula, and Luukkainen, 2011a). The results show reduced drop-out rates. Outcomes using Extreme Apprenticeship produced the following guidelines for presenting assignments:

- Providing students with smaller goals to help them identify intermediate goals.
- Presenting problems that have a defined starting point to assist students in beginning the problem-solving process.
- Giving students examples of the output to better understand what the problem is trying to achieve.

Another publication found the assignment description constructed in multiple layers can provide struggling students with additional support to understand the instructional materials while continuing to challenge high-achieving students (Stevenson and Wagner, 2006). Within the layered assignment, there is a basic solution that all students can achieve, along with additional challenging layers. A layered approach in the assignment presentation gave struggling students the opportunity to succeed by achieving some layers in the assignment (Stevenson and Wagner, 2006). Achieving some layers in the assignment can increase the students' self-confidence and self-efficacy, possibly encouraging them to attempt more challenging layers in future assignments.

### 5.4.3 Hints

The final section of the assignment presentation provides students with additional assistance in understanding the problem. Four publications emerged from the search criteria that align with this section, focusing on visualisation and coding exemplars.

One publication (Ahoniemi and Lahtinen, 2007) investigated the use of a visualisation tool for reviewing instructional materials. The study concluded visual learners and students struggling to understand concepts benefited from using the visual tool, which resulted in higher grades. The 30 students involved in the study were also given a post-survey to provide their experience using the visualisation tool. Students' feedback in the survey showed them stating the visualisation tool helped them better understand the problem. Another publication (Ruocco, 2001) in the literature review uses the Unified Modelling Language (UML) (Object Management Group (OMG), 2017) in assignments administered at the United States Military Academy at West Point, New York. UML components, such as use cases, class diagrams, and associations replaced a previous *ad-hoc* visual language used at the academy. Students using UML valued the visualisation tool because it allowed them to better describe their programs' intentions. The visual aids helped students recognise interactions between objects, identify workflow, and better understand state changes in the problem.

Another publication (Lorenzen et al., 2012) discusses the presence of an external codebase in assignments to encapsulate complex programming concepts that are beyond the students' cognitive abilities. Providing external codebases give students the opportunity to develop more complex and interesting assignments earlier in the course. The study attempts three different approaches to integrate external codebases in CS1 assignments: manuals, memorisation, and libraries. The use of manuals overwhelmed students, and memorisation of code structures was ineffective when students attempted to apply them during the software-development process. The final approach uses a code snippet library comprised of external C++ and Java-based libraries to help reduce students' frustration during the software development

| Context | | | |
|---|---|---|---|
| **Element** | **Headings** | **Core Objectives** | **Context Information** |
| **Practice** | • Helps to identify sections | • Minimises hasty decisions<br>• Identifies tasks | • Gives background for problem<br>• Promotes long-term retention<br>• Deduces practicality |
| **Guidance** | • Provide clean guidelines (Vihavainen, Paksula, and Luukkainen, 2011a) | • Uses bold face text (Carbone et al., 2000) | • Provide basis for assignment (Carbone et al., 2001) |

| Program Description | | |
|---|---|---|
| **Element** | **Assignment Complexity** | **Subject Matter** |
| **Practice** | • Increases confidence<br>• Reduces Poor Learning Tendencies | • Increases motivation<br>• Improves self-efficacy |
| **Guidance** | • Reduces the number of new concepts (Bouvier et al., 2016)<br>• Adjusts incrementally the content (Settle, Vihavainen, and Miller, 2014)<br>• Incorporates smaller goals (Vihavainen, Paksula, and Luukkainen, 2011a)<br>• Encapsulates complexity in external modules (Layman, Williams, and Slaten, 2007) | • Provide interesting (Gal-Ezer, Lanzberg, and Shahak, 2004), socially-relevant (Layman, Williams, and Slaten, 2007) and realistic problems (Wolfe, 2004)<br>• Provide concepts to students with interesting contexts (Stevenson and Wagner, 2006)<br>• Instructional material related to students' known background (Stevenson and Wagner, 2006)<br>• Contextualisation for motivation (Bouvier et al., 2016) |

| Hints | | | | |
|---|---|---|---|---|
| **Element** | **Diagrams** | **Code Segments** | **Examples** | **Visualisation** |
| **Practice** | • Helps with design strategy<br>• Explains software design | • Encourages coding standards<br>• Helps develop algorithms | • Learns from a worked solution | • Provides further insight<br>• Develops mental model |
| **Guidance** | • Include UML diagrams for illustration (Ruocco, 2001) | • Include code segments as references (Lorenzen et al., 2012) | • Include relevant examples (Vihavainen, Paksula, and Luukkainen, 2011a) | • Include visual aids to assist visual learners (Ahoniemi and Lahtinen, 2007) |

FIGURE 5.1 CS1 Assignment Design Framework

process and help students develop programming algorithms. The external codebase exposes students early to experts' code, a teaching approach used in apprenticeship learning (Vihavainen, Paksula, and Luukkainen, 2011a). The last publication (Vihavainen, Paksula, and Luukkainen, 2011a) suggests using apprenticeship learning through examples that give the students the opportunity to interact with interesting problems, which has been shown to decrease dropout rates.

**Averaging Rainfall**

The purpose of this program is to **practice programming elements covered this week** by converting a math formula into a program. Here are the programming elements to use in your assignment:

| | | |
|---|---|---|
| **Variables** | **Loops** | **Input/Output** |
| **Integers** | **If Then/Else** | |

**Program Description:**

**Create a program that averages rainfall for a certain number of days.** Your program first receives the rainfall from the keyboard. The days are entered one day at a time until the user enters the number 99999. 99999 is a signal to stop accepting rainfall information. Then your program should calculate the Arithmetic Mean (Average) of the rainfall. Lastly, it should display the result on the screen in the following format:
*The average rainfall for 'n' days is 'x'.*
The '*n*' is the number of days, and '*x*' is the average of those days.

**Hints:**

- You can use Integers for the rainfall numbers.
- Do not accept negative rainfall, since this does not exist in the real world. Provide a message letting the user know they made a mistake, such as: *Sorry, unacceptable rainfall.*
- You will also need to calculate the Arithmetic Mean (Average) in your program.

$$Average = \frac{x_1 + x_2 + ... + x_n}{n}$$

- For more hints on the math formula, visit: http://mathsisfun.com/data/mean-machine.html

Context
- Title
- Core objectives
- Context information

Assignment Domain
- Assignment complexity
- Subject Matter

Knowledge Embellishments
- Diagrams
- Visualisation

FIGURE 5.2 Example of Highly Scaffolded *Rainfall Problem*

### 5.4.4 Framework

This section presents the assignment design framework constructed from the design treatments identified by the study's search criteria. This section also demonstrates

> Write a program that repeatedly reads in a positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average.

FIGURE 5.3 Original *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983)

how the framework is applied to form a high scaffold version of an existing programming problem.

The framework is shown in Figure 5.1, and is divided into three categories from the multi-level assignment educational design pattern (Köppe and Pruijt, 2014): *Context*, *Program Description*, and *Hints*. In the presentation template, *Context* design treatments appear first, followed by the *Program Description*, and *Hints*.

The prevalence of design treatments within an assignment presentation can define the scaffolding level that best suits the students' abilities. A high number of design treatments across the three categories would indicate a high-scaffold environment, designed for students needing additional guidance in understanding and solving the assignment. Fewer design treatments would indicate a low-scaffold environment, designed for students to solve the problem in an independent learning environment, and practise their problem-solving skills (Knobelsdorf, Kreitz, and Böhne, 2014).

### 5.4.5 Example Assignment Presentation

This section demonstrates the application of the assignment presentation framework to a CS1 problem, using Soloway's *Rainfall Problem* (Soloway, 1986). Background on this problem was presented in Section 4.3. The demonstration constructs a highly scaffolded learning environment from the original problem, shown in Figure 5.3.

Figure 5.2 shows the highly scaffolded version of the *Rainfall Problem* after applying the framework. The assignment presentation was designed for CS1 students with little to no programming experience. The construction involves design treatments from each category in the framework, incorporating a high number of design treatments to provide a high level of support.

On the right side of the Figure 5.2 are lists describing the design treatments applied to the example assignment. In the figure, the *Context* Section presents previously learned programming concepts, to encourage students to use these concepts within their solutions. The *Program Description* Section contextualises the averaging algorithm. The *Hints* Section is a link to a worked example that provides students with a visualisation tool for averaging. The *Hints* Section provides students with the averaging formula to help them engage their existing mathematical knowledge.

## 5.5 Summary

This chapter presents a study performed on the Codification Pedagogy's first activity, the *Assignment Presentation*. This study was designed to identify previously published design treatments, to help improve students' program comprehension. A systematic literature review was used as the study method to collect these design treatments. The study engages an assignment design pattern as a template for structuring the presentation of programming assignments.

The results from the study presented 14 publications that identified design treatments that helped students' understanding of the problem description. The identified design treatments were classified into categories to form an assignment presentation framework. The application of the framework to form a highly scaffolded version of the *Rainfall Problem* demonstrates how the framework can contribute to the field of Computer Science Education. The framework can be used as a guide for educators, enabling them to select design treatments based on their students' abilities.

There are limitations to the study. The keywords used in this study were selected to maximise results, but publications on CS1 assignment design use a broad list of terms for describing assignments and understanding. As a result, the search criteria might have missed some of these keywords, potentially reducing the number of publications.

There are future research opportunities with this study. This study presents a highly scaffolded version of a CS1 programming problem. Future research can examine the fading of support, evaluating how students compensate when support is removed. The example assignment presented in this chapter was developed by the researcher, to evaluate the process of using the framework in the assignment design process. Future research can invite other educators to develop CS1 assignments with this framework. The results can provide insight into areas in the framework needing additional development. The framework developed in this study focuses on CS1 programming assignments. Another future research opportunity would be to evaluate the framework when applied to upper division CS programming assignments.

The next study, described in Chapter 6, builds on the results from this study. The next chapter continues to examine the *Assignment Presentation* learning activity. The next study applies the example assignment constructed in this study as the instructional instrument, using the instrument formed in this study to measure the impact the assignment presentation has on students completing programming tasks.

# Chapter 6

# Assignment Comparative Study

This chapter presents a study that continues to examine the *Assignment Presentation* learning activity. Section 6.1 provides an overview of the study. Sections 6.2, 6.3, 6.4, and 6.5 present the study method, analysis, and results. Finally, Section 6.6 presents the summary.

## 6.1 Overview

A pedagogical goal for this thesis is helping students improve their program comprehension through the problem description. Background literature on program comprehension was presented in Chapter 2, describing the process students take when forming a mental representation of a solution involving the textual representation of the problem (Schulte et al., 2010). This study builds on research in program comprehension by focusing on the textual representation of the programming assignments to improve students' mental models. Assignments can be designed to engage students in purposeful learning and encourage them to complete programming assignments (Feldman and Zelenski, 1996; Veerasamy, D'Souza, and Laakso, 2016), whereas assignments designed above CS1 students' cognitive abilities can contribute to their failure rate (Oliver et al., 2004). This study contributes towards identifying assignment presentations that may aide in improving CS1 students' understanding of programming problems.

Chapter 5 presented the first program comprehension study, researching design treatments from peer-reviewed publications that previously helped students better understand problem descriptions. Another outcome from the study presented in Chapter 5 was the classification of design treatments that formed an assignment presentation framework. This study demonstrated how the framework could be applied to a problem to form a scaffolded learning environment for the assignment presentation. The scaffold assignment presentation is being evaluated further as the instructional instrument in this chapter, to determine the design treatment's influence in a blended learning environment.

This chapter presents the second study into program comprehension, evaluating a highly scaffolded programming assignment developed for an existing CS1 problem, Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983). The study presented in this chapter compares the classroom application of the assignment to other *Rainfall Problem* studies, measuring understanding through completed programming tasks. This study then invites students to participate in a survey to provide feedback on the assignment presentation, getting their perspective on the assignment design. This study looks at addressing the following research questions related to program comprehension:

- ***RQ1.1***: *How does scaffolding the assignment presentation influence the student's ability to identify goals and subgoals necessary to complete a procedural programming assignment?*
- ***RQ1.2***: *What presentation treatments within the programming assignments support students in their understanding of the problem?*

Results from this study will show the hints and bullet points design treatments helped students understand the programming assignment. The chapter concludes with hints and bullet points as design treatment recommendations and opportunities for future research.

## 6.2 Methods

The study methods are presented in this section. Section 6.2.1 presents the participants involved in the study. Section 6.2.2 describes the design study of the comparative study. Section 6.2.3 presents the survey developed, giving students the opportunity to provide feedback on the assignment presentation.

### 6.2.1 Participants

This study was conducted in a 12-week Introductory Programming (CS1) course offered during the August 2017 semester at the University of Adelaide. The study involved 134 students from the course. Students were given the instructional instrument during week 5 of the semester and given nine days to complete the assignment. During the nine days, students had the opportunity to work on the problem during the lab sessions, where they could receive support from the tutors. Additional background for the study context was previously presented in Section 4.3, providing information on the course programming language, learning environment, and students' programming experience. The information shared in Section 4.3 provides commonalities across studies incorporating student participation.

### 6.2.2 Comparative Study Design

This chapter describes a comparative study that builds on the prior work presented in Chapter 5. The comparative study presents Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983), using design treatments from the assignment presentation framework (See Chapter 5) and evaluating students' completion of tasks to determine if the presentation helped students better understand the problem to complete the task. A comparative study can help identify correlations between previous research results that were performed in diverse settings, providing the opportunity to observe and quantify the relationships between the research variables (Esser and Vliegenthart, 2017). The comparative studies also help in providing additional explanations on the previous findings, potentially strengthening the prior results.

The comparative study examines the students completed programming tasks through the lens of assignment design, and continuing the research on the *Rainfall Problem* from this view. The comparative study uses data collected from prior research using the *Rainfall Problem* (de Raadt, Toleman, and Watson, 2004; Ebrahimi, 1994; Fisler, 2014; Lovellette et al., 2017; Lakanen, Lappalainen, and Isomöttönen, 2015; Lovellette et al., 2017; Seppälä et al., 2015; Simon, 2013). These prior *Rainfall Problem* studies were selected based on the presentation of the results, which identified completed programming tasks in the results and presented an exemplar

problem description of the *Rainfall Problem*. Completion of tasks can indicate that the student identified the task in the problem description. Having access to task completion rates and different assignment designs is an opportunity to compare and contrast design treatments.

Included in the comparative study is the instructional instrument presented in Chapter 5. The instructional instrument generated from Chapter 5, shown in Figure 6.1, presents the *Rainfall Problem* in a highly scaffolded environment that consists of design treatments from peer-reviewed publications, designed to help students better understand the problem. The development of the instructional instrument was described in Section 5.4.

**Averaging Rainfall** (8 marks)
In your assignment, you get to **practise your coding skills** using:

- **Variables**
- **Loops**
- **Arrays**
- **Integers**
- **If Then/Else**

**Program Description:**
**Create a program that averages rainfall for a certain number of days.** Your program first receives the rainfall from an array. The days are processed one day at a time from the array until an array entry is number 99999. 99999 is a signal to stop accepting rainfall information. Then your program should calculate the Arithmetic Mean (Average) of the rainfall. Lastly, it should display the result on the screen in the following format:

*The average rainfall for 'n' days is 'x'.*

The '*n*' is the number of days, and '*x*' is the average of those days.

**Hints:**

- You can use Integers for the rainfall numbers.
- Do not accept negative rainfall, since this does not exist in the real world. If a negative rainfall is provided, convert that day's rainfall to 0. Here is an example rainfall list for 5 days: (15,0,-53,5,2,99999), with rainfall 15 mm, 0 mm, 0 mm, 5 mm, and 2 mm. Please note that -53 is an invalid rainfall value, so your program converts the 3rd day to 0 mm.
- You will also need to calculate the Arithmetic Mean (Average) in your program.

$$Average = (x_1 + x_2 + ... + x_n) \,/\, n$$

- For more hints on the math formula, visit:
http://mathsisfun.com/data/mean-machine.html

FIGURE 6.1 Instructional Instrument Used in the Comparative Study

There are seven separate *Rainfall Problem* studies (de Raadt, Toleman, and Watson, 2004; Ebrahimi, 1994; Fisler, 2014; Lovellette et al., 2017; Lakanen, Lappalainen,

> Write a program that repeatedly reads in a positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average.

FIGURE 6.2 Original *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983)

and Isomöttönen, 2015; Lovellette et al., 2017; Seppälä et al., 2015; Simon, 2013) used as benchmarks for students' problem-solving abilities in this comparative study. These studies used different presentation approaches, focusing on programming language design (Ebrahimi, 1994), context (Lovellette et al., 2017), and programming paradigms (Fisler and Castro, 2017; Seppälä et al., 2015). Other studies (Bonar and Soloway, 1985; Porter, Zingaro, and Lister, 2014) have used the *Rainfall Problem*, but did not report on completed tasks. As a result, these studies are excluded from the comparative study. The completed programming tasks reported in these studies are (Fisler, 2014):

1. `Sentinel` ignores inputs after the sentinel.
2. `Negative` ignores invalid inputs.
3. `Sum` totals the valid inputs.
4. `Count` accumulates the non-negative inputs.
5. `DivZero` guards against division by zero.
6. `Average` averages the valid inputs.

The following are the studies that did report on completed tasks and are examined in this comparative study. These studies were selected because they provided completed programming tasks for the *Rainfall Problem*, and explained the assignment presentation used in the study. The following list provides the previous *Rainfall Problem* research used in this comparative study. The list includes a brief description of the research, its outcomes, and the details of its assignment presentation. Design treatments from these studies are discussed, to demonstrate any additional support provided to the students.

1. Lovellette et al. (2017) attempts to determine whether contextualisation had an impact on students' success rate. This research uses the original *Rainfall Problem* problem, shown in Figure 6.2. The original problem is presented in a non-scaffolded learning environment, presenting the problem given as two sentences without elaboration on the task. The study concludes that contextualisation has no impact on the task completion and overall grades.

> **"Rainfall" problem:** Write a program that will read the amount of rainfall for each day. A negative value of rainfall should be rejected, since this is invalid and inadmissible. The program should print out the number of valid recorded days, the number of rainy days, the rainfall average over the period, and the maximum amount of rain that fell on any one day. Use a sentinel value of 9999 to terminate the program.

FIGURE 6.3 Ebrahimi, 1994 Problem Variant

2. Ebrahimi (1994) study uses the *Rainfall Problem*, shown in Figure 6.3, as a benchmark for students' problem-solving abilities. The presentation contains additional hints about the `Negative` tasks: 'A negative value of rainfall should be rejected...'. The study conducts four separate experiment using Pascal, C, FORTRAN, and LISP programming languages. The study shows that the assignment presentation's additional context has no significant impact on students' results.

3. Seppälä et al. (2015) aims to identify factors during the problem-solving process that might challenge students when solving programming problems. In this study, the *Rainfall Problem* is presented with examples for the `Sum` task, such as 'a run with three positive inputs, a zero, and a negative number before the sentinel' (Seppälä et al., 2015, p. 88). The study was conducted across three institutions, resulting in a list of factors, such as time constraints, that contribute towards challenges faced by students when completing programming problems.

/* Implement the 'Average' function, which takes the amounts of rainfall as an array and returns the average of the array. Notice that if the value of an element is less than or equal to 0 ('lowerLimit'), it is discarded, and if it is greater than or equal to 999 ('sentinel'), stop iterating (the sentinel value is not counted in the average) and return the average of counted values. */

```
public class Rainfall{
  public static void Main(){
    double[] rainfalls = new double[] { 12,
      0, 42, 14, 999, 12, 55 };
    double avg = Average(rainfalls, 0, 999);
    System.Console.WriteLine(avg);
  }
  public static double Average(
    double[] array, double lowerLimit,
    double sentinel){
    // Write implementation here
  }
}
/* Bonus: Write unit tests. */
```

FIGURE 6.4 Lakanen, Lappalainen, and Isomöttönen, 2015 Exam *Rainfall Problem*

4. Lakanen, Lappalainen, and Isomöttönen (2015) uses a version of the *Rainfall Problem* that provides students with both an example and hints. Hints were presented as parameters in the prototype and comments within the code, shown in Figure 6.4. The study was conducted in an exam environment, concluding students struggled with combining language constructs.

5. Fisler (2014) researches functional programming languages, to determine if the programming language contributes towards students' success. Their version of the *Rainfall Problem* includes a hint for the `Negative` task: 'There may be negative numbers other than -999'. The problem also includes a hint to emphasise that the array may or may not include the sentinel marker: 'lists up to the first -999 (if it shows up)'. The study invited three institutions (two university,

one high school) to participate, which showed that students performed better (81-90% success rate) than other *Rainfall Problem* studies using the procedural programming languages.

6. Simon (2013) presents a version of the *Rainfall Problem*, shown in Figure 6.5, was presented in a time-contained environment, devoting two paragraphs on the `Sentinel` task. This produced different errors than the original problem, concluding that arrays used for data entry had their own challenges over keyboard data entry.

A program has a one-dimensional array of integers called iRainfall, which is used to record the rainfall each day. For example, if iRainfall[0] is 15 and iRainfall[1] is 0, there was 15mm of rain on the first day and no rain on the second day.

Negative rainfall values are data entry errors, and should be ignored.

A rainfall value of 9999 is used to indicate that no more rainfall figures have been registered beyond that element of the array; the last actual rainfall value recorded is in the element immediately before the 9999.

The number of days represented in the array is open-ended; it might be just a few days, or even none; it might be a month; it might be several years. The number of days is determined solely by the location in the array of the 9999 entry.

The array pictured here shows that rainfall was recorded for five days, with falls of 15mm, 0, 0, 5mm, and 2mm.

| i | 0 | 1 | 2 | 3 | 4 | 5 | ... |
|---|---|---|---|---|---|---|---|
| iRainfall[i] | 15 | 0 | -53 | 5 | 2 | 9999 | ... |

Write a function method to find and return the average rainfall over all the days represented in the array. A day with negative rainfall is still counted as a day, but with a rainfall of zero. The following code might help.

```
private int AverageRainfall(int[] iRainfall) {...}
```

FIGURE 6.5 Simon, 2013 Exam *Rainfall Problem*

7. de Raadt, Toleman, and Watson (2004) study constructs a framework using the Goal/Plan approach (Soloway, 1986) to develop assessments. The motivation for the study is to help students become better problem solvers through assessments constructed by this framework. To evaluate the framework's influence, the study uses the original *Rainfall Problem*, shown in Figure 6.2. Results from the study shows that the framework can be used to construct assessments and help students problem solve.

The extracted quantitative features were placed in a spreadsheet for comparison. Thirteen different reports were used, because two studies (Ebrahimi, 1994; Seppälä et al., 2015) performed separate experiments, using different programming languages and institutions. This study treats these experiments as separate studies, raising the number of studies analysed to thirteen, including the analysis performed on the instructional instrument.

### 6.2.3 Student Survey

A survey was used in this study to seek insight into the problem description, using it to gain insight to support the quantitative results. The survey was administered after the assignment due date, and when the students' solutions were graded by the tutors. The process of grading involved a tutor sitting with the student to run their program and discuss any errors or missing components within the solutions. Students were invited to participate in a voluntary online survey and given two weeks to respond. The survey was administered through Google Forms (Google LLC, 2019a), where students were asked to provide their student IDs to validate that they completed the assignment. The invitation to participate in the survey was done through Canvas, the LMS that administered the course materials.

The survey contains four questions, two Likert scale questions and two open-text, and the assignment presentation for reference. The first two questions relate to the participants' understanding of the problem statement. The first question is a Likert scale question, asking the participant to rate their understanding of the problem from 'Completely understand' to 'Completely did not understand'. The second question is open-text, allowing the participants to explain any issues when reading the problem. The third and fourth questions relate to the presentation of the programming problem, where the third question is a Likert scale question, asking the participant if the layout was helpful; and the fourth is open-text for participants to elaborate on the third question. Appendix B.2.3 provides the full survey administered to the participants.

## 6.3 Analysis

This section describes the analysis performed in this study. Section 6.3.1 describes the comparative analysis performed on the quantitative features extracted from the prior studies reporting results using the *Rainfall Problem*. Section 6.3.2 describes the coding analysis performed on the survey results.

### 6.3.1 Comparative Data Analysis

This section presents the quantitative analysis used for the data collected for the comparative study. Two analysis approaches are used. The first is for the study performed on the data collected from the instructional instrument. The second analysis is on the data collected from prior *Rainfall Problem* research. The results are analysed by comparing the completion rate of the six *Rainfall Problem* programming tasks: `Sentinel`, `Negative`, `Sum`, `Count`, `DivZero`, and `Average` tasks. These tasks were previously described in Section 6.2.2.

To analyse the data collected from the instructional instrument, the teacher for the Introductory Programming course established grading consistency on students' programming solutions to ensure accurate grade reporting. One teacher and seven

| Research | Year | Completed *Rainfall Problem* Tasks | | | | | | Scaffolding Treatments | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | Sentinel | Negative | Sum | Count | DivZero | Average | Context | Hints | Examples |
| This Study | 2018 | 70% | 95% | 94% | 83% | 43% | 76% | • | • | • |
| Lovellette et al., 2017* | 2017 | | | | | 5% | | | | |
| Seppälä et al., 2015 Context 1 | 2015 | 91% | 83% | 97% | | 56% | | | • | • |
| Seppälä et al., 2015 Context 2 | 2015 | 93% | 75% | 95% | | 75% | | | • | • |
| Seppälä et al., 2015 Context 3 | 2015 | 98% | 91% | 98% | | 86% | | | • | • |
| Lakanen, Lappalainen, and Isomöttönen, 2015† | 2015 | 86% | 85% | 91% | 50% | 40% | 92% | | • | • |
| Fisler, 2014 | 2014 | 81-90% | 57% | | 85-89% | 55-61% | | | • | |
| Simon, 2013† | 2013 | 22% | 40% | 36% | 21% | 0% | | • | • | • |
| de Raadt, Toleman, and Watson, 2004* | 2004 | | | 45.2% | 42.9% | 9.5% | 88.2% | | | |
| Ebrahimi, 1994 Pascal | 1994 | 71% | | | | | | • | | • |
| Ebrahimi, 1994 C | 1994 | 73% | | | | | | • | | • |
| Ebrahimi, 1994 FORTRAN | 1994 | 77% | | | | | | • | | • |
| Ebrahimi, 1994 Pascal | 1994 | 74% | | | | | | • | | • |

* Used the original Soloway *Rainfall Problem* Soloway, Bonar, and Ehrlich, 1983
† Presented in a time constrained environment

FIGURE 6.6 Correctly Implemented Tasks and Scaffolding Treatments Used in *Rainfall Problem* Studies

graders were involved in marking the students' programming solutions and assigning a grade. The seven graders met with the teacher the day of grading the assignments, where they discussed how to grade based on the assignment's rubric. Table 6.1 shows the rubric for the instructional instrument.

| **Component** | **1 Mark for Each** |
|---|---|
| Averaging Rainfall (8 marks) | Program correctly calculates the average rainfall. |
| | For loop is used to iterate over the array. |
| | The 99999 signal is implemented correctly. |
| | Negative rainfall is handled correctly. |
| | Array is declared and initialised correctly. |
| | Program displays the correct output. |
| | Code styled appropriately. |
| | Code is commented. |

TABLE 6.1 Rainfall Problem Rubric for Instructional Instrument

Students' grades and programming solutions are associated by de-identified numbers (IDs) for student anonymity and saved in Canvas. The teacher moderated the grading requirements by meeting with the tutors prior to grading requirements, to establish how to mark the programming tasks. The grades where recorded in the Canvas Learning Management System, identifying the completed programming tasks within the students' solutions.

Analysis of means is applied to each task to identify the number of solutions included in the task. The assignment average is included in the data for analysis, to determine if there might be a correlation between successfully completed tasks and academic success. Students that did not turn in their assignments are excluded from the study.

For the seven studies, presented in Section 6.2.2, quantitative features on the design treatments reported within the publications are extracted. The extraction process includes identifying results from the studies on the six programming tasks and the overall average of the *Rainfall Problem*. If the publication does not report on a programming task or the overall average, the data point is left empty and not included in the analysis.

### 6.3.2 Survey

This section presents the analysis performed on the survey results. Correspondence analysis is performed on the two multiple choice survey questions. The correspondence analysis includes a crosstabulation table of frequencies aligned with the students' choices. The crosstabulation table is used to generate the relative frequencies for each multiple choice option, to produce the distribution of students' choices using a Google Sheets spreadsheet (Google LLC, 2019b).

Content analysis was performed on the open-text survey questions, where common themes were identified in the responses. The analysis was initially hand coded, to categorise students' responses to form a thematic framework to discuss the results (Gibbs, 2007). The analysis was then performed using NVivo version 12 for formal coding, mapping the data to themes in the coding framework. The analysis of the

student surveys involved identifying segments in the responses representing a particular theme. The response segments were mapped into the coded framework to identify common themes.

## 6.4 Comparative Study Results

This section reports on the results from the comparative study. Figure 6.6 includes the completion rate of the six programming tasks, the overall average, and the design treatments applied to the assignment presentations. The design treatments are classified as context, hints, and examples. The first row in Figure 6.6 shows the results from the instructional instrument, representing 92 (69%) students from the 134 students enrolled in the course. The remaining rows list the other studies in chronological order.

The overall average of the completed assignments are represented in the table, to determine academic success. However, this study cannot draw any conclusions from the overall averages reported, because only three (23.08%) studies reported grades. The remainder of the section presents the results by the six programming tasks, discussing the design treatments as potential factors for the results.

### 6.4.1 Sum and Average Programming Tasks

Analysis of the completed programming tasks show that students did well on the `Sum` task (94%), with similar results generated in prior studies (95-98%) (Lakanen, Lappalainen, and Isomöttönen, 2015; Seppälä et al., 2015). Evaluation of the presentation approaches for the three studies show that hints and examples were applied. Upon examining the `Sum` task in students' programming solutions, students understood the need for defining a global variable for accumulating rainfall, and for using this global variable within the loop to accumulate the rainfall values contained within the array. Other studies with low completion rates for the `Sum` task are either administered in time-constrained environments (42.9%) (Simon, 2013) or use the original *Rainfall Problem* presentation (21%) (de Raadt, Toleman, and Watson, 2004) without scaffolding treatments.

Results from the study show the `Average` task has a 76% completion rate. Students' programs with incorrect solutions to this task showed their implementation did not check for a valid denominator. These solutions showed students assuming a non-zero value for the denominator, which is related to the `DivZero` task. Students might not have considered division by zero because of how Programming.js handles this state. Instead of halting the program with a failure at runtime, a Processing.js program will continue, setting the `Average` variable to undefined. Students are able to inspect the value of the variable to determine the undefined value, but programming solutions show students did not evaluate the variable's value. Since the program did not halt and they did not validate the variable's value, perhaps students perceived the denominator behaviour to be acceptable. The `DivZero` task is discussed further in Section 6.4.5.

### 6.4.2 Sentinel Programming Task

The results from the instructional instrument show the `Sentinel` task (70%) are comparable to the results from Ebrahimi (1994) study (71-77%). When comparing the scaffolding treatments, both studies used additional context, but result in similar, lower completion rates. When students attempted to combine schemas for the

`Sentinel` and `Sum` tasks, they seem to struggle (70%). These findings are supported by Soloway (1986), concluding novices were challenged by the abutment of plan compositions. Figure 6.7 shows a student's solution that does not properly merge schemas. In his solution, the student places the sentinel in a position that does not guard the count variable (`days`) when encountering invalid rainfall.

```
for(int i = 1; i < rainfall.length; i = i + 1)
{
  if(i <= 9999)
  {
   if(rainfall[i] < 0)
   {
     rainfall[i] = 0;
   }
   else if(rainfall[i] > 0)
   {
     rainfall[i] = rainfall[i];
   }
   average = average + rainfall[i];
   days = i + days;
  }
}
```

FIGURE 6.7 Incorrect Solution for the `Sentinel` Task

### 6.4.3 Negative Programming Task

When comparing the completion rates for the `Negative` tasks, students had difficulty combining plan compositions. For the `Negative` task, the students participating in the instructional instrument performed well (95%), possibly due to the task residing within the loop independently from other schemas.

### 6.4.4 Count Programming Task

When examining the results from the `Count` task, the instructional instrument has a similar success rate (83%) to another study (85-89%) (Fisler, 2014) using a functional programming language. Comparing the presentations, both studies included hints. When examining the incorrect solutions for the `Count` task, 65% of the errors are due to students assuming the length of the array provides the number of rainfall days. For example, Figure 6.8 shows a student's solution using the length attribute as the denominator for averaging. These findings are similar to a study by Simon (2013) that notes that students assumed all array entries were valid. The assumption is an example of a student potentially focusing on the explicit cues, properties within the text that satisfy certain properties necessary to solve the problem (Ginat, 2003); in this case, the student associates the array length to the number of days.

### 6.4.5 DivZero Programming Task

The results from the study using the instructional instrument show that majority of the students did not address the `DivZero` task. With 43% of the participants correctly implementing the `DivZero` task for the instructional instrument, the results correspond to prior studies (Fisler, 2014; Lakanen, Lappalainen, and Isomöttönen,

```
text("The average rain fall for "+
  rainfall.length+"days is:"+
  sum/(rainfall.length)+" mm.",0,height/2 +5);
```

FIGURE 6.8 Incorrect Solution for the `Count` Task

2015; Seppälä et al., 2015). The instructional instrument and prior studies (Bonar and Soloway, 1985; de Raadt, Toleman, and Watson, 2004; Ebrahimi, 1994; Lakanen, Lappalainen, and Isomöttönen, 2015; Simon, 2013; Soloway, Bonar, and Ehrlich, 1983) do not explicitly identify the `DivZero` task, and posit the low success rates for the `DivZero` task in this section. The absence of the task description in the assignment might give students the impression that the `DivZero` task is not required. Ebrahimi previously suggest that alternative use cases were not being addressed by the teacher nor the course learning materials (Ebrahimi, 1994), which might also account for the low completion rate.

For the instructional instrument, a potential reason for the low success rate might be how Programming.js handles the division-by-zero state. Details on how Processing.js handles division-by-zero was presented in Section 6.4.1. The fact that Processing.js did not provide students with an error, might explain why they did not address `DivZero` in the survey. Another reason for low completion rates might be the low transfer skills (Britton et al., 2007), where students do not apply their mathematical knowledge to solve this problem. In the results for students using the instructional instrument, shown in Figure 6.1, the average formula is presented as a hint, to remind them of the division-by-zero rule. However, the low success rate (43%) from the instructional instrument supports the notion that students are not transferring skills. Another reason for the low completion rate might that while other tasks are explicitly identified in the assignment presentation, the `DivZero` is not explicitly stated.

Yet another reason for the low completion rate might be students' apprehension about adjusting their solutions after starting or approaching completion of the assignment. Adjustments to the code might introduce additional testing and debugging, a trepidation shown by Simon (2013), where a student decided to provide a lengthy description to justify the absence of the `DivZero` task instead of refactoring.

## 6.5 Survey Results

The survey questions invited students to reflect on their understanding of the *Rainfall Problem*. Overall, 25% (n=33) of 134 students responded to the non-compulsory online survey. There is selection bias in the survey results, since students who did not complete the assignment did not participate in the survey. Students also had their solutions graded by the tutors when the survey was released. This timing gave students the opportunity to discuss with the tutors a correct solution of the problem, and better understand any problems within their solutions. This section divides the discussion of the survey results into two parts. Section 6.5.1 reports on responses relating to the understanding of the problem, and Section 6.5.2 presents the opinions students had on how the assignment presentation helped them solve the problem.

| Question 1 Options | Responses n=33 (25%) |
|---|---|
| Completely understood | 11 (33%) |
| Mostly understood | 18 (55%) |
| Mostly did not understand | 3 (9.1%) |
| Completely did not understand | 1 (3%) |

TABLE 6.2 Response to Survey Question 1

### 6.5.1 Understanding the Problem

The following are two questions for the student survey related to understanding the *Rainfall Problem*. Table 6.2 presents the students' responses to the first question. All students (n=33) responded to Q1 with majority (88%, n=29) selecting they understood the problem.

- **Q1:** Rate your understanding of the *Rainfall Problem* statement.
- **Q2:** Explain any problems you encountered when reading the *Rainfall Problem* statement.

For question 2, 64% of the students (n=21) responded to the open-text questions, to support their response to Q1. Content analysis was applied to the responses, which identified emerging categories about the tasks and other challenges faced by the students. The analysis identified students reporting on issues with the `Sentinel`, issues understanding the problem in general, misunderstanding edge cases, and problems creating the test cases.

From the survey, 30% (n=10) of the responses were related to issues with the `Sentinel`. Most of the problems were regarding how to use the sentinel to stop reading data (21%, n=7). Figure 6.7 shows a solution from a student having problems integrating the sentinel. The example shows that the student did not use the 9999 marker to exit the loop. This student acknowledges having *"...trouble understanding what the question wants"*. Two other (6%, n=2) responses were related to misreading the sentinel due to 'design-by-keyword syndrome' (Ginat, 2003). Design-by-keyword syndrome is when the student carelessly associates textual patterns in the problem description to programming patterns. Figure 6.9 shows a student misinterpreting the sentinel with a 99999.99999 marker. The students misinterpreted two sentences in the problem statement as a single sentence: *"The days are processed one day at a time from the array until an array entry is number 99999. 99999 is a signal to stop accepting rainfall information."* A response from a student commented that he *"... read 9999.9999"*. One student (3%, n=1) stated he misunderstood the `Sentinel` task's intention , while two students acknowledged their misreading of the sentinel (99999). Out of the two students, one rectified the problem by re-reading the assignment, stating *"...that the rainfall trigger to stop was 99999.99999 when I first read it"*. The statement highlights their use of a Self-Reflective learning strategy to better understand the task's intention (Zimmerman and Pons, 1986).

Responses (18%, n=6) expressing difficulties understanding the problem demonstrate that the students did not recognise that the programming problem was related to solving an Arithmetic Mean equation. For example, *"I found that it took me a while to grasp the concept and had to speak to the tutors before I understood how to attack the question, I feel maybe the wording was a little difficult to comprehend"*. Figure 6.1 shows the hint provided in the problem description, guiding students to use the Arithmetic Mean. The comment made by the student might suggest he did not transfer

his mathematical skills to the CS problem space, a behaviour previously observed with students not transferring mathematical skills to other STEM disciplines (Britton et al., 2007). Another student stated *"Unsure if what the question desired covered the scope of our material thus far at this time"*. This statement might suggest the student is comparing the problem context with previous problems, looking for cues in the text to help solve the problem (Ginat, 2003), instead of looking at the problem to identify reuseable schemas, mental structures that comprise conceptual knowledge for solving a problem (Rist, 1995).

Students also reported misunderstanding the edge cases (12%, n=4), and problems creating test cases (9%, n=3). Examining the responses, it seems students made assumptions on the edge and test cases. For example, a student assumed the sentinel marker was always present. Examples include *"I didn't realise I have to cover a case that an array doesn't contain 99999"* and *"I assumed by the description given that 99999 would always be provided"*.

```
for (int j=0; j<days;j++)
{
 if (rainfall[j]==99999.99999) { //when to stop
    text("The average rainfall for " + days +
      " days is "+average/days+" mm",25,40);
 }
 else if (rainfall[j]<0)
 {
   //for negative values of rainfall
   rainfall[j]=0;
   //finding average
   average=average+rainfall[j];
 }
 else
 {
   //finding average
   average=average+rainfall[j];
 }
}
```

FIGURE 6.9 Misreading of the `Sentinel` Marker

Responses for Q1 and Q2 were analysed by word coverage, the percentage of words used to answer a question associated with a given category. The majority of words (54%) used to answer Q2 were related to the challenges with the sentinel. 18% (word coverage 27%, n=6) of the answers for Q2 provided vague feedback about their overall understanding. For example, *"I have trouble understanding what the question what to be to do [sic]"*. Four students (12%, word coverage 23%) had issues understanding edge cases, while three students (9%, word coverage 17%) felt the assignment presentation did not provide enough guidance. For example, *"I found it unclear on whether we were supposed to create our own data to test"*. Two students (6%, word coverage 2.5%) gave opinions unrelated to the survey question, stating *"Not notable"*.

The three students who reported that they 'Mostly did not understand' provided additional information on how they tried to better understand the problem. One approach includes using the example in the assignment presentation. For example, *"The example gave a bit of clarification"*. Another approach was taking the time to evaluate the problem description. For example, *"On first read, very difficult, but*

| Question 2 Options | Responses n=33 (25%) |
|---|---|
| Yes, the format was helpful | 22 (67%) |
| No, the format was not helpful | 5 (15%) |
| I did not notice | 5 (15%) |
| Not quite | 1 (3%) |

TABLE 6.3 Response to Survey Question 2

*when worked through, it was reasonable"*. The last approach provided in the survey was to ask peers for help. For example, *"I mostly discuss with my peers on what the question meant before starting"*. Approaches included asking peers for help, using the examples provided in the assignment presentation, and taking the time to read through the assignment–all employing self-reflective strategies to resolve their misunderstandings (Zimmerman and Pons, 1986).

Survey results reveal no mention of the `DivZero` task, even though their solutions were discussed with the tutors around the time the survey was released to the students. Students who did not implement the task were not aware of the task's existence, while the remaining students who did implement successfully might not have had any complaints. The significance of the absence of `DivZero` from the survey is supported by previous research (Lakanen, Lappalainen, and Isomöttönen, 2015; Simon, 2013), where the task is not viewed by students as a requirement. Because the survey was open for two weeks to receive students' feedback, another possibility is that students might have forgotten their discussion with the tutors on this task, and did not raise in the survey when answering at a later date.

### 6.5.2 Assignment Presentation

The last two survey questions invited students to reflect on the presentation of the instructional materials to help them solve the problem. Students were asked to respond to the following questions:

- **Q3:** Did the layout of the *Rainfall Problem* statement help you in creating your solution?
- **Q4:** Explain how the *Rainfall Problem* statement's layout helped you create your solution.

Table 6.3 shows the students' responses to Question 3, Q3. There were 33 (25%) students responding to this question, with 67% (n=22) reporting the layout was helpful.

Students reported that the hints in the assignment presentation provided additional support in understanding language constructs, for example, *"It helped me understand the statement 'break' in the for loop"*. This indicates that the presentation helped the student reflect on the natural language description and translate his understanding to what he already knew about the `break` statement. Three students (9%, word coverage of 25%) reported that the hints section was helpful, with one student commenting *"The examples and hints were helpful to know exactly what the rules and constraints meant"* and *"The example gave me a bit of clarification"*.

In terms of the assignment format (12%, n=4), two students made comments about the bullet points being useful, for example, *"Bullet points clarified some of the criteria"*. Three students mentioned that the presentation provided design guidance.

For example, *"The layout help me to envision what the program should do...which is a good way to start building the program"*.

Two students (6%) mentioned that the presentation was unrelated to their understanding of the problem, and instead used different strategies. For example, *"I mostly discussed with my peers on what the question meant before starting"*, suggesting that external factors, when not in a constrained test environment, are used by students to better understand the problem. Students who reported that the format was unhelpful (15%, n=5) did not provide additional feedback on Q4.

## 6.6 Summary

This chapter presents the second study on improving students' program comprehension through the textual representation of CS1 procedural programming assignments. This study builds on the first study presented in Chapter 5, by evaluating an assignment presentation constructed from design treatments that help students better understand programming assignments. This study takes the assignment presentation constructed in the first study and compares the completed programming tasks with other *Rainfall Problem* studies using different design treatments. The study invited students to participate in a survey, to provide their feedback on how the presentation may have helped them better understand the problem.

The study identified, through the comparative study and survey, what contributed towards students' completion of programming tasks. The comparative study showed hints helping with the `Negative` task. The survey concluded that the students appreciated the bullet point design treatment, enabling them to identify tasks through the listing of concepts. Within the instructional instrument, the combination of hints within bullet points might have contributed to the high success rate (96%) in the `Negative` task. The study also finds students using 'design-by-keyword' syndrome, demonstrated by students combining two sentences in the assignment presentation and interpreting the `Sentinel` as *'99999.99999'*. These types of misinterpretations could be avoided by educators reviewing the instructional materials, to anticipate these misreadings. If students still engage the 'design-by-keyword' syndrome, this can also provide insight into how students are interpreting the problem descriptions.

There are limitations and contextual variables to this research. The survey reported on just 25% of the enrolled students, potentially not reflecting the entire cohort's opinions on the design treatments. The survey also did not include those students who did not complete the assignment. Inviting students to who did not complete the assignment to participate might provide insight into their struggles. Other limitations are the factors potentially contributing to the previous *Rainfall Problem* studies' results; factors, such as programming languages and time constraints might have influenced task completion. Additional research is needed to test the treatments in a controlled environment, to remove these factors.

There are future research opportunities with this study. Future research opportunities could focus on drawing comparisons between scaffold and non-scaffolded learning environments. Evaluating the learning environments might provide insight into tasks that benefited from design treatments.

This study used the instructional instrument from the previous chapter, by using the assignment design framework developed to support students in their understanding of CS1 programming assignments. This study confirmed the application of the framework helped better support students' understanding. These findings

contribute to the field of Computer Science Education by providing assignment presentation recommendations to educators for future programming assignments.

One assignment was used in this study to evaluate the design treatments and how it influences students' understanding and completion of tasks. More research can help strengthen these findings. Therefore, another study, described in the next chapter, examines design treatments using a different assignment and involving deeper analysis with students participating in a qualitative study.

# Chapter 7

# Assignment Design Interview Study

This is the last study on the *Assignment Presentation* learning activity. The chapter is organised as follows. Section 7.1 provides an overview of the study. Sections 7.2, 7.3, and 7.4 present the study method, analysis, and results. Finally, Section 7.5 presents the summary.

## 7.1 Overview

The evaluation of the *Assignment Presentation* learning activity continues with a final study, designed to collect more in-depth feedback from students on the assignment design. The purpose of the study is to gain insight into how students perceive that the learning activity supports them in their understanding of the problem, and the extent to which the activity engages students in SRL strategies. In this chapter, the study addresses the following research questions that focus on the *Assignment Presentation* learning activity:

- *RQ1.1: How does scaffolding the assignment presentation influence the student's ability to identify goals and subgoals necessary to complete a procedural programming assignment?*
- *RQ1.2: What presentation treatments within the programming assignments support students in their understanding of the problem?*

This chapter addresses these research questions by performing in-depth analysis with students. Students were invited to participate in a qualitative study using a mixed-methods approach of a questionnaire and interviews, to gain their perspective on an assignment design developed with the framework presented in Chapter 5. With support from the *Assignment Presentation*, this study seeks to identify design treatments that help students identify core concepts and support their use of SRL strategies. By better understanding the core concepts behind the problem, students may be able to form a better mental model of the solution.

## 7.2 Methods

This section presents the study methods. This study uses an education research design (Creswell, 2012) that can help produce 'new theories, artifacts, and practices that account for and potentially impact learning and teaching in naturalistic settings' (Barab and Squire, 2004, p. 2). The education research design uses mixed-methods approach of data collection with a questionnaire and a 30-minute interview session

We are looking for 2-3 volunteers to talk about problem-solving strategies you might show students during class or labs. The interview should not take longer than one hour, and you will get treated to a cafe snack and beverage for your time.

Your input would greatly help the research and would be greatly appreciated. If you are interested in getting involved, please contact the Student Researcher, Rita Garcia, at rita.garcia@adelaide.edu.au

The course lecturer is sending out this email on behalf of the researchers. The lecturer has not shared with the researchers contact information for potential participants. The participation is completely voluntary and that participation, non-participation, or withdrawal will not impact your ongoing employment at the university.

FIGURE 7.1 Class Announcement for Student Volunteers

to form the study method. The remainder of this section presents the study methods, including the study's participants, context, and instruments.

### 7.2.1 Participants

The study was conducted in a 12-week Introductory Programming (CS1) course offered at the University of Adelaide, during the February 2019 semester. Students were invited to participate in the study by an announcement posted in the Canvas Learning Management System, shown in Figure 7.1. From the 95 students enrolled in the course, four male students volunteered for the study. The volunteers, for their participation, received a movie poster and a $50 voucher.

The study was conducted in the lab environment with the participants. Because the study examined students' perceptions on how the assignment design help them understand a problem, the study was performed on the same day the assignment was due, for optimal recall performance. Additional study context was previously presented in Section 4.3, providing background on the course programming language and learning environment.

### 7.2.2 Instructional Instrument Design

This section describes the instructional instrument used in this study. This instructional instrument is a practical programming assignment provided in the middle of the semester (week 6) for the Introductory Programming course and is the fifth practical programming assignment for the semester. Using the fifth practical programming assignment for this study gives the participants prior exposure to four previous assignments developed with the assignment presentation framework.

The instructional instrument was developed using the assignment presentation described in Chapter 5. The instructional instrument was constructed in collaboration with the teacher for the Introductory Programming course. The teacher provided the problem's context through presentations used in a prior semester.

The instrument was formed by using design treatments from the framework, with the assignment context provided by the teacher. Design treatments were selected for the instructional instrument based on the students' prior exposure to the programming concept.

For this problem used in this study, students practise the `functions` programming concept, a concept first introduced in week 6 of the semester. New concepts, such as the `functions` and `refactoring` concepts, are highly scaffolded in the assignment design, to support learning and guidance for these concepts. Previously practiced CS concepts, such as `loops` and `variables`, are presented with less scaffolding.

Another consideration in determining the amount of scaffolding provided for the instrument was the time it would be released to the students during the semester. Assignments administered at the start of the semester are highly scaffolded, containing headers, core objectives, context information, code segments, smaller goals, contextualisation, examples, hints, and visualisations. As students progress through the semester, and practise CS concepts using other assignments, the design treatments in the assignment presentation are reduced, fading the scaffolding in the assignment presentations. Reducing scaffolding allows students to take control of their learning, enabling them to solve learning objectives with less guidance (Knobelsdorf, Kreitz, and Böhne, 2014). Reducing the scaffolding means the removal of design treatments surrounding the programming concept or task. For example, Figure 7.2 does not identify `loops` to construct the gradient scale, because they had prior experience with `loops` in earlier programming assignments. Reducing the scaffolding in this case, describes the looping process, stated in Figure 7.2 as *"The gradient starts with 0 (black) and lightens until it reaches **endValue**"*.

Figure 7.2 shows the results of combining the assignment context from the teacher and the design treatments, placing considerations on the instrument being administered in week 6 of the semester. `Refactoring` is a new CS concept the students practise in the instrument; therefore, the design treatments supporting `refactoring` provide students with the existing code to incrementally adjust content (Settle, Vihavainen, and Miller, 2014), and present the `refactoring` requirements in list format to provide detailed and clean requirements (Kussmaul, 2008; Vihavainen, Paksula, and Luukkainen, 2011a). Figure 7.2 shows the detailed requirements in table format, giving the students a visual illustration of the software design. Boldface format (Carbone et al., 2000) is used to ensure students identify that the word *drawGradient* refers to a function name and not its natural-language meaning. The variable *endValue* is also bold face, to reduce students mistaking with the natural language meaning. Design treatments absent from the instructional instrument are the context and hints sections. Without the context section, CS concepts such as `loops` and `variables` are not presented to the students, encouraging them to identify the concepts independently. See Appendix A.2.2 for the full assignment presented with the Codification Pedagogy.

### 7.2.3 Questionnaire

This section describes the questionnaire used in this study. The questionnaire is designed for students to self-assess their prior programming experience and their use of problem-solving strategies. The questionnaire is given to the participants in paper format prior to the start of the first interview session. Students' answers were transcribed into a Google Sheets spreadsheet (Google LLC, 2019b) for analysis.

**Program Description**
Below is code that draws a number of lines to form a grayscale gradient. The gradient starts at 0 (black) and lightens until it reaches **endValue**. You should copy this code into the Processing IDE to see how it works. Try passing in different values to the **drawGradient** function to see how the output changes.

```
void setup(){
  size(400,400);
  background(255);
  //Call our function, draw a gradient from white to black
  drawGradient(255);
}

// drawGradient - Draws a grayscale gradient,
// the gradient starts at 0 (black) and lightens
// until it reaches endValue.
void drawGradient(int endValue){
  for(int i = 0; i <= endValue; i++){
    stroke(i);
    line(10,i,60,i);
  }
}
```

For your assignment, you will need to copy the above code and extend the **drawGradient** function to have the following parameters:

| Parameter | Description Possible Values | Possible Values |
|---|---|---|
| x | X-coordinate of the top left corner of the gradient. | |
| y | Y-coordinate of the top left corner of the gradient. | |
| vertical | Determines if the gradient is drawn vertically or horizontally. | True - if the gradient is drawn vertically<br>False - if the gradient is drawn horizontally |
| endValue | The end value of the gradient . (gradient will start at colour 0 and go to endValue) | 0 to 255 |
| opacity | The opacity of the gradient. | 0 to 255 |
| col | Determines the colour of the gradient. | 0 - grayscale<br>1 - red<br>2 - green<br>3 - blue |
| widHei | The width or height of the gradient. | |

If any of the input parameters are invalid, your **drawGradient** function should display a warning, and not draw the gradient. See above for the valid values. Use a **for loop** to call **drawGradient** several times, to design a pattern. The pattern should be produced by modifying the gradient attributes: end value, width/height, opacity, orientation (vertical/horizontal), or colour. To achieve full marks for this part, please ensure you create a pattern with at least 5 gradients, and alter at least 2 of the gradient attributes in your pattern.

FIGURE 7.2 Assignment A5.1 Problem Description

The questionnaire contains eight questions: five open-text and three Likert scale questions. See Appendix B.2.2 for the full questionnaire. For this study, three questions pertaining to the student's prior programming experience are selected. The three questions are composed of one Likert scale (Q4) and two open-text (Q7 and Q8) questions. The Likert scale question uses a 5-point scale, asking participants to rank their prior programming experience from 'Very Experienced' (5 points) to 'Very Inexperienced' (1 point). The two open-text questions ask the number of years the participants have programmed and which programming languages they used.

### 7.2.4 Narrative Interviewing

This study adopts a narrative interview protocol (Powell, Fisher, and Wright, 2005), to collect data from students, and to get their perspectives on the assignment design. Narrative interviews enable students to share in their own words their experiences using the assignment presentations, and measure their success in solving the assignment through perceived understanding.

The 30-minute interview involves the study's instructional instrument shown in Figure 7.2. The interview is conducted on the assignment due date for optimal recall performance, using a more in-depth data collection with multiple interviews (Thomson, 2011).

In narrative interviewing, the interview begins with a broad question to encourage the student to share their experiences in their own words. For example, *"Can you tell me how you use the assignment description to better understand the problem?"* The interviewer then encourages additional narrative information with open-text questions, such as *"Is there anything else you can say to ...?"* The narrative interview concludes with a question to recall points raised, such as *"What else can you say that helped you with the problem's context in the previous assignment?"* The questions used in the narrative interviews are in Appendix B.1.1, and are used as a guide for conducting the interviews.

The interview sessions were conducted in the lab environment, with other students in the course developing and discussing their assignments with peers and tutors. The participants did not have privacy from the other students when these interviews were conducted. During the interview sessions, audio-visual materials were used to collect data from the interviews for analysis. `SimpleScreenRecorder` (Baert, 2019) was available on all the lab computers and used to collect the audio-visual material. Upon completion of the interview sessions, the recorded material was placed in Dropbox (Dropbox, Inc., 2019) cloud storage for the researcher to get the interview sessions professionally transcribed as Microsoft Word documents for further analysis.

## 7.3 Analysis

This section presents the analysis performed in the study. Section 7.3.1 presents the analysis on the questionnaire responses, while Section 7.3.2 presents the analysis on the narrative interviews.

### 7.3.1 Questionnaire

This section describes the analysis approaches used on the data collected from the questionnaire. The participants answered eight questions in the questionnaire, but

only three of the questions, those related to prior programming experience, were used in this study.

1. How do you estimate your programming experience. *(Likert scale question)*
2. Besides Processing.js, how many additional languages do you know? *(Open-text question)*
3. For how many years have you been programming? *(Open-text question)*

Two different analysis approaches were performed on the three questionnaire questions: analysis of means, and directed content analysis. Analysis of means was performed on the Likert scale question, where the analysis is performed in a spreadsheet. Analysis of means identified the most frequent rating from the participants. Participants were asked to rate their programming experience using a 5-point Likert scale question. The scale ranges from 'Very experienced' (5 point) to 'Very inexperienced' (1 point).

Thematic content analysis (Marshall and Rossman, 1999) was performed on the two open-text questionnaire questions. These questions were related to previously used programming languages and years of experience. Thematic content analysis was used to identify and classify keywords within the answers that relate to the participants' prior programming experience. Identifying their prior programming experience can help determine how much prior experience they have with interpreting programming assignments, which can influence how they process the instructional instrument. Thematic content analysis can be based on previously defined and emerging categories. Emerging categories can arise during the analysis process, which can add more themes to the coding framework.

Students' answers were transcribed in a spreadsheet and imported into NVivo version 12 to identify coded themes. The thematic content analysis started with two nodes to identify the two open-text questionnaire questions. Two initial nodes were created to identify no prior programming experience and programming languages. Any prior experience was an emerging node, labeling the node with the amount of time provided by the participant. Any identified programming languages were also emerging nodes, labeled with the programming language provided by the participant.

The answers to the two open-text questions can be coded with multiple nodes. For example, the answer can relate to their prior experience and programming language, so the answer would be coded with nodes identifying their programming experience and the programming language they have previously used. The coded themes for the two open-text questions for the participants (n=4) were extracted from NVivo as a matrix, to identify frequencies in the content analysis. The matrix table was generated from the nodes within the coding framework.

### 7.3.2 Narrative Interview

Directed content analysis (Hazzan et al., 2006) was used to analyse the data from the interview sessions. Directed content analysis allows for the coding of data within pre-existing frameworks, while allowing for emerging categories to derive from the participants' responses. The initial coding framework used in this analysis was based on the fifteen design treatments within the assignment presentation framework, shown in Table 5.1. Though not all the design treatments were used in this study's instructional instrument, the intention was to establish a coding framework

| ID | Experience | Self Rating |
|---|---|---|
| StudentA | No prior experience | Inexperienced |
| StudentB | No prior experience | Inexperienced |
| StudentC | Matlab experience | Inexperienced |
| StudentD | Python, but forgot language constructs | Inexperienced |

TABLE 7.1 Participants' Programming Experience

for future studies evaluating assignment presentations constructed with the framework. Key design treatments used in the instructional instrument are presenting information in list format (Venables, Tan, and Lister, 2009), decomposing the programming goals into subgoals (Venables, Tan, and Lister, 2009), using boldface text to identify important concepts (Carbone et al., 2000), using visualisations (Ahoniemi and Lahtinen, 2007), and presenting code fragements as reference (Lorenzen et al., 2012). The analysis process is also looking for these key design treatments in students' responses, to determine if they did make an impact in the students' understanding of the programming problem.

The directed content analysis was refined to include two nodes to identify positive or negative responses based on the participants' perceptions of the design treatment. Another node was created to identify volunteers' responses on the overall problem statement. Any emerging data that did not adhere to the existing coding framework resulted in a newly created node.

The data from the transcribed interview sessions for each participant were imported into NVivo version 12 for formal coding, mapping the data in the interviews to the coding framework. When the coding was completed, the coding frequencies were generated to interpret the findings from the interviews. The coding frequencies were evaluated using a matrix table generated from the established and emerging codes from volunteers' responses. Frequencies from the content analysis were extracted using this matrix table in NVivo version 12, to further discuss the findings. Alternative forms reliability (Creswell, 2012) compared the two interview sessions performed in the qualitative study. These two sessions used alternate forms of the instructional instrument, but measured the same variables in the coding framework, which provides a different method of measuring internal consistency.

## 7.4 Results

This section presents the results from the questionnaire, and the results from the narrative interview.

### 7.4.1 Questionnaire

This section presents the results from the questionnaire, shown in Table 7.1. Of the four participants, two stated they had no prior experience. StudentC stated he had Matlab experience when enrolled in a non-CS science course. StudentD stated he learned basic Python, but did not provide details on being self-taught or learned in another CS course. StudentD did state he forgot all the programming concepts using Python. Table 7.1 also presents the participants' self-assessment on their programming experience. All participants ranked themselves as 'Inexperienced', the lowest level of programming experience in the Likert scale. These results show that majority of the participants (75%) do not have prior programming experience.

### 7.4.2 Interview Results

This section presents the results from the narrative interviews. The following are sequential excerpts from the narrative interview conducted with StudentA, to demonstrate how the interviews were conducted and the coding was performed. The interviewer begins with asking the student to draw on prior assignments to compare how this presentation helped with their understanding. Because the participants are unfamiliar with writing code, they are drawing on past experiences with the assignments previously administered in the course. StudentA stated, *"I do definitely prefer this one because this one very clearly states what's needed. The last one that I had a bit of difficulty with, it was just a paragraph of requirements. This one is very clearly stated what's needed."* StudentA's statement shows his overall appreciation of the presentation, providing positive feedback (Coded as *T16 General statements* and *T17 Positive opinion*). StudentA also compared the instructional instrument to a previous assignment, which presented the problem in paragraph format. He claimed the paragraph format was unhelpful (Coded as *T19 Paragraph format* and *T18 Negative opinion*).

The interviewer then draws out StudentA's opinion about the paragraph format by asking why he believes the assignment was easier to understand. StudentA responds, *"This one was much nicer that I could basically check it off a list."* This statement shows the student using the list format to support his use of Self-Regulated Learning (SRL) strategies and self-evaluation (Zimmerman, 1989) to validate his programming exercise. This narrative segment is coded as *T4 List Form*, *T20 Problem-solving strategies*, and *T17 Positive opinion* from the framework.

The interview concludes with the interviewer asking StudentA if there is anything else he would like to say on the assignment design. StudentA concludes, *"Nothing wrong with the assignment. It was really good for how I look at the problems."* StudentA's final statement reaffirms his positive experience with the presentation. This statement is coded as *T16 General statements* and *T17 Positive opinion* from the framework.

All interviews are coded using the same coding approach as the example segment with StudentA. Table 7.2 shows the results from the coding framework, along with example coded segments. Table 7.2 lists the total number of narrative segments coded to themes containing students' responses. In the reliability analysis, Cronbach's alpha was found to be 0.73, an acceptable internal consistency. As stated in Section 7.3.2, all the design treatments were included in the coding framework. Because some of the design treatments were not used in the instructional instrument, these nodes were not included in the results. The table also shows the percentage the theme appeared in the overall results. The examples show whether it is coded as a positive or negative statement. The final coding framework contains 20 themes, including two themes (*T17 Positive opinion* and *T8 Negative opinion*), representing the narrative segment as a positive or negative influence in the students' understanding of the problem. Prior to coding participants' responses, there were 18 nodes in the framework (15 design treatment, 2 positive/negative opinion, and 1 general statement). While coding the participants' responses, two themes emerged: the presentation of the information in paragraph format (*T19*), and using the assignment presentation to support problem-solving skills (*T20*).

In the results, four of the themes relate to guidelines in the presentation framework: *T1 Bold face text format* (15.79%) (Carbone et al., 2000), *T2 Code fragments* (10.53%) (Lorenzen et al., 2012), *T3 Goal decomposition* (15.79%) (Venables, Tan, and Lister, 2009), and *T4 List format* (31.58%) (Venables, Tan, and Lister, 2009). These results align with the instructional instrument, where these design treatments were

| ID | Results | Theme | Examples |
|---|---|---|---|
| T17 | 16 (84.21%) | Positive opinion | *"This actually makes you sit down and methodically think about how you going to approach it."* StudentB (T16) |
| T4 | 6 (31.58%) | List format (Vihavainen, Paksula, and Luukkainen, 2011a) | *"I think having it set out as dot points really, clearly gives you all the points that you just need to address. It's just laying it out for you."* StudentB (positive)<br>*"When you've got dot points it's more concise, so you've got less fluff around it, so it's easier to get straight to the point."* StudentD (positive) |
| T20 | 4 (21.05%) | Problem-solving skills | *"I generally will reference back to it as I'm doing the assignment just to make sure that I'm ticking all the boxes that they want to see ticked."* StudentB (positive) |
| T1 | 3 (15.79%) | Boldface text format (Carbone et al., 2000) | *"I don't understand why that's (function name) is in bold. That is odd to me."* StudentA (negative)<br>*"Maybe if it's emboldened, people are paying attention to where the conditions are in a statement."* StudentB (positive) |
| T3 | 3 (15.79%) | Goal decomposition (Vihavainen, Paksula, and Luukkainen, 2011a) | *"was a bit more step by step, this one. So it had it really laid out for what you need to kind of add into the program."* StudentC (positive) |
| T18 | 3 (15.79%) | Negative opinion | *"Actually had confusion with this problem that I got help with earlier, I started asking about it."* StudentC (T16) |
| T2 | 2 (10.53%) | Code fragments (Lorenzen et al., 2012) | *"structure is useful, so you sort of know how to start if off, because it's very intimidating to look at that and create it."* StudentD (positive) |
| T16 | 2 (10.53%) | General statements | *"The last one (assignment) that I had a bit of difficult with, it was just a paragraph of requirements. This one is very clearly stated what's needed."* StudentA (positive) |
| T19 | 2 (10.53%) | Paragraph format | *"The ones that are in paragraph form are harder, and every time I check back I have to read the entire paragraph again."* StudentA (negative) |

TABLE 7.2 Narrative Interview Coded Framework

used in the presentation. However, the participants did not remark on the table format in the instructional instrument, a visualisation designed to help the students identify the parameters in the **drawGradient** method. Though the visualisation was

not explicitly stated, volunteers might have referenced when identifying other treatments, such as *"structure is useful"* (StudentD) and *"laid out for what you need"* (StudentC).

The remaining themes did not relate to the design treatments and emerged when coding students' responses. These themes are *T19 Paragraph format* (10.53%), *T20 Problem-solving skills* (21.05%), and *T16 General statements* (10.53%). The participants' preference for bullet points over paragraphs demonstrates their difficulty in identifying programming tasks with the natural language description. StudentA stated *"The ones that are in paragraph form are harder, and every time I check back I have to read the entire paragraph again"*. This statement describes how StudentA is using the problem description for guidance, and has to process the instructional material again, possibly becoming unfocused from their immediate goal. The bullet points seem to help them better find the information, allowing them to remain focus. For example, StudentB stated *"dot points really, clearly gives you all the points that you just need to address"*. There are some subgoals listed in these dot points. The statement from StudentB demonstrates that the presentation of the subgoals helps this student better identify what they need to do to successfully complete the problem.

Common themes include the presentation of goals in list format (31.58%) and the support of problem-solving skills (21.05%), while bold face formatting (15.79%) and decomposition of programming goals (15.79%) are rated equally by the participants' responses.

Most of the comments are positive (84.21%), such as *"very clearly stated what's needed"* and *"I think that's a great way to structure the problem."* This might be due to how the first question was posed, such as *"Can you tell me how you use the assignment description to better understand the problem?"* Any negative narratives (15.79%) raised in the interviews are presented as comparisons to support of the use of the another guideline. For example, Student A stated, *"The last one (assignment) that I had a bit of difficulty with, it was just a paragraph of requirements. This one is very clearly stated what's needed."*

The results show participants' concerns with identifying the explicit steps in solving the problem, such as StudentC stating, *"This was the one that kind of was a bit more step by step, this one. So it had it really laid out for what you need to kind of add into the program."* Participants' responses show that they use: the list format to progress through the problem-solving process, acknowledging the list format helps them quickly identify the problem's goals. Participants use the assignment presentation to validate their work when solving the problem, using it as a checklist for requirements when solving their programming problems. StudentA states, *"I just make sure that I've got all the requirements checked"*, and StudentB states, *"I think having it set out as dot points really, clearly gives you all the points that you just need to address. It's just laying it out for you."* The results show participants raising concerns about the paragraph formatting (*T19*), stating it is difficult to understand because of the parsing of the requirements from the paragraph. For example, StudentA states, *"The ones that are in paragraph form are harder, and every time I check back I have to read the entire paragraph again."*

## 7.5 Summary

In this chapter, the final study into the pedagogy's *Assignment Presentation* learning activity was presented. The study was designed to gain insight into how students use the assignment presentation to better understand the problem and support their

use of Self-Regulated Learning (SRL) strategies. The narrative interviews showed that the decomposition of programming goals, when presented in list format, helped students when re-reading the problem description. Students were able to quickly identify the subgoals when presented in list format. Students noted that when goals were presented in paragraph form, they had difficulty parsing the information. The results also showed that decomposition of programming goals, presented in list format, also supported students' use of SRL strategies. Students would return to the assignment presentation as a reference to determine the next implementation steps and validate their work upon completion.

There are limitations to this study. Not all the design treatments were integrated into the study's instructional instrument, so participants were unable to comment on all the design treatments. More research is needed to have students evaluate each of the design treatments. The interviews were conducted in a lab environment with other students developing their assignments, and asking the tutors for assistance. Another limitation is the small sample size (n=4) of students participating in the narrative interviews. The smaller sample size allowed for repeated and involved observational data collection that generated in-depth information on how the assignment presentation helped them better understand the problem and support their use of SRL strategies. Another limitation is the results from the study comes from male participants. Future research can evaluate other approaches to encourage more female students to get involved in participating in studies.

There are also other future research opportunities from this study. More work is required to map students' perceptions to their academic success. This study demonstrates how narrative interviewing can get students involved with identifying presentation approaches they feel are helpful or difficult to comprehend. This study method could be applied to assignments, where students could get more involved in the design of future assignment presentations.

This study demonstrates through narrative interviewing how students can get involved in the assignment design process, providing educators students' perspectives on the instructional materials. Bringing students into the assignment design process can help identify presentation aspects that they struggle with, such as paragraph formatting. The results from this chapter contributes to the Computer Science Education community by demonstrating to CS educators how they can help students better understand problems through design treatments. Depending where CS1 students are in their learning, educators can use design treatments that identify the decomposition of programming goals with bullet points.

The studies focusing on program comprehension, explained in this chapter and Chapters 5 and 6, were designed to help students form accurate mental models of the solution through better understanding of the programming assignment. With a better mental model, students can take the next steps in the design process towards solving the problem. The next step in the Codification Pedagogy is to help students reflect on what they know to help their understanding of the problem. The Codification Pedagogy attempts to help students through this next process of self-reflection using a learning activity, described in Chapter 8.

# Chapter 8

# Questioning Activity Study

This chapter focuses on the second learning activity in the Codification Pedagogy, the *Questioning Activity*. The chapter is structured as follows. Section 8.1 presents the overview of the study. Sections 8.2, 8.3, 8.4, and 8.5 present the study method, analysis, and results. Finally, Section 8.6 presents the summary.

## 8.1 Overview

This chapter presents the first intervention in the pedagogy, designed to encourage students to reflect on the programming problem through questioning. Questioning can promote Good Learning Behaviours (GLBs), improving students' ability to comprehend assignments, deconstruct problems, and develop solution plans (Boyer et al., 2010). Questioning can help students take an active role in their learning (Graesser and Person, 1994) and overcome aspects of their fragile knowledge, by encouraging them to reflect, seek out information (Perkins and Martin, 1985), and form opinions on the problem (Bloom, 1956). Questioning can promote the use of critical thinking skills, 'thinking explicitly aimed at well-founded judgement, utilising appropriate evaluative standards in an attempt to determine the true worth, merit, or value of something' (Paul and Elder, 2007). Posed questions can help reduce 'monotonous ineffective tutoring dialogs' (Nielsen et al., 2008), encouraging positive learning behaviours (Weusijana, Reisbeck, and Jr, 2004). Through questioning, students can learn to use Self-Regulated Learning (SRL) strategies (Zimmerman, 1989).

This chapter presents the *Questioning Activity* design and describes the study method used to examine the learning activity. The learning activity is designed to improve students' comprehension and support their use of Self-Regulated Learning (SRL) strategies. Measuring these pedagogical goals is done through Bloom's Taxonomy, an educational objective taxonomy consisting of six cognitive levels to address the student's depth of learning (Bloom, 1956). Details on Bloom's Taxonomy, describing the cognitive levels, are in the background literature, Section 2.3. This study is designed to answer the following research questions:

- ***RQ2.1:*** *How do the 23 CS-focused instructional question types align with Bloom's Taxonomy cognitive levels?*
- ***RQ2.2:*** *Does eliciting the questions in an online CS1 learning environment promote the expected cognitive levels from students answering the questions?*

An outcome from this study is the construction of an Instructional Questioning Framework, a framework that assists in the development of questioning activities. Results from the study will show the *Questioning Activity* encourages students to

reflect on the problem and helps them identify misconceptions in their initial interpretation of the problem. Results also show the activity encourages students to use SRL skills to plan a solution.

## 8.2 Methods

This section presents the study methods used to construct and analyse the *Questioning Activity*. Section 8.2.1 describes the participants for this study. Section 8.2.2 describes the development of a questioning framework used to construct the learning activity. Section 8.2.3 presents the instructional instrument used to examine the learning activity.

### 8.2.1 Participants

The *Questioning Activity* is evaluated in a 12-week Introductory Programming (CS1) course. Additional background for the learning environment was presented in Section 4.3. The information presented in Section 4.3 consists of commonalities across the studies. This study contains three study groups.

1. A control group (C1) is from the August 2017 semester, with 134 enrolled students. Group C1 did not receive the learning activity. This group received the practical programming assignment using the *Rainfall Problem* as the context. Background on the *Rainfall Problem* can be found in Section 4.3.

2. An experiment group (E2.1) from the March 2018 semester, containing 129 students. This group received an isomorphic version of the *Rainfall Problem* and the *Questioning Activity* as a non-compulsory activity, but this group did not submit answers to the *Questioning Activity*. Isomorphic problems presents the learning objectives in the same way so that the learners development the solutions along the same problem-solving paths. Isomorphic problems can reduce the threats to validity, allowing for data comparison across cohorts.

3. An experiment group (E2.2) from March 2018 semester, containing 120 students. This group received an isomorphic version of the *Rainfall Problem* and the *Questioning Activity* as a non-compulsory activity. This group E2.2 did submit their answers to the *Questioning activity*.

### 8.2.2 Framework Design

This section describes the development of an Instructional Questioning Framework that helps with the development of the *Questioning Activity*. Developing the framework involves building on prior questioning research that identified 23 Instructional Question Types (IQTs) used by CS teachers during student-teacher dialogues (Boyer et al., 2010). The Boyer et al. research aimed to help teachers identify questions to help guide students through the problem-solving process. Background on the 23 IQTs was introduced in Section 3.3, listing the IQTs in Table 3.2. The table also lists the IQTs with exemplar questions provided by Boyer et al. (Boyer et al., 2010). The table contains eight emerging questions, represented with asterisks.

This thesis selected these 23 IQTs to construct the framework, because the classification contains a combination of questions that support lower and higher-order critical thinking skills. Having a variety of question types provides the opportunity to construct a range of question activities for students' different cognitive abilities.

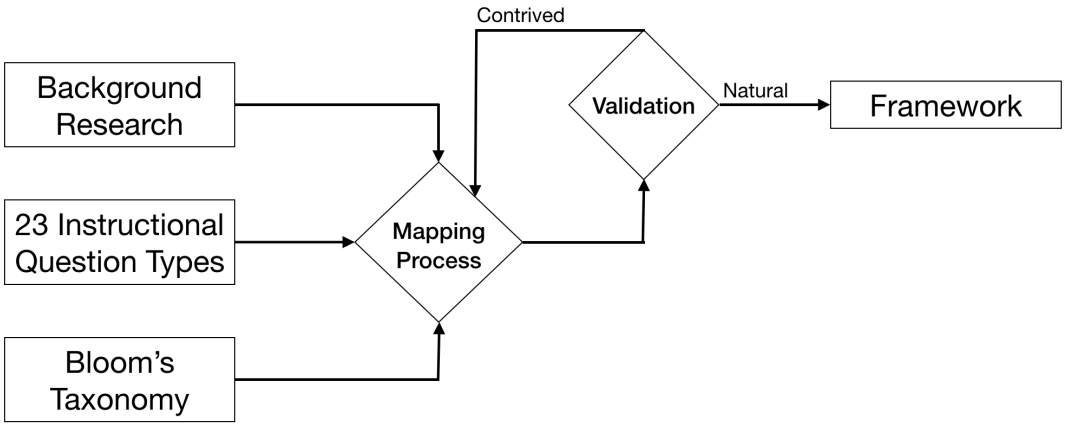



FIGURE 8.1 Questioning Framework Development Process

To identify the critical thinking level for IQTs, the 23 IQTs are mapped to Bloom's Taxonomy's six cognitive levels. Bloom's Taxonomy is an educational objective taxonomy with six lower and higher-order cognitive levels (Bloom, 1956). Background on Bloom's Taxonomy was presented in Section 2.3, providing the definitions for the six cognitive levels in Table 2.1.

To align the question categories to the Bloom's cognitive levels, research was performed to better understand the cognitive requirements for the question types. This included reviewing previous question taxonomies (Boyer et al., 2010; Collins and Stevens, 1983; Nielsen et al., 2008). For eight emerging IQT categories, denoted with asterisks in Table 3.2, additional research was performed to better understand these categories. The additional research included evaluating definitions for the categories and evaluating a model for detecting dialogue acts (Stolcke et al., 2000) for the categories *Backchannel*, *Knowledge*, and *Clarify*.

After collecting the background materials to better understand the question types, the 23 IQTs were classified within Bloom's using the problem context for this study, Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983). The classification process was similar to the task assessments constructed by Thompson et al. (2008), where assessments were first constructed by the researchers individually and later discussed as a group, to identify discrepancies and form a classification for each cognitive level. Analysing the classification process is discussed in Section 8.3.1. Figure 8.1 shows the process of the framework design and analysis. The figure is a visualisation of the mapping process, bringing the background literature, 23 IQTs, and the Bloom's Taxonomy cognitive level to form the framework. Section 8.3.1 describes the analysis process in this figure, validating the placement of the IQTs in the Bloom's cognitive levels.

### 8.2.3 Instrument Development

This section describes the construction of a questioning activity using the Instructional Questioning Framework described in Section 8.2.2. The questioning activity developed in this section is referred to as the 'instructional instrument' for the remainder of this chapter, since it is used in this study as the measurement device.

The instructional instrument is constructed after the development of the Instructional Questioning Framework. The instructional instrument is analysed for proper placement of the IQTs in the Bloom's cognitive levels. To construct the instructional instrument, question types are selected from the framework that engage lower and

- *Q1: What do you already know about collecting and organising data for products sold that might help you complete this assignment? (open-text)*
- *Q2: Breaking down the problem into smaller tasks will help you complete the assignment. How would you go about breaking down this problem into smaller tasks? (open-text)*

FIGURE 8.2 Instructional Instrument Used in the Study

higher-order thinking skills. The lower-order thinking skills help the students to reflect on previously learned language constructs, while higher-order thinking skills can promote the application of programming knowledge to the assignment's problem domain. By practising both thinking skills, students might develop a cross-referencing comprehension strategy, where the program and problem domain layers are utilised for a solution, resulting in a higher level of understanding about the problem (Pennington, 1987). Also, providing different views of the questions gives students the opportunity to articulate solving the problem with their existing knowledge, diversifying their learning process (Ragonis, 2012).

The instructional instrument contains two questions developed from the assignment's problem context, and administered as Assignment A4.1 The assignment is Soloway's *Rainfall Problem* (See Appendix A.2.1). Figure 8.2 shows the instructional instrument that follows the *Assignment Presentation* activity in the Codification Pedagogy. The instructional instrument is designed to use the cognitive levels aligned with the assignment's problem context, giving students a context for answering the activity.

The first question in the instructional instrument is from the *Knowledge* IQT, mapped to Bloom's *Recall* level. This question is designed to help students reflect and apply their internal knowledge to solve the problem. The second question is a *Plan* IQT, is mapped and designed to the *Analysis* level to encourage students to deconstruct the problem's tasks. The questions are designed to complement each other, layering onto what students already know on their problem-solving skills.

Figure 8.2 shows the instructional instrument containing two open-text questions. The open-text questions allow students to respond in their own words, and are the closest to providing students with free-form typing, an effective method for students to provide their thinking processes (Hume et al., 1996). Answers to the activity are collected anonymously in a Google Sheets spreadsheet (Google LLC, 2019b). Qualitative analysis involved coding segments of the answers into the appropriate Bloom's cognitive levels and knowledge dimensions. Section 8.3.2 provides the analysis of the answers provided by the students for the instructional instrument.

## 8.3 Analysis

This section describes the analysis performed on the Instructional Questioning Framework, described in Section 8.3.1; and analysis of students' answers to the instructional instrument, described in 8.3.2.

### 8.3.1 Framework

Analysing the Instructional Questioning Framework involved validating the placement of the 23 IQTs in the Bloom's cognitive level. Analysis involved applying the IQT within the context of a CS1 learning environment and using the cognitive level

where the IQT was placed during the construction of the framework. Figure 8.1 provides a visualisation of the framework development process, including validating the placement of the IQTs in the Bloom's cognitive levels. The analysis process generates three questions using Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983), administered as Assignment A4.1 (See Appendix A.2.1). Difficult question generation with the *Rainfall Problem* context suggests the IQT is in the wrong cognitive level. When difficulties are encountered, the IQT's placement is reassessed in another level. The three questions are generated within the new cognitive level. This process continues until question generation becomes easier, signifying proper placement of the IQT in the framework.

### 8.3.2 Analysis of Students' Answers

This section describes the analysis performed on the data collection from students answering the instructional instrument. A simple word frequency analysis is applied to the answers, to identify the most frequent words used by students that might highlight their concerns. The frequency analysis was performed by importing the answers into NVivo version 12, and using the word coverage feature across all the nodes.

Directed content analysis (Hazzan et al., 2006) was conducted using a developed coding framework based on the Bloom's Taxonomy cognitive and knowledge dimensions, to identify and classify themes in the students' answers. The questioning activity contained two questions for the students to answer, so the directed content analysis starts with creating two nodes to identify the first and second question in the activity. Six additional nodes are created to identify the six Bloom's cognitive levels and the four nodes for the revised Bloom's knowledge dimensions (Anderson and Krathwohl, 2001): factual, conceptual, procedural, and metacognitive knowledge. Knowledge dimensions were added to the revised Bloom's Taxonomy (Anderson and Krathwohl, 2001), where the original taxonomy was presented in one dimension, and the additional knowledge dimension's taxonomy was presented in two dimensions. The cognitive level in both the original and revised version identifies the type of process used by the student to learn and the added knowledge dimension represents the 'kind of knowledge to be learned'. Three additional nodes are added to denote behavioural attributes: answers related to not understanding the question, answers in list format, and answers copied from the assignment description.

Each answer for the instructional instrument can be coded with multiple nodes. For example, the answer is either placed in the first or second question node followed by the answer added to nodes related to the knowledge and cognitive dimensions and behavioural attributes.

A pilot project was created to evaluate the coding criteria, consisting of a subset of student answers. Table 8.1 shows examples from the pilot project. After the answers from the pilot project were coded, and a negotiation process transpired between the researcher and her advisors, to strengthen the coding reliability. The negotiation process was influenced by the Thompson et al. interpretation of Bloom's knowledge dimensions for CS programming assessments (Thompson et al., 2008). The negotiation resulted in an agreed-upon coding criteria that was applied to the remaining answers. When evaluating the answers during the negotiation process, the answers were discussed to identifying the delineation between the *Application* and *Analysis* cognitive levels. The delineation established answers in the *Analysis*

| Level | Coding Criteria | Example Student Answers |
|---|---|---|
| Recall | • When student provides the name of the schema or language construct.<br>• When the student provides the definition for the language construct. | • Arrays are used<br>• if and else<br>• you can use arrays to store the data |
| Comprehension | • When the student incorporates part of the current problem description into their answer, along with basic description of the schema or language construct. | • An array with n elements (representing n days) can be created, and filed with the number of elements sold for each day.<br>• use if and else to decide the number is positive or negative and distinguish between -789 and other numbers |
| Application | • When the student maps tasks or constructs to a part of the problem-solving process. | • First generate the random numbers of products sold.<br>• Setting something to read out how many things are in the array before -789. |
| Analysis | • When the student connects constructs to form parts of the problem-solving process.<br>• When the student identifies the relationship between tasks to solve the problem. | • Create an array of products sold, program goes through array and calculates how many days (how many elements in array) until reach value -789 (end loop), calculate average of products sold (loop from array (0 to N-1) |
| Synthesis | • When the student describes how all of the tasks or constructs work collectively to solve the problem. | • shopkeeper who can simply enter the products sold in chronological order.<br>• Next, to find the average, all the elements of the array (now with no negatives) |
| Evaluation | • When the student expresses judgement or assessment of the suitability or quality of an approach. | • The former because sales aren't always consistent and the later because of holidays, unforeseen events, etc<br>• make sure that works before testing edge tests |

TABLE 8.1 Excerpt of the Coding Criteria for Students' Answers

cognitive level, representing the relationships between tasks as part of the problem-solving process, while answers that mapped tasks or constructs to parts of the problem-solving process were coded in the *Application* cognitive level. All coding was performed in NVivo version 12.

A matrix table was generated using the Bloom's cognitive levels and knowledge

dimensions. Frequencies from the content analysis (Hazzan et al., 2006) were extracted using this matrix table in NVivo version 12 and imported into IBM SPSS Statistics version 2.5 for statistical analysis (Greasley, 2008). To ensure coding reliability, the researcher's supervisor coded 25% of the students' answers, and negotiated the partial coding with the author. To strengthen the coding validity, due to the software limitation in inter-rater reliability, alternate coding reliability was performed.

| Level | Category | Example |
|---|---|---|
| Uncategorised | Backchannel* | Right? |
| | Focus* | See where the array is declared? |
| | Hints* | We didn't declare it; should we do it now? |
| Recall | Definition | What does that mean? |
| | Knowledge* | Have you ever learned about arrays? |
| Comprehension | Calculation | What is 13 % 10? |
| | Casual Consequence | What if the digit is 10? |
| | Clarification* | What do you mean? |
| | Confirmation* | Does that make sense? |
| | Enablement | How are the digits represented as bar code? |
| | Procedural | How do we get the $i^{th}$ element? |
| | Quantification | How many times will this loop repeat? |
| Application | Free Creation | What shall we call it? |
| | Goal Orientation | Did you intend to declare a variable there? |
| Analysis | Casual Antecedent | Why are we getting that error? |
| | Feature/Concept Completion | What do we want to put in digits[0]? |
| | Free Option | Should the array be in this method or should it be declared up with the other private variables? |
| | Judgement | Would you prefer to use math or strings? |
| | Justification | Why are we getting the error? |
| | Plan | What should we do next? |
| Synthesis | Improvement | Can you see what we could do to fix that? |
| Evaluation | Assessment* | Do you think we're done? |
| | Status* | Do you have any questions? |

TABLE 8.2 Instructional Framework (* Denotes Emerging Question Category (Boyer et al., 2010))

## 8.4 Framework Results

This section contains the results from generating the Instructional Questioning Framework. Table 8.2 presents the Instructional Questioning Framework, with the IQTs presented in the Bloom's cognitive levels. The table includes the IQTs and example questions from the Boyer et al. (2010) study.

Validating the IQTs using question generation resulted in the reassessment for three categories. Both *Feature/Concept Completion* and *Plan* IQTs were initially categorised in the *Application* cognitive level, but the questions generated were focusing on how tasks were related to solve the problem, which is more aligned to the *Analysis* level. Subsequent generated questions for these categories were easier to construct when moved to *Analysis*. The third category, *Focus*, was initially categorised in *Comprehension*, but like *Backchannel* and *Hints*, mapping became difficult due to the context, with generated *Focus* questions being applicable to different cognitive levels. Therefore, for the preliminary framework, the three IQTs (*Backchannel*, *Focus*, and *Hints*) remain uncategorised.

There was success in mapping most of the IQTs, except for *Backchannel*, *Focus*, and *Hints*. These IQTs were found to span multiple cognitive levels based on the question's context. Results from Boyer et al. (2010) showed *Hints* as the most frequently posed question type, perhaps due to student-teacher discourse generating hints within multiple contexts and cognitive levels. The inability to classify these categories might be due to the way the questions are applied in the two studies. Boyer et al. (2010) used questioning in a student-teacher discourse, showing that this type of questioning can encourage deeper learning by the student. This study utilises instructional questioning as a learning activity within a programming assignment.

The *Comprehension* cognitive level was observed to contain the most instructional question types. These findings support educators' frequent use of comprehension-based questions in classroom environments (Graesser and Person, 1994). Perhaps in the Boyer et al. (2010) study, educators used familiar question-asking strategies they employed in the classroom. Results also demonstrate students predominately using *Procedural Knowledge* that referenced language constructs to answer the questions. The predominate use of *Procedural Knowledge* correlating to Bloom's *Analysis* level might be a unique behaviour for CS over other disciplines, since students might draw on algorithms and programming techniques when analysing problems.

Answering the question might require different cognitive levels, depending on the students' domain knowledge and abilities. A previous study (Buckley and Exton, 2003) observed students using lower than the intended cognitive levels to complete CS assessments, showing that students' prior experience and domain knowledge influenced their cognitive level. When using the framework to construct the *Questioning Activity*, educators might observe students answering with different or unexpected levels. If answers fall into unexpected cognitive levels, this might indicate that students are struggling with concepts, might not understand the questions, or are neglectful in forming satisfactory answers; or tht the question does not elicit the desired cognitive or knowledge levels, which may require question revision by the educator.

## 8.5 Questioning Activity Results

This section presents the results from students answering the instructional instrument. The results were evaluated from the instructional instrument using directed context analysis to determine the cognitive levels students used when answering. The overall average of the programming assignment was also examined to determine if the *Questioning Activity* might have impacted the students' programmed solution.

The average grades were evaluated for the three groups' programming assignments. Group C1 with no exposure to the *Questioning Activity* had an overall average

of 83%. Group E2.1 who received but did not submit answers to the activity had an overall average of 86.6%. Group E2.2 answered the activity with an overall average of 86.9%. Comparing the grades for the groups showed no significant differences when students answered the activity. Results from t-tests were also analysed, the control group $t(211)_{C1}$ = -1.194, p=0.861, and the experiment groups E2.1 and E2.2, $t(247)_{E2.1}$ = -0.140, p=0.889 and $t(242.241)_{E2.2}$ = -0.140, p=0.889, with the results showing no significant differences.

Keyword frequency analysis of student responses in NVivo were analysed. The results showed *array* (f=202) was the most frequently used, followed by *number* (f=162), and *products* (f=157). Whereas *array* focused on language constructs, the second and third commonly used words relate to the problem's context. The language construct *loop* was observed as the next most commonly used term, ranking 12$^{th}$ (f=80) in the frequency list.

The remainder of this section analyses 141 unique responses to the *Questioning Activity* from group E2.2. Table 8.3 represents the coded answers aligned with Bloom's cognitive levels, while Table 8.4 shows the coded answers against the knowledge dimensions. Both tables provide the percentages of the coded answers, representing the frequency of lexical items used to answer each question.

A bivariate correlation was performed across both Q1 and Q2 in the instructional instrument for more substantial data analysis. Additional reporting is done on high correlations between the Bloom's cognitive levels and knowledge dimensions, shown in Tables 8.3 and 8.4. The analysis shows a high correlation between *Recall* and the *Factual* (r=0.515) and *Conceptual* (r=0.399) knowledge dimensions, while *Analysis* has a high correlation with the *Conceptual* (r=0.430) and *Procedural* (r=0.418) knowledge dimensions.

### 8.5.1 Analysis of Activity Question 1

This section discusses the answers from the first question in the instructional instrument: *Q1: What do you already know about collecting and organising data for products sold that might help you complete this assignment?* The first question, Q1, is designed to target the Bloom's *Recall* cognitive level, to engage the students' internal knowledge-base. Of the 120 participates in group E2.2, 12.8% (n=18) did not answer Q1. Table 8.3 shows answers predominately residing in the *Recall* 13.31% (n=54) and *Comprehension* 14.43% (n=58) cognitive levels. Answers in *Recall* include the definitions of language constructs, such as 'For loops can be used to check elements', 'arrays are useful for storing and organising data', and 'Data is best stored and organised in an array'. The *Comprehension* level show students answering in more detail, such as 'An array with n elements (representing n days) can be created, and filled with the number of elements sold for each day' and 'Number of products sold and the number of days that they are sold on can vary'.

| **Bloom's Taxonomy** | **Q1** | **Q2** |
|---|---|---|
| Recall | 54 (13.31%) | 8 (1.96%) |
| Comprehension | 58 (14.43%) | 25 (5.07%) |
| Application | 32 (6.5%) | 38 (10.91%) |
| Analysis | 7 (1.51%) | 77 (23.07%) |
| Synthesis | 3 (0.45%) | 4 (0.55%) |
| Evaluation | 3 (0.63%) | 4 (0.21%) |

TABLE 8.3 Group E2.2 Results for Bloom's Cognitive Levels

Evaluating Q1 against the Bloom's knowledge dimensions, shown in Table 8.4, shows most answers residing in *Principles and Generalisations* 8.51% (n=42) and *Determine Procedures* 6.93% (n=39). Answers for *Principles and Generalisations* were about the use of the array to 'create an array to store data', and data manipulation, where 'collecting and organising data will help to sort the data out.' Some answers go into greater detail on selecting the procedures and criteria for using the array. For example, 'Your program processes each day from the array until your program encounters the value -789 in the array', 'more efficient to use arrays when several values of the same variable are used', and 'the program reads data in array. Each element which is collected in this array represents the number of products sold for one day.'

| **Knowledge Dimensions** | **Q1** | **Q2** |
|---|---|---|
| **Factual Knowledge** | | |
| Terminology | 25 (4.69%) | 10 (1.61%) |
| Details & elements | 30 (5.84%) | 30 (3.83%) |
| **Conceptual Knowledge** | | |
| Classifications & categories | 3 (0.64%) | 0 (0%) |
| Principles & generalisations | 42 (8.51%) | 35 (4.44%) |
| Theories, models & structures | 8 (1.54%) | 13 (3.26%) |
| **Procedural Knowledge** | | |
| Skills & algorithms | 18 (2.61%) | 21 (4.32%) |
| Techniques & methods | 8 (1.65%) | 24 (4.83% |
| Determine procedures | 39 (6.93%) | 41 (6.82%) |
| **Metacognitive Knowledge** | | |
| Strategic knowledge | 9 (1.89%) | 28 (5.11%) |
| Cognitive tasks | 1 (0.35%) | 10 (1.79%) |
| Self-knowledge | 5 (0.93%) | 1 (0.09%) |
| **Other** | | |
| Does not understand | 5 (1.27%) | 3 (1.06%) |
| Unrelated | 1 (0.35%) | 1 (0.35%) |

TABLE 8.4 Group E2.2 Results for Knowledge Dimensions

### 8.5.2 Analysis of Activity Question 2

This section discusses the answers from the second question in the instructional instrument: *Q2: Breaking down the problem into smaller tasks will help you complete the assignment. How would you go about breaking down this problem into smaller tasks?* The second question, Q2, is designed to target the *Analysis* level, to encourage the use of strategic knowledge and to promote the identification of tasks prior to implementation. Of the 120 students that submitted answers, 90% (n=108) responded to question Q2. Within the Bloom's cognitive levels, shown in Table 8.3, answers mainly resided in the *Analysis* 23.07% (n=77) and *Application* 10.91% (n=38) levels. Answers residing in the *Application* layer identified parts of tasks, for example, 'That the program needs to be written so that it's able to read the data that is inputted and then sorted in some way to produce the desired outcome', 'breaking down the multiple conditions in the correct orders allows coding easier', and 'figure out which parts of the code would be repetitive and decide on a way to do them easier.'

*Determine Procedures* 6.82% (n=41) and *Principles and Generalisation* 4.44 % (n=35), shown in Table 8.4, were the predominately used knowledge dimensions. In *Determine Procedures*, students described procedures in detail, for example 'using if loops.

Make it such that any negative number will be set to zero and such that if -789 is found, the products sold will stop', 'make program processes each day from the array until it encounters the value -789 in the array', and 'checking for the -789 value, otherwise the value would be changed to zero and the stop requirement wouldn't be met until the whole array was checked.' The last example demonstrates the student's ability to describe the algorithm by identifying tasks, such as the sentinel marker (-789). Answers within *Principles and Generalisation* contained fewer task details, for example 'finally did the calculation and displaying of data', 'calculating the average value of array elements', and 'determining the number of days and the number of products sold each day.'

## 8.6 Summary

The *Questioning Activity* was developed as a learning activity within the Codification Pedagogy, designed to encourage students to further reflect on the problem to reduce any misconceptions in approaching the problem. To construct the *Questioning Activity*, an Instructional Questioning Framework was developed to identify questions that encourage students to engage different cognitive levels to view and reflect on the problem. The results show students achieving the desired cognitive levels, using both lower and higher-order thinking skills. Students predominately engaged the *Recall* and *Comprehension* levels for lower-order thinking skills, while *Analysis* was prevalent for higher-order thinking skills. These results are aligned with the design of the first question in the instructional instrument, encouraging students to reflect on what they know for solving the problem. The encouragement can be seen influencing the answers, with the *Recall* level correlating to *Factual* and *Conceptual* knowledge dimensions. For example, a student took a critical approach to answering the question by drawing on what they already knew (*Factual*) and bringing the factual knowledge together to solve the problem or parts of the problem (*Conceptual*). The results from the study demonstrate to the Computer Science Education community that layering questions from lower and higher-order thinking skills can help promote the use of critical thinking skills.

The construction of the framework involved the classification of 23 Instructional Question Types (IQTs) into Bloom's Taxonomy, using Soloway's *Rainfall Problem* as the context for the questions, and ensuring the IQTs were classified in the correct cognitive level. The framework can help diversify learning activities, which could help students utilise the appropriate cognitive level in their CS1 learning. Student-initiated questions have been previously classified (Graesser and Person, 1994) to Bloom's, but this is the first attempt mapping instructor-initiated CS question categories. Once the framework was developed, it was applied to a *Questioning Activity* within a blended CS1 learning environment to support students in better understanding how to solve a programming assignment. The study was performed across across multiple cohorts. The results showed that students exposed to the *Questioning Activity*, both answering and not answering, had similar scores on the assignment, potentially showing students that did not answer also benefited from exposure to the activity. Additional analysis is needed to confirm this theory.

There are limitations to this study. Additional measurements are needed to determine the impact of the *Questioning Activity* as a problem-solving intervention, since the results showed no significant gains in the overall class average. Additional analysis of the students' approach to implementing their solution could determine

whether the *Questioning Activity* helped reduce the completion time and students' mistakes.

There are future research opportunities with this study. This study applied the IQTs framework to CS1 courses, but the framework could be examined to construct questioning activities for upper-division CS courses. Though the presented framework is based on Bloom's Taxonomy, the process of question classification could be performed using SOLO Taxonomy, with future research comparing and contrasting the results between the taxonomies.

The *Questioning Activity* is designed to engage critical thinking skills to help students better understand the problem, so that they can think of ways to problem solve. The next part of the Codification Pedagogy takes advantage of students thinking of ways to solve the problem, by helping them looking at organising a plan. The following chapter describes the study on the next learning activity that supports students in organising an implementation plan.

# Chapter 9

# Design Strategy Activity Study

This chapter presents the study on the last learning activity, the *Design Strategy Activity*. The chapter is structured as follows. Section 9.1 presents the overview of the study. Section 9.2 presents a learning tool, Parsons problems, integrated into the *Design Strategy Activity*. Sections 9.3, 9.4, 9.5.1, 9.6, and 9.7 present the study method, analysis, and results. Finally, Section 9.8 presents the summary.

## 9.1 Overview

This chapter presents the study performed on the *Design Strategy Activity*, the last learning activity in the Codification Pedagogy, and builds on the *Questioning Activity*. The *Questioning Activity* has students reflecting to better understand the problem and its goals, while this activity, the *Design Strategy Activity*, is designed to encourage them to think about organising a plan and requirements they identified in the problem description. The *Design Strategy Activity* is designed to encourage the development of the CS1 students' design knowledge. Design knowledge is metaknowledge containing methods for finding answers to problems (Hoadley and Cox, 2009). The *Design Strategy Activity* is designed to help students through a scaffolded learning environment, to help them successfully complete an ordered task list. The learning activity integrates Parsons problems (Parsons and Haden, 2006), an existing learning tool that provides the scaffolded learning environment. The Parsons problems tool has been previously applied (Morrison et al., 2016) to successfully teach students CS programming concepts within the scaffolded learning environment, which makes this tool a viable solution for providing support for CS1 student during the design process.

This chapter presents how the *Design Strategy Activity* was developed, describing the study method to measure the activity's ability to achieve the pedagogical goal of supporting Self-Regulated Learning (SRL). Because this is the first known attempt to use Parsons problems during the design process, the study examines students' strategies in organising plans within the learning tool. This study is designed to answer the following research questions:

- *RQ3.1: How do students use Parsons problems during the design process for solving CS1 procedural programming assignments?*
- *RQ3.2: What Self-Regulated Learning (SRL) strategies are supported by Parsons problems when used as a design-based intervention for programming assignments?*

Results from the study show that the learning activity supports various Self-Regulated Learning (SRL) strategies during the design process. The results also show students using the learning activity outside the activity's original intention,

to help them to validate their work during the software development process and ensure they have completed all the programming tasks.

## 9.2 Parsons Problems

Before further presenting the *Design Strategy Activity*, this section describes Parsons problem [1] (Parsons and Haden, 2006), the underlying tool responsible for the presentation of the *Design Strategy Activity*, providing the learning materials in a highly scaffolded learning environment. The Parsons problems' highly scaffolded learning environment gives students the opportunity to build their design knowledge in a curated learning environment designed to help them succeed in completing the learning task. This section provides background on Parsons problems, describing the tool's contribution to the design of the *Design Strategy Activity*. This section also presents related work using Parsons problems, work that has made compelling arguments for using this tool to help CS1 students through the design process.

Parsons problems is a learning tool originally designed to help students construct a working program by arranging fragments of code within a scaffolded learning environment (Parsons and Haden, 2006). Parsons problems has previously shown (Morrison et al., 2016) higher learning gains when teaching programming concepts. Morrison et al. showed this was accomplished by reducing the students' cognitive load (Morrison et al., 2016), the effort placed on the student's working memory when performing a task (Sweller, Ayres, and Kalyuga, 2011). Parsons problems has shown students using less trial and error problem-solving strategies and instead engaged logical problem-solving skills (Denny, Luxton-Reilly, and Simon, 2008) when developing programming solutions. Subgoal labeling is a feature introduced later to Parsons problems, which provides natural language descriptions to code fragments (Margulieux and Catrambone, 2014). When using subgoal labeling to help students identify the fundamental structure of the working program, results showed it increased their comprehension on 'the meaning and sequence of programs without having to also generate syntax' (Morrison et al., 2016).

Figure 9.1 shows an example Parson problem that contains fragments of code which the student arranges to form a working program. This figure shows the Parsons problem constructed with `js-parsons` (Karavirta et al., 2019), a Javascript library that provides Parsons problems in online learning environments. Features available in `js-parsons` include different types of feedback to students upon request, executing code to evaluate variable values, running unit tests, supporting multiple programming languages, and supporting distractors. Distractors are extraneous fragments that can help educators identify students' misconceptions about a problem, but were found to show 'no difference in task transfer performance' with learners (Harms, Chen, and Kelleher, 2016). The `js-parsons` library contains line-based, variable-check, and unit-test feedback that are initiated by the user. The line-based feedback validates the placement of code fragments, while variable-check and unit-test feedback executes the code to ensure the constructed program works properly.

Figure 9.1 shows an example problem using the `js-parsons` library. The interface provides students with three buttons: *Start Again*, *Check Answer*, and *Submit Final Answer*. The *Start Again* button returns the fragments of code to the start state

[1] A survey of the literature shows the tool referenced as Parsons puzzles, Parsons problems, Parsons, and Parson's. Parson's was the original label given to the tool by the original author, Dale Parsons. This thesis uses the term Parsons problem.

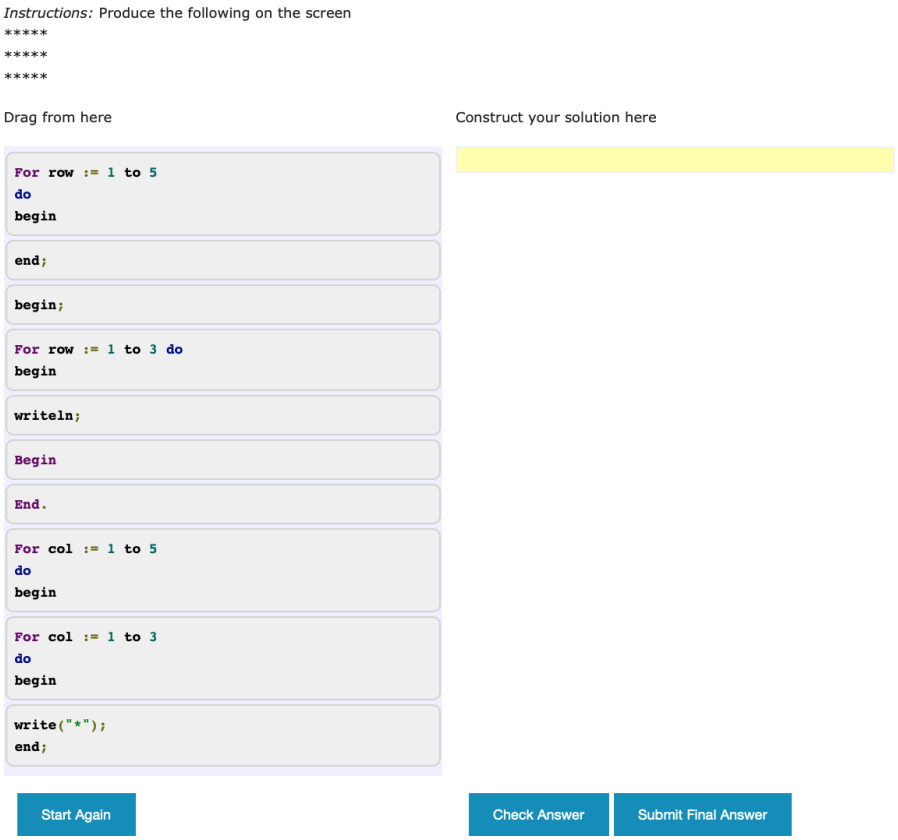



FIGURE 9.1 Example Parsons Problem with Code Fragments (Parsons and Haden, 2006)

in the left-hand column, listing fragments in random order. The *Check Answer* button initiates feedback on students' answers. The responses vary based on the feedback selected for the activity, such as line-based feedback that highlights incorrectly placed fragments in the student's arranged solution. The *Submit Final Answer* button logs the student's interactions.

Parsons problems was selected as the underlying presentation approach for the *Design Strategy Activity*. Parsons problems was used to build students' design knowledge through a scaffolded learning environment, which has previously demonstrated success in supporting the learning of CS concepts. Parsons problems minimises other operations that might distract from the students' learning of CS concepts, such as focusing on syntax details. Because CS1 students might not have developed design-based strategies to solve a problem, they might need assistance on where to begin in the process. The intention of using Parsons problems during the design process is to provide students with a learning environment that shows them where to begin in the design process. This thesis builds on the prior Parsons problems research, to investigate whether students also have learning benefits when the tool is applied in the design space. This study can help determine whether Parsons problems can be used as a design strategy. The environment provided by the Parsons problem may also help students focus on the design process, giving them guidance through the process.

## 9.3 Methods

This study uses an education research design study method (Creswell, 2012) that can help produce 'new theories, artifacts, and practices that account for and potentially impact learning and teaching in naturalistic settings' (Barab and Squire, 2004). The education research design uses two different approaches to collect data for this study. One approach uses a quantitative method for collecting data on how students interact with the *Design Strategy Activity*. The other approach is a usability testing study, comprised of a mixed-methods approach of a questionnaire, think-alouds, and interviews to gather detailed data into students' cognitive processes and experiences when interacting with the activity.

This section describes the study methods used to develop and analyse the *Design Strategy Activity*. Section 9.3.1 describes the participants and context for this study. Section 9.3.2 describes the design of the learning activity, forming the instructional instrument used in the quantitative data collection. Section 9.3.3 presents the questionnaire used in the usability testing, while Section 9.3.4 describes the think-aloud and interview sessions included in the usability testing study.

### 9.3.1 Participants

The *Design Strategy Activity* was evaluated in a 12-week Introductory Programming (CS1) course at the University of Adelaide. Section 4.3 previously described the shared properties across all the studies, presenting information on the course programming language, its blended learning environment, and assignments used in the study. This section describes unique aspects of the participants and context for this study.

The study methods presented in this chapter contain four study groups, shown in Table 9.1. The table shows the study groups used in the quantitative and usability testing study methods. The table provides the name of each group, the semester the group participated in the study, and the size of the study groups. Groups E1 and E2 participated in the quantitative study, where each group was divided into two sections: participants using the learning activity, and non-participants. Because the instructional instrument was non-compulsory, 'non-participants' refer to students in the course that either did not interact with the activity or did not submit their answers for the activity upon completion. A potential reason for non-participants is struggling students viewing the activity as additional work. The data collected for this study did not divide the non-participant group further to account of these struggling students. Table 9.1 shows the distribution of participants and non-participants in groups E1 and E2.

For the quantitative study, both groups E1 and E2 received instructions on how to use the learning activity. Instructions were provided by the teacher, where she explained how to use the learning activity prior to releasing Assignment A4.1. The instructions were provided as a slide presentation, which included an explanation on how to submit their answers in the learning activity.

Groups V1 and V2 were involved in the usability testing study. Students involved in the usability testing volunteered, and met with the researcher individually for more in-depth data collection, observing their use of the activity. Table 9.1 shows the groups enrolled in the August 2018 and February 2019 semesters. Both groups were recruited from an announcement posted in Canvas Learning Management System (LMS). Volunteers for group V1 were selected based on their successful completion of the learning activity for Assignment A4.1. Group V2 is comprised of

any student from semester 1 2019 (February 2019) who responded to the class announcement asking for study volunteers. Having a mix of volunteers previously using the *Design Strategy Activity* and those who have not provides the opportunity to observe students with different usage background. Both groups V1 and V2 were given vouchers and movie posters as incentives for their participation. Because group V1 is a subset of group E2, this group received the learning activity instructions, as a slideshow presentation with the rest of group E2. Volunteers in group V2 were presented with the instructional slides before the first usability testing study session.

### 9.3.2 Intervention Design

This section describes the development of the instructional instrument for evaluating the *Design Strategy Activity*. This study uses the instructional instrument as an intervention for a CS1 procedural programming assignment. The intervention is designed to help students through the design process.

The instructional instrument was developed within the context of a programming problem, Soloway's *Rainfall Problem* (Soloway, Bonar, and Ehrlich, 1983). Background on the *Rainfall Problem* was previously presented in Section 4.3, describing the problem and its six programming tasks. The version of the *Rainfall Problem* used in this study has a different requirement for reading data. The version in this study uses an array to collect rainfall information, instead of using the keyboard for data entry.

The *Design Strategy Activity* is designed using Parsons problems as the presentation approach to providing the learning materials. The Parsons problem tool was presented in Section 9.2, providing an example problem using code fragments to form a working program (See Figure 9.1). For the *Design Strategy Activity*, the fragments of code are replaced with plans to organise, creating an implementation plan for the programmed solution. This is the first attempt to use Parsons problems with implementation plans during the design process. Three types of plans were considered for students to organise. These plans are *strategic*, *tactical*, and *implementation* plans (Soloway et al., 1982). *Strategic* plans define the overall strategy of the problem, *tactical* plans define a local strategy for solving the problem, and *implementation* plans focus on programming language specific approaches for analysing the strategic and tactical plans. For this study, *strategic* plans were selected for use in the *Design Strategy Activity*, to continue to encourage students to think of the overall problem. The *strategic* plans align with the goal of the first intervention, the *Questioning Activity*, which encourages students to reflect on the overall problem. The *Design Strategy Activity* is the second intervention following the *Questioning Activity* in the Codification Pedagogy.

| Group | Course | Total | Participants | Non-Participants |
|---|---|---|---|---|
| **Quantitative Study** | | | | |
| E1 | Feb 2018 | 249 | 139 (55.8%) | 110 (44.2%) |
| E2 | Aug 2018 | 145 | 79 (54.5%) | 66 (45.5%) |
| **Usability Testing Study** | | | | |
| V1 | Aug 2018 | | 6 | |
| V2 | Feb 2019 | | 5 | |

TABLE 9.1 Four Study Groups Involved in the Study

The *strategic* plans were developed using the problem's context, Soloway's *Rainfall Problem*. Because this is the first time students use the learning activity and are new to the design process, the instructional instrument was developed within a highly scaffolded learning environment, providing students a learning environment that would help them succeed in completing the learning tasks. For the highly scaffolded environment, the *strategic* plans are closely related to the *Rainfall Problem*'s context, which are designed to help students relate the *strategic* plans to the programming problem. When developing the *strategic* plans for the *Rainfall Problem*, sentence structures that resemble task variation were avoided, since task variation within instructional materials presents previously mastered skills with new learning objectives (Winterling, Dunlap, and O'Neill, 1987). A lack of task variation can discourage trial and error strategies from students when solving the activity (Denny, Luxton-Reilly, and Simon, 2008), because students might focus on the mastered skills and try to test the placement of the remaining unknown plans until they receive a correct answer. In mathematical word problems, sequence sentences provide guidance in the form of simple phrase-by-phrase translation of the problem, and though sequential sentences might minimise students' frustration and mistakes (Silbert and Stein, 1990), using sentence structure might provide students with ordering clues. Minimising these cues can help students think about the purpose of each *strategic* plan, instead of forming the correct order based on sentence structure.

| **Ordered Plans** | **Associated Programming Tasks** |
|---|---|
| 1. Define variables to create your program. | |
| 2. Collect the rainfall. | `Sum` totals the valid inputs |
| 3. Check is it time to stop collecting the rainfall. | `Sentinel` ignores inputs after the sentinel |
| 4. Check rainfall is valid number. | `Negative` ignores invalid inputs |
| 5. Calculate the average rainfall. | `Average` the valid inputs |
| 6. Display average rainfall. | |

TABLE 9.2 Order of the *Strategic* Plans with the Associated Programming Tasks

Table 9.2 shows the *strategic* plans presented for the instructional instrument, where some of the strategic plans are associated with the programming tasks (Fisler, 2014) for the *Rainfall Problem*. The plans are closely related to the programming tasks, to help students transition from the design process to the implementation process of writing the programmed solution. Table 9.2 shows the instructional instrument with six *strategic* plans, presented in the correct order for implementing a programmed solution. This table shows the *strategic* plans used in the instructional instrument for groups E2 and V1. Figure 9.2 shows the instructional instrument presenting using the `js-parsons` library, the technology used to present the *Design Strategy Activity* within the Canvas Learning Management System. To reduce plagiarism and threats to validity, groups E1 and V2 received an isomorphic version of the *Rainfall Problem*. The isomorphic version contains six *strategic* plans varying contexts, but the ordering of the plans and their association to the programming tasks remained the same. Appendix A.2.1 contains the isomorphic version of the assignment presented to groups E1 and V2.

The method for this study lets participants interact with the learning activity

throughout the development of the programming assignment. The activity provides students with line-based feedback when selecting *Check Answer* button. Line-based feedback was selected because the other feedback approaches provided by `js-parsons` are designed to validate coded solutions, and are not appropriate for the *strategic* plans. The Parsons problems approach used in this study presents students with natural language options that do not work with variable-check and unit-test feedback. When participants are ready to submit their interactions with the activity, they can select the *Submit Final Answer* button. This button sends their interactions to a Google Sheets spreadsheet (Google LLC, 2019b), along with a de-identified student ID for anonymity and the time the student submitted their answer. The de-identified ID allowed for tracking students' interactions with the activity across the assignments. The data collected in the spreadsheet are used for further analysis.

### 9.3.3 Questionnaire

This section presents the questionnaire administered to groups V1 and V2 prior to performing the usability testing study. The questionnaire is designed for the volunteers to self-assess their prior programming experience and problem-solving strategies. The questionnaire was provided to the volunteers in paper format, and their answers were transcribed to a spreadsheet for further analysis.

The questionnaire contains eight questions: five open-text questions and three Likert scale questions. See Appendix B.2.2 for the full questionnaire. The questionnaire does not ask if the volunteer has prior experience to Parsons problems. Three questions use the Likert scale, while the remaining questions are open-text. Using a 5-point Likert scale, the questions ask participants to self assess their programming experience and to compare it with that of classmates and experts. The scale ranges from 'Very Experienced' (5 points) to 'Very Inexperienced' (1 point). Two open-text questions ask students to elaborate on their prior programming experience, asking for previously used programming languages. The remaining three open-text questions ask participants about previously used problem-solving strategies, including strategies they use when encountering frustrating situations during the problem-solving process. Students were asked prior strategies they use to overcome negative emotions during the problem-solving process. These questions help identify whether these known strategies could interfere with the students' use of SRL strategies (Webster and Hadwin, 2014). The questions on prior problem-solving strategies are asked in the context of a mathematical problem, to assess how the participant could relate to the questions based on past problem-solving experiences.

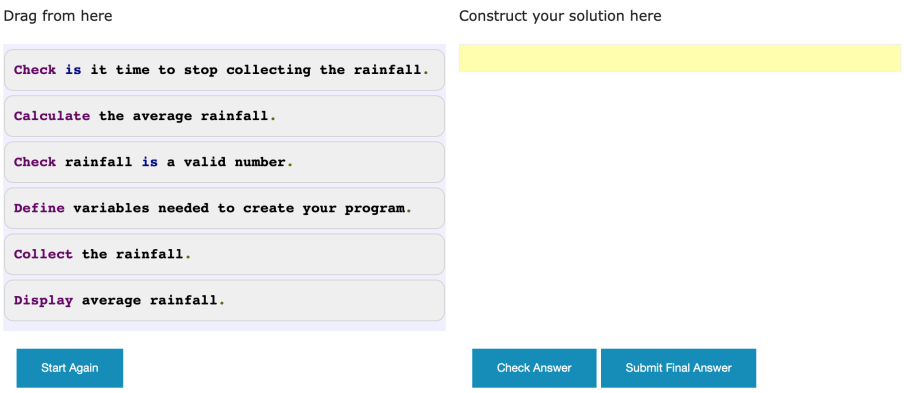



FIGURE 9.2 Instructional Instrument Presented in `js-parsons` Library

### 9.3.4 Usability Testing Methods

This section describes the usability testing methods performed on the *Design Strategy Activity*. The methods involved are think-aloud and interview sessions, which were conducted twice for each study group in the lab environment. Two assignments were used in the usability testing, with sessions were conducted three weeks apart. The sessions for group V1 were conducted on weeks 8 (Assignment A5.1) and 11 (Assignment A6.1) of the semester. See Appendices A.2.2 and A.2.3 for the full assignments provided to group V1. Group V2 sessions were conducted on weeks 5 (Assignment A4.1) and 11 (Assignment A6.1) of the semester. See Appendices A.1.1 and A.1.3 for the full assignments provided to group V2.

Think-alouds were conducted to observe and measure students' cognitive processes and experiences while interacting with the learning activity. The think-aloud sessions provide the most exact method of collecting the student's thoughts while problem solving. The think-aloud sessions use protocols developed by Ericsson and Simon (1993), where students participating in the study were asked to verbalise their problem-solving processes and their pre-existing knowledge while interacting with the learning activity (Ericsson and Simon, 1993).

For this study, prior to the first session with each volunteer, the researcher provided students with instructions on the think-aloud protocol, instructing them to verbalise their thoughts prior to performing actions. The first think-aloud was conducted after administering the questionnaire. During the 30-minute think-alouds, the researcher took observational notes and prompted the volunteers when an extended period of time lapsed with no utterances.

Following the think-alouds, a 30-minute narrative interview was conducted (Powell, Fisher, and Wright, 2005), to get students' perspectives on the learning activity, and to give the researcher the opportunity to follow-up with them on remarks made during the think-alouds. The interviews enabled students to share in their own words their experiences using the learning activity. Nine interview questions were prepared as a guide, related to the three learning activities in the Codification Pedagogy: the *Assignment Presentation*, the *Questioning Activity*, and the *Design Strategy Activity*. Appendix B.1.1 shows the pre-set interview questions, with additional questions asked by the researcher when making observations of participants' utterances and behaviours during the usability testing. The 30-minute interview session was audio recorded using a hand held device while the researcher took notes. The audio recordings were professionally transcribed for further analysis.

The usability testing study was conducted in the lab environment, during the regular tutoring period. The audio-visual materials were collected to help the researcher review and analyse the volunteers' interactions with the learning activity. `SimpleScreenRecorder` (Baert, 2019) was the application used to collect the audio-visual material for the think-aloud session, containing voice and computer screen recordings. This application was available on the lab computers. The audio portion of the audio-visual recordings were professionally transcribed for analysis. Upon completion of each usability testing session, the recorded materials were uploaded to Dropbox (Dropbox, Inc., 2019) Cloud storage as de-identified files. These files were then professionally transcribed into Microsoft Word documents for further analysis.

## 9.4 Quantitative Analysis

This section presents the analysis performed on the students' system interactions with the Parsons problems. Interactive analytics were used to provide different views into the collected data. Interactive data analysis examines the exchange between the human and computer, which 'can lead to actionable observations about the phenomena being investigated' (Turkay et al., 2017). The collected data were students' interactions with the learning activity and analysed to observe how they are using the activity. Students were in the study groups E1 and E2, previously described in Section 9.3.1. The study groups were subdivided into participants and non-participants. The data was collected in a Google Sheets spreadsheet (Google LLC, 2019b) for the participants selecting the *Submit Final Answer* button in the learning activity. The participants' answers were de-identified for anonymity, and mapped to their grades for the programming assignment for analysis. The Assignment A4.1 grades for the non-participants were included for comparison with those students that used the instructional instrument.

Two sets of data were analysed. The first set was the classification of participants' interaction with the activity. System analysis (Ericsson and Simon, 1993) was used to annotate the participants' recorded interactions. The participants' interactions contained their selection of *strategic* plans in an ordered list, their feedback requests, and their requests for restarting the activity.

The participants' interactions were collected within spreadsheets. The spreadsheet were exported as comma-separated values (CSV) files that were processed by a parsing tool with pre-defined pattern rules that identify the type of strategy performed by the student. Pre-defined pattern rules were established by evaluating a subset of the data, to determine patterns in the participants' interactions. These patterns formed a *strategy* that was given a label for easier identification. The patterns describe the organisation and placement of plans until a correct ordered list was achieved by the participant. For each interaction, the parser produced a recommended strategy used by the participant to solve the activity. This recommendation was manually validated by checking the generated patterns with the pattern rules, to ensure the recommended strategy was accurate. Any interactions that could not be identified by the parser were manually examined to determine if the interactions were existing strategies, or if a new strategy needed to be identified. The identified strategies were grouped together, to provide usage frequencies for comparisons.

The second quantitative analysis approach compared groups E1 and E2 grades for Assignment A4.1. The analysis was designed to identify any trends from the strategies used to solve the instructional instrument, comparing the academic success between participants and non-participants. The comparative analysis might also demonstrate whether the pedagogy influenced participants' completion of the assignment's programming tasks. The comparative analysis uses the grades for Assignment A4.1 contained within Canvas Learning Management System. The comparison used analysis of means on the grades for participants and non-participants for groups E1 and E2. The comparative analysis was performed after grading was completed for Assignment A4.1. The grades were exported from Canvas Learning Management System and imported into a Google Sheets spreadsheet (Google LLC, 2019a) to generate the mean used in the comparative analysis.

## 9.5 Qualitative Analysis

This section presents the qualitative analysis performed in the study. Section 9.5.1 presents the analysis performed on the questionnaire. Section 9.5.2 discusses the analysis on the think-alouds. Section 9.5.3 presents the analysis performed on the interviews with the volunteers from groups V1 and V2.

### 9.5.1 Questionnaire Analysis

This section describes the analysis on the data collected from the questionnaire. The participants in the usability testing study answered eight questions in the questionnaire. The participants answered the questionnaire in paper format, which was then transcribed into a spreadsheet. Two different data analysis approaches were performed on the questionnaire responses.

The first analysis approach is analysis of means. Analysis of means was performed on the three Likert scale questions, to identify the participants' most frequent response. The Likert scale responses were represented by a numeric value corresponding to the Likert scale level within a spreadsheet, to generate the mean across all responses to the questions. Internal consistency reliability (Creswell, 2012) was used on the Likert scale questions to measure consistency of the volunteers' answers across both groups.

The second analysis approach was thematic content analysis (Marshall and Rossman, 1999). Thematic content analysis was used to identify words within the answers that form themes in the participants' prior programming experience, and problem-solving skills usage that could be used for further discussion. Thematic content analysis can be based on previously defined and emerging categories. Emerging categories can arise during the analysis process, refining the coding framework. The thematic content analysis was performed on the five open-text questions related to past programming and problem-solving experiences. Identifying the themes of participants' prior programming experience and use of problem-solving strategies was also performed to provide insight into how they might use the learning activity. The participants' responses to the open-text questions were exported from a spreadsheet and imported into NVivo version 12, to code responses. The coded themes were extracted from NVivo as a matrix, to identify common themes. Thematic content analysis was used to create a coding framework, to identify and classify themes in the students' answers.

The thematic content analysis started with five nodes to identify the five open-text questions. Nineteen additional nodes were created, building on previously identified categories related to Self-Regulated Learning and Emotional Regulation strategies. Fourteen of these nodes identify the Self-Regulated Learning (SRL) strategies identified by Zimmerman (1989). The 14 SRL strategies were introduced in Chapter 3, presented in Table 3.1. The remaining five nodes identify Emotional Regulation strategies (Boekaerts, 2011): *Expressing (venting) emotions*, *Suppressing emotions*, *Denial and distraction*, *Re-appraising the situation*, and *Acquiring and Providing Social Support*.

Coding accuracy involved revisiting the coded responses four weeks after the initial coding to include the Emotional Regulation strategies (Boekaerts, 2011). The initial coding was performed using the SRL strategies. The decision to include the Emotional Regulation strategies in the coding framework was due to open-test questions in the questionnaire asking about known strategies students use to reduce frustration. As a result of introducing Emotional Regulation strategies to the coding

framework, all the existing coded responses were revisited, to determine if the coded responses contained in the coded segments included utterances related to Emotional Regulation strategies.

The answers to the five open-text questions can be coded with multiple nodes. For example, the answer is identified with the open-text question ID, followed by the SRL and any Emotional Regulation Strategy stated by the student. When the coding was completed, the coded themes for the five open-text questions were extracted from NVivo as a matrix, to identify frequencies in the content analysis. The matrix table was generated from the nodes within the coding framework.

### 9.5.2 Cognitive Task Analysis

Cognitive task analysis was performed on the data collected in the think-alouds. Cognitive task analysis is a type of task analysis used to understand tasks that require cognitive processing, such as problem solving (Crandall, Klein, and Hoffman, 2006), that generates 'first approximation of the model from information about the task without taking specific psychological factors into account' (van Someren, Barnard, and Sandberg, 1994). Cognitive task analysis incorporates procedural analysis methods (Smith and Ragan, 1999), because the analysis approach involves the temporal ordering of the students' procedural tasks. The procedural analysis defines the mental and physical steps the volunteer goes through to complete the learning activity.

This study first applies the procedural analysis to the video recordings of the volunteers' think-alouds, identifying their procedural steps with the activity. A flowchat is then generated from the identified procedural steps. This study used the presentation approach by Schaafstal (1999) to visualise the procedural analysis, showing the results in a table representation. The table lists all the procedural tasks performed by the students, with their actions mapped to the procedural tasks. This view demonstrates the common actions and problem-solving approaches performed by the students.

After the flowchart is generated for the student, the think-aloud transcript is applied over the flowchart states. Applying the transcript over the students' actions helps to identify the cognitive processes that occur during the procedural step. The result is a protocol fragment explaining the student's actions.

### 9.5.3 Interview Analysis

Directed content analysis (Marshall and Rossman, 1999) was used to analyse the data from the interview sessions. Directed content analysis was selected to analyse the interviews because data can be placed in a context of theory, which allows the data to be applied within an established theory. For the interview sessions, the established constructs are the 14 Self-Regulated Learning (SRL) strategies previously introduced in Chapter 3 and presented in Table 3.1. The 14 SRL (Zimmerman, 1989) and five Emotional Regulation (Boekaerts, 2011) strategies were used as the grounded approach to coding the data, since a goal of the instructional instrument was to support the students' use of SRL strategies, and the questionnaire provided open-test questions strategies to reduce frustration during the problem-solving process.

When the coding was completed, the coding frequencies are generated to interpret the findings from the conversations. Frequencies from the content analysis were extracted using the matrix table from NVivo version 12, to further discuss the findings.

To verify the coding accuracy by the researcher, results from the initial coding were discussed with her advisors. In these results, only the SRL strategies were included in the coding. The results were re-coded four weeks later to include the Emotional Regulation strategies (Boekaerts, 2011). The decision to include the Emotional Regulation strategies in the coding framework was due to open-test questions in the questionnaire asking students to identify known strategies they use to reduce frustration. As a result of introducing Emotional Regulation strategies to the coding framework, all the existing coded responses were revisited, to determine if the coded responses pertaining to the *Design Strategy Activity* contained utterances associated with Emotional Regulation strategies.

## 9.6 Student Interactions Results

This section presents the quantitative results examining the strategies participants in groups E1 and E2 used to solve the instructional instrument, by analysing the participants' system interactions. The strategies are identified as *Top-down*, *Known-First*, and *Experimenting* strategies. These strategies are visually represented in Figures 9.3, 9.4, and 9.5. The figures show the plans represented as circles. The numbers in the circle represent the six *strategic* plans identified in Figure 9.2. The arrangement of the *strategic* plans are depicted from left to right in the figures. The blue circles represent the last plan added to the ordered list. The squares in the figures represent the number of times the participant requests feedback using the *Check Answer* button. The figures show the number of passes used to complete the activity. Six is the smallest number of passes to complete the instructional instrument.

When administering the instructional instrument to group E1, there was a discrepancy with two plans in the activity. The discrepancy affected 70 participants that completed the activity prior to the researcher finding the discrepancy. The discrepancy did not affect the participants' assignment grades. To adjust the results from the 70 participants' interactions with the instructional instrument, their interactions were analysed to the point where they arranged the plans correctly. Identifying where these participants correctly arranged the *strategic* plans removed their extraneous interactions needed to achieve positive feedback from the instructional instrument.

| **Strategy** | **E1** | **E2** |
|---|---|---|
| *Top-Down* | 58 (41.7%) | 38 (48.1%) |
| *Known-First* | 45 (32.4%) | 23 (29.1%) |
| *Experimenting* | 34 (24.5%) | 16 (20.3%) |
| Incomplete | 2 (1.4%) | 2 (2.5%) |
| Total Strategies | 139 | 79 |

TABLE 9.3 Strategies Used by Groups E1 and E2 to Complete the Learning Activity

Table 9.3 provides an overview of the strategies used by the participants solving the activity for groups E1 and E2. The table shows the total number of unique responses from groups E1 and E2. The interactions are divided into the three strategies used to solve the instructional instrument. The most commonly used strategy for both groups is the *Top-Down* strategy, followed by *Known-First*, then *Experimenting*. After presenting these strategies in detail, the results from the comparative analysis are discussed. The results from the comparative analysis bring the different

strategies into the discussion, to determine if any associations can be made between the strategies and academic success. Section 9.6.1 presents the *Top-Down* strategy. Section 9.6.2 presents the *Known-First* strategy, and Section 9.6.3 presents the *Experimenting* strategy.

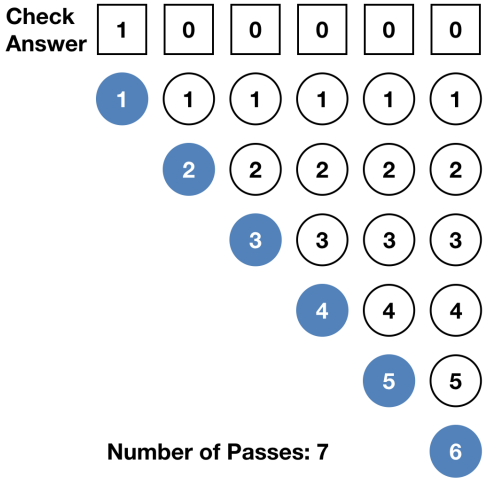



FIGURE 9.3 *Top-Down* Strategy for Solving *Design Strategy Activity*

The rest of this section discusses the participants' problem-solving strategies when using the activity. The three strategies are discussed in detail, along with other behaviours observed when interacting with the activity. Table 9.3 is referenced in the remainder of this section to draw comparisons between the strategies. Table 9.3 shows the *Top-Down* strategy as the most commonly used approach to solving the activity by groups E1 and E2.

### 9.6.1 Top-Down Strategy

This section describes the most used (E1=41.7%, E2=48.1%) strategy to solve the instructional instrument. Figure 9.3 shows an example of a participant using the *Top-Down* strategy. The *Top-Down* strategy has on average the least number of passes for completion (E1=12, E2=10) compared to the *Known-First* and *Experimenting* strategies.

This strategy identifies the participant arranging the *strategic* plans sequentially until all plans are arranged in the correct order in the list. Figure 9.3 depicts the participant arranging the *Rainfall Problem* plans by selecting 'Define variables to create your program' then initiating feedback. The remaining passes show the participant arranging the plans in sequential order: 'Collect the rainfall', 'Check is it time to stop collecting the rainfall', 'Check rainfall is valid number', 'Calculate the average rainfall', and 'Display average rainfall'. During the ordering process, some participants initiate feedback from the activity, but they continue to systematically order the plans in the list. In the example shown in Figure 9.3, the early feedback request might suggest the participant was exploring the activity prior to proceeding further in the learning process.

### 9.6.2 Known-First Strategy

This section describes the *Known-First* strategy, shown in Figure 9.4. This strategy identifies participants selecting plans in non-sequential order to place in the ordered list. *Known-First* is differentiated from the *Top-Down* strategy by the method in which

the plans are selected. The *Top-Down* strategy primarily focuses on selecting and placing plans in sequential order, while participants using the *Known-First* strategy seem to draw 'on past examples to tackle new situations' (Riley, 1981). Figure 9.4 depicts a participant arranging the *strategic* plans for the *Rainfall Problem* in the following manner.

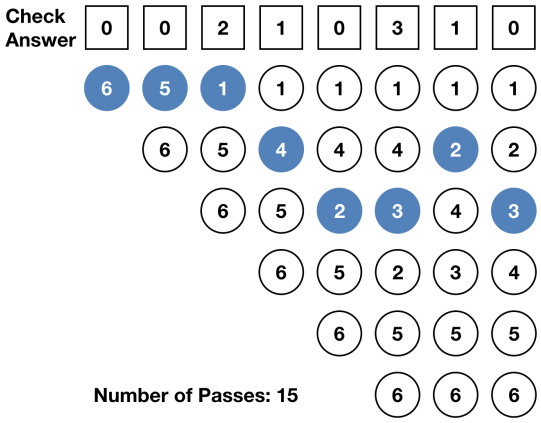



FIGURE 9.4 *Known-First* Strategy for Solving the *Design Strategy Activity*

1. Participant first arranges the 'Display the average rainfall', 'Calculate the average rainfall', and 'Define variables to create your program' plans in the ordered list.
2. Participant selects the *Check Answer* button twice for feedback.
3. Participant adds the 'Check rainfall is valid number' plan to the ordered list.
4. Participant selects the *Check Answer* button once for feedback.
5. Participant adds 'Collect the rainfall' and 'Check it is time to stop collecting the rainfall' plans to the ordered list.
6. Participant selects the *Check Answer* button three times for feedback.
7. Participant arranges the 'Collect the rainfall' plan correctly in the ordered list.
8. Participant selected the *Check Answer* button once for feedback.
9. Participant arranges the 'Check it is time to stop collecting the rainfall' plan correctly in the list ordered.

The example shows the participant struggling with the placement of the 'Check it is time to stop collecting the rainfall' plan. This plan relates to the *Rainfall Problem*'s `Sentinel` task, which ignores inputs once the sentinel is processed. The `Sentinel` task has been shown in prior studies (Ebrahimi, 1994) to be a challenge for novices. The participant selecting 'Check it is time to stop collecting the rainfall' last and then adjusting to place in the right order could indicate that the challenge for novices might begin with understanding and identifying the plan within the problem description.

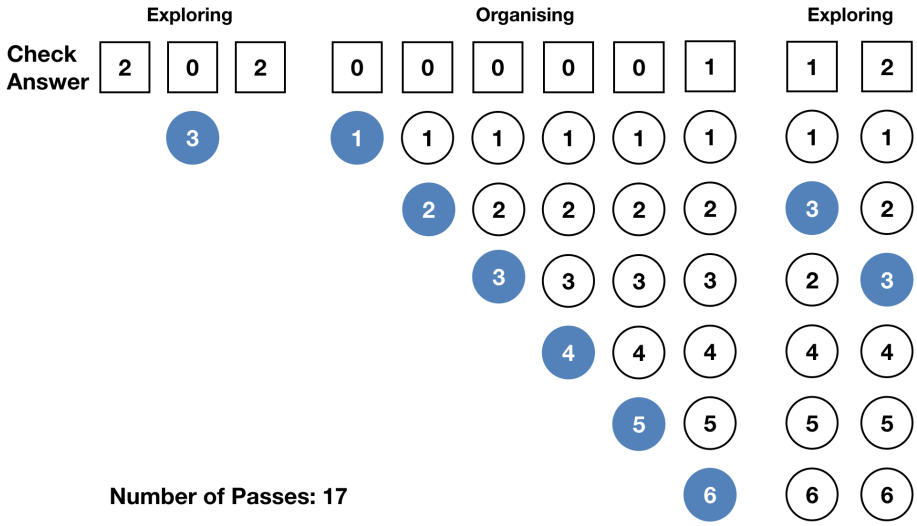



FIGURE 9.5 *Experimenting* Strategy for Solving the *Design Strategy Activity*

### 9.6.3 Experimenting Strategy

This section presents the final strategy identified in this study, the *Experimenting* strategy, shown in Figure 9.5. The *Experimenting* strategy shows participants exploring the instructional instrument by combining the following actions: using the *Restart* button to restart the activity, re-arranging plans within the list, and selecting the *Check Answer* button to receive feedback to validate their answers.

The *Known-First* strategy is different from the *Experimenting* strategy by the haphazard approach participants take when using the *Experimenting* strategy. The *Experimenting* strategy shows participants excessively arranging plans and using the feedback button when interacting with the activity. The average number of passes to complete *Experimenting* is double (E1=24, E2=21) the average approach of the *Top-Down* strategy. The higher number of passes might demonstrate trail and error or *ad hoc* strategies the participants use to complete the activity. This behaviour could suggest that the participant does not fully understand the problem, which might result in the participant having problems identifying plans to arrange in the correct order. Closer examination of the *Experimenting* strategy shows the usage of three different behaviours to solve the activity: *Exploration*, *Trial and error*, and *Incomplete activity*.

1. *Exploration*: Shown in Figure 9.5, this behaviour demonstrates the participant initially exploring the activity by clicking on buttons and arranging plans. Once the participant is satisfied exploring, they proceed to arrange the plans using either the *Top-Down* or *Known-First* approach. When the plans are placed in the list, the user initiates feedback; and upon successful completion, they either stop or continue to interact with the activity. Because this is the participants' first intervention, the *Exploration* behaviour is understandable.

2. *Trial and error*: The participant erratically arranges the plans and presses buttons in the activity until they achieve the correct answer. This behaviour is similar to haphazard tinkering of code, where the participant hopes for a working program without understanding the implications of their code changes (Perkins and Martin, 1985).

| Group Description | E1 | E2 |
|---|---|---|
| Cohort | 86.6% (n=249) | 79.4% (n=145) |
| Participants | 86.9% (n=139) | 81.0% (n=79) |
| Non-Participants | 86.5% (n=110) | 77.8% (n=66) |
| Incomplete Parsons | 0% (n=2) | 78.1% (n=2) |
| Experimenting Strategy | 84.6% (n=34) | 80.1% (n=16) |
| Top-Down Strategy | 87.9% (n=58) | 80.3% (n=38) |
| Known-First Strategy | 84.1% (n=45) | 83.8% (n=23) |

TABLE 9.4 Overall Grades by Groups E1 and E2

3. *Incomplete activity*: The participant begins exploring in the activity, but decides to not complete the activity. An extreme example of this behaviour is the participant pressing feedback buttons without interacting with the plans to form an ordered list.

### 9.6.4 Grade Comparisons

Table 9.4 provides a comparison of groups E1 and E2 overall grades for Assignment 4. Table 9.4 divides the overall grades between participants and non-participants, showing a percentage gain of 3.2% for participants in group E2, but no significant gains for group E1. To validate the results, paired t-tests were performed on the overall grades, showing no statistical differences, $t(119) = 0.1914$, $p = 0.8485$. Both cohorts used the *Top-Down* strategy, followed by *Known-First*, then *Experimenting* to solve the activity. However, no significant academic success was demonstrated when a particular strategy was applied by participants. All three strategies provided similar academic outcomes from the participants.

Overall, participants performed slightly better than the overall cohort average, but not significantly. Comparing the grades are compared based on strategy used by the participants, Table 9.4 shows a percentage gain of 3.3% for E1 students using the *Top-Down* strategy, and a percentage of 3.5% for E2 students using *Known-First* strategy.

## 9.7 Usability Testing Results

This section presents the results from the qualitative study using a questionnaire, think-alouds, and interviews with volunteers from two study groups, V1 and V2. Section 9.7.1 presents the results from the questionnaire. Section 9.7.2 provides the results from the think-alouds, and Section 9.7.3 presents the results from the interviews with groups V1 and V2.

### 9.7.1 Questionnaire Results

This section presents the results from the questionnaire performed in the qualitative study. The study groups V1 and V2 received the same questionnaire, and were asked to self-assess their prior programming and problem-solving experiences. Table 9.5 provides the results from the questionnaire self-assessment of prior programming experiences. This table provides the questions number associated with the questions presented in the questionnaire (See Appendix B.2.2). Table 9.6 presents known problem-solving strategies.

| Group | Size | Q4. Self-Assessment | Comparison To | | Q7. Programming Languages |
|---|---|---|---|---|---|
| | | | Q5. Experts | Q6. Classmates | |
| V1 | 6 | Somewhat Experienced (2.17) | Very Inexperienced (1.5) | Experienced (3.33) | Python (n=1), C++ (n=2) |
| V2 | 4 | Inexperienced (1.8) | Very Inexperienced (1.0) | Inexperienced (2.4) | Mathlab (n=1), Python (n=1) |

TABLE 9.5 Volunteers' Self-Assessment of Prior Programming Experience

Table 9.5 collates the responses for three questions related to prior programming experiences, where volunteers answered with the best Likert scale level (1-5). In the reliability analysis for these questions, Cronbach's alpha was found to be 0.97, a relatively high internal consistency. Included in the table are the analysis of means for study groups' answers. The table shows their answers to questions related to their prior programming, and their abilities compared to their classmates and experts. Table 9.5 shows group V1 ranking their experience higher ($\Delta_{V1V2}$=0.37) than group V2. This is due to two V1 volunteers stating two and three years experience using Python and C++. The remaining V1 volunteers and all volunteers in group V2 stated their programming experience started with this course. When comparing their experience to experts, both groups acknowledged they were very inexperienced (V1=1.5, V2=1.0) compared to experts. Group V1 rated themselves experienced (V1=3.33) compared to their classmates, while group V2 rated themselves a bit below experienced (V2=2.4). The higher rating for V1 was due to the two volunteers with prior programming experience.

The questionnaire asked the volunteers to provide their known problem-solving strategies. Table 9.6 shows the volunteers' responses to the questions using the coding framework that includes SRL (Zimmerman, 1989) and Emotional Regulation (Boekaerts, 2011) strategies. The table classifies the responses by the SRL and Emotional Regulation strategies, providing a description of the strategy and example responses. From the fourteen SRL strategies, four of the fourteen strategies were used in the volunteers' responses, while four out of the five emotional strategies were used in their responses.

The most common SRL strategy cited for both groups (V1=21.1%, V2=15.8%) was *Organising & transforming*, 'student-initiated overt or covert rearrangement of instructional materials to improve learning' (Zimmerman, 1989). Volunteers stated they seek assistance through online resources and peers (V1=2, V2=1), and review notes (V1=1, V2=0). They also identified *Goal-setting & transforming* strategies, such as *"Collect the data for the assignment, and then plan how I will write it down"*, and time management (V1=1, V2=2), for example, *"Try and evenly distribute the workload over the week to give myself the least stress to completion."*

Another questionnaire question asked the volunteers about strategies they use when they become frustrated during the problem-solving process. Table 9.6 shows the response to this question, coding the responses with Emotion Regulation (ER) strategies (Boekaerts, 2011). The table shows example responses from the volunteers, where the most common response was *Taking breaks from solving the problem* ($n_{V1}$=3, $n_{V2}$=3). Most of the students' responses were emotional strategies that could help complete the problem, except for *Denial & distraction*, a procrastination strategy

**Self-Regulated Learning Strategies**

| | | Group | | |
|---|---|---|---|---|
| **SRL Strategy** | **Description** | **V1** | **V2** | **Examples** |
| Organising & transforming | Deconstructs the problem | 4 (21.1%) | 3 (15.8%) | *"I try to divide the problems in parts for easier understanding then solve each of those"* and *"Draw out and expand the problem. Explain each part to myself."* |
| Other | Asks for help from tutors and friends | 3 (15.8%) | 1 (5.3%) | *"First I will try to solve it by myself. After that I will get some help from tutors, online resources, friends."* |
| Goal-setting & planning | Sets goals to complete activities | 2 (10.5%) | 2 (10.5%) | *"Collect the data for the assignment, and then plan how I will write it down."* |
| Seeking information | Uses online resources | 2 (10.5%) | 1 (5.3%) | *"I refer to online resources and books to solve the problem."* |
| Keeping records & monitoring | Reviews notes | 0 (0.0%) | 1 (5.3%) | *"Using techniques I've been taught - refer to notes."* |

**Emotional Regulation Strategies**

| | | Group | | |
|---|---|---|---|---|
| **ER Strategy** | **Description** | **V1** | **V2** | **Examples** |
| Re-appraise situation | Takes breaks from solving the problem | 3 (25%) | 3 (25%) | *"Take a break. Go and do something recreational to get my mind off of it"* and *"Revisit it at a later date if it's that bad. Fresh perspective can help."* |
| | Starts the problem again | 0 (0.0%) | 1 (8.3%) | *"Start again with a blank canvas."* |
| Social support | Asks for help from peers or online resources | 2 (16.7%) | 0 (0.0%) | *"I refer a lot to online resources and books to solve the problem."* |
| Denial & distraction | Leaves the problem entirely | 1 (8.3%) | 0 (0.0%) | *"...if its a very difficult problem, I usually might just leave it based on the time left."* |
| | Moves onto other problems | 0 (0.0%) | 1 (8.3%) | *"Often move onto other problems that need solving and if these are also difficult then rotate through them."* |
| Expressing emotions | Uses positive thinking | 1 (8.3%) | 0 (0.0%) | *"Try to make negative things into positive."* |

TABLE 9.6 Identified SRL and Emotional Regulation Strategies

that can interfere with the learning process (Webster and Hadwin, 2014). None of the responses included negative strategies, such as *Suppressing emotions*. However, it is unclear from the questionnaire whether the volunteers used *Denial & distraction* at the end of the problem-solving process. Using this strategy can promote task avoidance (Boekaerts, 1996), which can result in an incomplete solutions. From the responses, it is unclear whether the participants used used *Denial & distraction* in combination with another strategy, such as *Re-appraise situation*, to complete problems.

### 9.7.2 Think-Aloud Results

| | Volunteers | | | | | |
|---|---|---|---|---|---|---|
| | Group V1 | | Group V2 | | | |
| **Procedural Tasks** | **V1.2** | **V1.4** | **V2.1** | **V2.3** | **V2.4** | **V2.6** |
| Environmental structuring | • | • | • | | | |
| Reads problem description | • | • | • | • | • | |
| Begins programming | | • | • | | | |
| Seek information | | | • | | | |
| Evaluates all strategic plans | | • | | • | | • |
| Identified first strategic plan in activity | • | | • | | • | • |
| Evaluates remaining strategic plans | • | | • | | • | • |
| Arranges strategic plans | • | • | • | • | • | • |
| Self-evaluation | • | | • | • | • | |
| Re-organises incorrect plan | | | • | | • | |
| Initial or revisit self-evaluation | | | • | | • | |
| Submit Answer | • | • | • | • | • | • |
| Begins or continue programming | | | | • | | |

TABLE 9.7 Task Analysis for Groups V1 and V2 for Assignment A6.1

This section reports on the results from the cognitive task analysis of the think-alouds. Table 9.7 presents the first analysis of the procedural steps taken by the volunteers for both groups when interacting with the learning activity in Assignment A6.1. The table shows the steps in temporal order, to show the volunteers' interactions in the order they were performed. The dots in the table denote the volunteer performing the procedural step.

Table 9.7 shows the results from two out of the six volunteers in group V1. Two volunteers excluded from the results completed the activity prior to the think-aloud. These two volunteers identified in the questionnaire having prior programming experience (2-3 years). With the prior programming experience, they might have been ahead in the lab activities, and approached the programming assignments earlier than those volunteers that had no previous experience. Another volunteer in group V1 was excluded from the results because she was behind in her work, and was unable to participate in the think-aloud session for Assignment A6.1. The fourth excluded volunteer's collected data was corrupted and could not be analysed.

All four of the volunteers for group V2 were reported in this section. The reported results from the think-aloud sessions using Assignment A6.1 show all volunteers in both groups V1 and V2 using the *Top-Down* strategy to solve the activity.

| Line | Student's Comments | Analysis |
|---|---|---|
| 1: | `I'm not making a full-on plan when I see them,` | Student acknowledges how he is viewing the activity to help organise the plans. |
| 3: | `besides probably the very last one, because those ones are very obvious.` | Student identifies the last plan as easy, thinking of where it resides in the organised plan. |
| 7: | `But the ones in the middle, they're usually just not the first and they're not the last.` | Student acknowledges mentally reviewing all the plans, but the middle *strategic* plans are not what he selects first to construct the organised plan. |
| 11: | `Once I see the first one, then I move onto the next one.` | Student visits all the plans, and identifies the first one in the organised plan. |
| 14: | `But the other ones aren't already in my head yet.` | Student has not reflected on all the plans. |
| 17: | `I don't put them into an order in my head first.` | Student has not reflected on these plans. |
| 19: | `But the middle ones, I don't order them at all.` | Student acknowledges certain *strategic* plans occur in the middle of the organised plan. |
| 21: | `Basically, if it's more simple then I compare them to each other, but I don't keep track of which one should be in which order until I get to the next one.` | Student uses negotiating tactics to determine the ordering of the middle plans, because they require more programming logic to perform. |

FIGURE 9.6 Protocol Fragment for Student V2.1

A few observations can be made from the results shown in Table 9.7. Three (50%) volunteers use the *Environmental structuring*, an SRL strategy to prepare their environment for better learning, prior to starting other steps in the learning process. The three volunteers adjusted their desktop environment to view the assignment while using the Integrated Development Environment (IDE), preparing the student to view the problem while working on the program. Another observation is that all the volunteers participating in the think-aloud used the *Top-Down* strategy to solve the learning activity. The *Top-Down* strategy was the most commonly applied (E1=41.7%, E2=48.1%) strategy in the quantitative study, demonstrated in Section 9.6. Interacting with the activity while being observed might have motivated the volunteers to use this strategy above others. The remainder of this section presents the underlying thought processes the volunteers used when using the *Top-Down* strategy. The following examples demonstrate the volunteers using the activity in varying problem-solving processes.

| Line | Student's Comments | Analysis |
|---|---|---|
| 1: | I would have at least, maybe, two... Depending on how many options there are, two of these grey boxes, one here and one here. | Student begins to evaluate the overall layout of the activity, to determine plans he needs to arrange. |
| 7: | And so... And then I would, yes, organise what I would think definitely has to come first, what has to come second. | Student begins to negotiate the ordering of plans. |
| 13: | And then, I would look in this and see... I would see how to organise this. | Student evaluates that the selected order is correct. |
| 17: | And if something doesn't add up in this way, it's not possible, I would then check here | Student selects the feedback feature in the activity, to determine if his selection was correct. |
| 21: | and see what's in here that's missing from here, and then I would swap it out. | Student uses the corrective feedback from the activity, to adjust the plan he organised. |

FIGURE 9.7 Protocol Fragment for Student V2.2

To better understand the volunteers' cognitive processes when solving the activity using the *Top-Down* strategy, the second view reports on the cognitive tasks performed by the volunteers, layering their problem-solving process over the procedural steps. This analysis allows a deeper view into the cognitive process when the volunteers solve the activity using the *Top-Down* strategy. The cognitive task analysis shows different approaches performed by the volunteers, and demonstrates different cognitive processes used to complete the activity.

Figures 9.6, 9.7, and 9.8 are protocol fragments from three approaches that show students interacting with the activity without referencing the problem's context. Figures 9.9, 9.10, and 9.11 are examples of how students reference the problem's context when organising the plans. These figures show how participants use internal knowledge to arrange the plans, find relationships between plans during the organisational process, reference the activity to develop the solution's workflow, and adopt trial and error strategies to solve the activity. The figures display the volunteers' transcribed think-alouds in the left-hand column, and an explanation of the their verbalisations in the right-hand column. The figures also display line numbers associated with the transcribed think-alouds, which are used as references when discussing the results.

Figures 9.6 and 9.7 show the cognitive process of Students V2.1 and V2.2, using the *Top-Down* strategy to solve the *Design Strategy Activity* for Assignment A6.1 (See Appendix A.2.3). Student V2.1 demonstrates a student using the activity to think about the organising of plans, where he identifies the plans he is most familiar with first (line 3 Figure 9.6). He refers to the plan as 'obvious'. Identifying known

| Line | Student's Comments | Analysis |
|---|---|---|
| 1: | `I guess it gives an idea, I think` | Student acknowledges that the activity is helping her think more about the problem. |
| 3: | `and what you end up doing is trying to get it all green.` | Student performs trial and error strategy to complete the activity. |
| 6: | `Rather than actually thinking what you want to do it.` | Student acknowledges not thinking about the organisation of the plans to solve the problem. |
| 9: | `The hard part was actually getting a pattern to work.` | Student acknowledges that the trial and error strategy is not an easy problem-solving approach. |

FIGURE 9.8 Protocol Fragment for Student V1.1

plans first demonstrates the student is drawing on prior experiences to help solve the problem (Riley, 1981; Roll et al., 2007). The approach of Student V2.2 is to identify the first plan (line 7 Figure 9.7) instead of recognising a familiar plan. To help with the organisation process, Student V2.2 uses the Parsons problem feedback feature for help (line 17 Figure 9.7), whereas Student V2.1 evaluates the plans closer to decide the order (line 21 Figure 9.6). Student V2.1 approaches the problem using means-ends analysis (Sweller and Levine, 1982), where they examine the plans within the current problem-solving state and determine how the organisation of the plans will help them achieve the final goal state. The protocol fragments presented for Students V2.1 and V2.2 show the activity helped them think about the overall problem, modeling the problem-solving process after an expert's approach (Schulte et al., 2010; Gerdes, Juering, and Heeren, 2012). Both of these volunteers are mentally organising the plans, using the *Top-Down* strategy. From the questionnaire, volunteers cite *Organising & transforming* SRL strategy as the most common (36.9%) approach they use for solving problems. This questionnaire response might suggest their prior usage of this strategy might have resulted in their propensity to organise the plans mentally, instead of the using the learning activity as a supplement for the organisation process.

The protocol fragment presented in Figure 9.8 shows Student V1.1 using a trial and error problem-solving approach for the *Design Strategy Activity* in Assignment A5.1 (See Appendix A.1.2). The student uses the *Experimenting* strategy, where she was observed not thinking about the plans when interacting with the activity. This is the only think-aloud session demonstrating a student struggling while using the activity. The small number (n=1) of students struggling in the think-alouds can be supported by the quantitative results, where the *Experimenting* and Incomplete strategies were the least commonly used to solve the instructional instrument, See Table 9.3.

For this think-aloud, this is the third time Student V1.1 has used the activity, with prior exposure in Assignments A4.1 and A4.2. With her interaction, she acknowledges using the trial and error approach while solving the activity. Though she acknowledges the approach is not the best for understanding the problem (line 6 Figure 9.8), she acknowledges that it encouraged her to think about the problem, stating *"but I guess it gives an idea, I think"*.

Figure 9.9, like Figure 9.6, shows Student V1.2 using relationships between the

| Line | Student's Comments | Analysis |
|---|---|---|
| 1: | `So we'll set up a basic grid with all this. There we go. Well, setting up the basic grid obviously would have to come first.` | Student identifies and arranges the first plan in the list. |
| 6: | `That one has to be last, resetting. We don't want to check the square field yet.` | Student performs a negotiation process to eliminate plans from being next in the ordered list. |
| 9: | `So that probably and then that. So implementing the highlighting, then the filling.` | Student identifies and arranges the next plans in the list. |
| 13: | `Then we have to see when this grid is full and when that happens we have to reset the grid.` | Student identifies the final plans in the ordered list. |
| 17: | `Yes, seems good` | Student validates their work by checking their answer with the assessment in the activity. |

FIGURE 9.9 Protocol Fragment for Student V1.2

plans to help with the organisation process. Unlike Student V2.1 in Figure 9.6, Student V1.2 uses the purpose of the plan to reason its placement in the problem space (lines 3-5 Figure 9.9). Student V1.2 rationalises his selection of the first plan *Set up a basic grid*, stating *"setting up the basic grid obviously would have to come first"*. He also provides reasons for the placement of the last plan, *Reset grid when filled*, stating *"That one has to be last, resetting"*. This statement demonstrates that he understands the completion of the problem. Between identifying the first and last plan, he negotiates the placement of the other plans in the list, stating *"We don't want to check the square field yet"*. Comparing the relationship between the plans in the context of the problem's workflow was how Student V1.2 organised the plans.

| Line | Student's Comments | Analysis |
|---|---|---|
| 1: | `All right, so place ball on game board.` | Student assumes the second plan is the first in the organised list. |
| 3: | `Ooh, actually, construct basic game... Okay, that one is first.` | Student realises his mistake, and adjusts the ordered list. |
| 8: | `Okay, and then try building another and check the ball is placed in hole.` | Student identifies the remaining plans to add to the list. |
| 12: | `Let's have a look at that. Yes, done.` | Student validates work. |

FIGURE 9.10 Protocol Fragment for Student V2.7

Like Student V1.2, Student V2.7, shown in Figure 9.10 for Assignment A6.1 (See Appendix A.1.3), also negotiates the plan order by drawing on the problem's context. Student V1.2 starts with evaluating the first plan (line 1 Figure 9.10) in the randomised list. For this think-aloud session, the first plan was in the randomised list was *Place ball on gameboard*, which he assumed would be the first in the ordered list, but he adjusts his decision after evaluating the other plans in the randomised list, stating *"construct basic game... Okay, that one is first"*. Had the *Place ball on gameboard* plan was presented at the bottom of the randomised list, Student V1.2 might have approached the organisation process differently, possibly reflecting on the remaining plans before making a decision.

| Line | Student's Comments | Analysis |
|---|---|---|
| 1: | `You have to make the basic setup, where the ball's position is, that's what the first thing is.` | Student identifies the initial plan, and begins to associate it with other requirements in the problem. |
| 5: | `And then, yes, because, based on the ball's position, the hole is random, and so the game changes according to that.` | Student continues to bring in other requirements to the problem, such as randomising the location of the initial hole in the game. |
| 10: | `And then, the ball is... You have to check the ball's position to see if it's in there, because there's no point in tracking it around if the ball's already in the hole.` | Student considers ways to make his solution robust. |
| Student begins to interact with the activity by arranging plans. | | |
| 17: | `I think the meaning of this plan, the construct basic game board, is just creating the playing space.` | Student evaluates the first plan to organise. |
| 21: | `Okay. Oh, yes, that... If it's just that then it's... This should be in the front.` | Student is justifying the ordering of the first plan. |
| 24: | `Yes. Ball is placed in the hole. That makes sense. Yes, because I would try either way; it depends on how you see it.` | Student examines how the plans relate through his interpretation of the problem. |

FIGURE 9.11 Protocol Fragment for Student V2.3

The final example, shown in Figure 9.11, demonstrates the student using Assignment A6.1 activity (See Appendix A.1.3) to better understand the problem by reflecting on the plans to construct the workflow of the program. The first part of the protocol fragment shows Student V2.3 describing the workflow by using the plans to initiate thoughts on cases that will make this solution more robust. For example, the *Place ball on gameboard* plan has him consider how the ball will be tracked on the gameboard and cases where tracking is not required (lines 10-16 Figure 9.11).

Upon reaching the end of the workflow, with the game ending with the ball in the hole (lines 15-16 Figure 9.11), Student V2.3 interacts with the plans in the activity to arrange the ordered list, helping him to decide where the plans reside (lines 24-28 Figure 9.11) in the workflow he mentally constructed earlier. This example shows that though the student mentally constructed a workflow for solving the problem, his understanding of how to approach it was different from the correct path. The process of organising the plans through the activity helped adjust his understanding.

### 9.7.3 Interview Results

This section presents the results from the seven narrative interviews that followed the think-aloud sessions. Section 7.4.2 described the process of coding a narrative interview, and the method in which a narrative interview is coded.

Table 9.9 shows the results from the coding framework, along with example coded segments. The results of the coded segments are divided into groups V1 and V2, to determine if students' opinion on the learning activity were different due to prior exposure. The results demonstrate similar skills usage by participants regardless of prior exposure to the activity. The final coding framework contains seven themes. Table 9.9 lists the total number of narrative segments coded to each theme, and the percentage the theme appears in the overall results.

In the results, four out of the fourteen Self-Regulated Learning (SRL) strategies (Zimmerman, 1989) were identified by the participants. No Emotional Regulation strategy (Boekaerts, 2011) was identified by students when using the *Design Strategy Activity*. These strategies are *T1 Environmental structuring* (3.8%), *T2 Organising & transforming* (19.2%), *T3 Self-evaluation* (30.7%), *T4 Goal setting & planning* (15.4%).

Only group V1 had feedback on using the activity, but this may be due to having more practise using it over group V2. The first usability testing session with group V2 was the volunteers first exposure to the activity. Feedback was both positive, *T16 Motivation* (11.5%), and negative, *T18 Too easy* (15.4%), about the activity.

The most common response made about the activity was the SRL strategy *T3 Self-evaluation* (n=8, 30.7%). For example, Student V1.6 stated the activity checks his understanding; while Student V2.3 uses the activity to verify he is done with the assignment, stating *"I'm like, well, that, to me, feels like the end point. I'm going to figure out what steps I need to take to get to the end-point."*

Though only one volunteer acknowledged setting up the environment for learning, in the think-aloud results (described in Section 9.7.2) three (50%) volunteers were performing the procedural task of setting up their desktop space to view the Integrated Development Environment (IDE), and approaching the assignment with the *Design Strategy Activity*, using the *Environmental structuring* SRL strategy. It is possible students did not view setting up their environment as an important step in their learning process, or removed the process of setting up their environment from their conscious mental efforts (Paas, Renkl, and Sweller, 2004).

One student in group V1 acknowledged approaching the problem using Poor Learning Tendencies (PLTs) — habits used by students to compensate for their lack of understanding (Baird and Northfield, 1995) — to complete the activity. She acknowledged using the activity for the purpose of *"making it green"*, a visual feedback from the Parsons problem that denotes success. The motivation to complete the activity over understanding how to solve the problem is a behaviour previously observed when students debug code (Perkins and Martin, 1985), where they alter

| ID | Results | | Theme | Example |
|---|---|---|---|---|
| | V1 | V2 | | |
| T1 | 0 (0.0%) | 1 (3.8%) | Environmental structuring | *"...have it split screen, and have the coding open here...I have a problem of getting distracted and going off course occasionally, so I've gotten into the habit of having what I need to be doing very clear and obvious to me."* Student V1.1 |
| T2 | 2 (7.7%) | 3 (11.5%) | Organising & transforming | *"helps to find the start in the program"* Student V1.3, and *"Helps organise thoughts and what to do next"* Student V1.4 |
| T3 | 3 (11.5%) | 5 (19.2%) | Self-evaluation | *"validate as hints"* Student V1.1, and *"I guess in doing that also showed me what doesn't work in a way"* Student V2.3 |
| T4 | 2 (7.7%) | 2 (7.7%) | Goal setting & planning | *"ordering plans helpful"* Student V1.2, and *"Just what the process was to get to my end goal was the way I was thinking about it, like a logical order of how to get there"* Student V2.3 |
| T16 | 3 (11.5%) | 0 (0.0%) | Motivation | *"easier understanding and easier working while we do the problem"* Student V1.3 |
| T17 | 1 (3.8%) | 0 (0.0%) | Poor Learning Behaviours | *"making it green"* Student V1.1 |
| T18 | 4 (15.4%) | 0 (0.0%) | Too easy | *"too basic"* Student V1.5, and *"would like to have more task"* Student V1.4 |

TABLE 9.9 Narrative Interview Coded Framework

code without understanding the implications to their changes, in the hopes of getting the program to work.

There are comments made by group V1 related to improving the activity, stating the activity was easy (15.4%). Students provided feedback to make the activity more challenging by adding more plans.

The results from the interviews showed the number of plans provided in the learning activity played a part in how the participants interacted with the activity. The participants' comments on the activity being *"too easy"* (T18, V1=15.4%) referenced the amount of plans being arranged. A participant suggested that had there been more plans, the activities would be *"more challenging"* and *"Yes, I probably would use that to organise"*. When designing the instructional instrument used in this study, the plans were designed at a high level, resulting in a number (n=6) of *strategic* plans the participants claimed they could arrange mentally. Future designs could include more *strategic* plans, and observe participants' interactions and experiences using the activity with more detailed plans.

## 9.8 Summary

This chapter presented a study on the *Design Strategy Activity*, the last learning activity in the Codification Pedagogy. This study was conducted over three semesters, with different groups of CS1 students taking part in qualitative and comparative analysis. This chapter described how the learning activity was designed, developed, and tested. The results from this study showed students applying three different strategies for organising *strategic* plans in the learning activity. These strategies could relate to how the students are thinking about the problem. For example, students approaching the activity haphazardly, defined as *Experimenting*, might suggest they are struggling to understand the overall goal of the problem. This is demonstrated by results from the usability testing, where a volunteer admits to solving the activity for the purpose of solving it, and not thinking about the plans in the activity. However, the activity did give this student an idea for solving the problem, which this student might not have done otherwise without the activity. The results from the think-alouds showed the presentation of plans in the initial randomised list was a factor in the organising process. Future work can evaluate how the plan presentation in the randomise list changes the students' organisation process, potentially identifying patterns that were better aligned with steps taken during the design process.

The results from the usability testing show the learning activity supporting students' use of a variety of Self-Regulated Learning (SRL) strategies. Students were shown using the tool to organise and plan prior to coding the solution, and to helping them to better understand the problem by stepping through the workflow. The learning activity was built on Parsons problems, and was the first attempt using Parsons problems during the design process. The study also shows students using the activity to support them through the software development process. SRL strategies used by students during the software design process included self-evaluation, determining the next steps in the design process, and validating that all the programming tasks have been completed.

There are limitations and contextual variables to this work. A limitation to this study is the data collection method used for the usability testing. The data collection might have increased the volunteers' cognitive load when performing the think-alouds, and may have affected how the volunteers interacted with the activity. However, the think-alouds provided an exact method of collecting volunteers' thoughts while problem solving. Another limitation is the small group (n=6) of participants in this study, but the small sample size allowed for in-depth data collection that enabled the participants to share in their own words their experiences while interacting with the *Design Strategy Activity*. Another limitation is the gender disparity (male=83.33%, female=16.67%) for the participants. Future research opportunities can evaluate other approaches to encourage more female students to participate in the usability testing.

There are also other future research opportunities for the *Design Strategy Activity*. The *Design Strategy Activity* was evaluated as a non-compulsory activity. Evaluating as a compulsory activity might generate new strategies for solving the activity, or increase students' usage of a particular strategy. Future research opportunities can included evaluating the activity with different plans. This study was performed using *strategic* plans, but future evaluations could include *tactical* and *implementation* plans. Using these plans might generate different strategies when solving the learning activity. Another future research opportunity includes evaluating the activity

within the functional programming language paradigm, since this study was conducted in the procedural paradigm. The future research could draw comparisons on how students think about the design process based on the programming paradigm.

This chapter presents the last study evaluating individual activities in the Codification Pedagogy. This study allowed for deeper analysis of the *Design Strategy Activity*, providing the opportunity to measure the activity's contribution to the pedagogy. The next chapter brings all the learning activities together, to examine how the entire pedagogy achieves its pedagogical goals.

# Chapter 10

# Pedagogy Evaluation Study

This chapter presents the final study on the Codification Pedagogy, examining the pedagogy in its entirety. Section 10.1 presents the overview of the study. Sections 10.2, 10.3, 10.4, and 10.5 present the study method, analysis, and results. Finally, Section 10.6 presents the summary.

## 10.1 Overview

This is the final study on the Codification Pedagogy, examining the overall pedagogy. The goal of this study is to evaluate how the pedagogical goals are achieved through the lens of academic success and metacognitive awareness. Metacognitive awareness is a process taken by students when thinking about solving a problem, which includes their understanding of where they are at in the problem-solving process (Metcalfe and Shimamura, 1994). This study examines academic success because of the linear relationship between grades and the usage of study skills (Pepe, 2012). Study skills are used to support the students' learning process (Thomas, 1993), where SRL strategies are incorporated into the process when they evaluate their study skills for effectiveness (Winne, 2011). The relationship between academic success and study skills can help identify whether the Codification Pedagogy is achieving the pedagogical goal of supporting students' use of Self-Regulated Learning (SRL) skills.

This study was conducted over three semesters, using quantitative study methods, comparing students receiving the entire pedagogy to those who did not receive the *Questioning* and *Design Strategy* learning activities. When discussing the results from this study, findings from prior studies evaluating the pedagogy are integrated into the discussions.

## 10.2 Methods

This section presents the methods used in this study. Section 10.2.1 presents the participants and study context. Section 10.2.2 describes the development of the first study method, which is a test instrument designed to measure students' metacognitive awareness. Section 10.2.3 describes the second study method, using quantitative methods, to evaluate academic success.

### 10.2.1 Context

The study was conducted over three semesters in an Introductory Programming course offered at the University of Adelaide. Section 4.3 presented the background information on the programming language and learning environment for the studies. This section describes the unique context components for this study.

This study method contains three study groups, shown in Table 10.1. The table contains the study groups' name, cohort size, semester, instructor, and tutor-student ratio. The table shows the educator for each of the semesters. EducatorA was the teacher for study groups E1 and E2, and was involved in constructing all the instructional materials, including the programming assignments containing the Codification Pedagogy. EducatorB taught group C1 during the 2019 semester 1 course, and used the instructional materials developed for the February 2017 semester. The three groups were supported by an equal number of tutors, which were available in the lab environment for assistance. Two tutors were available during the two-hour lab sessions.

The table shows the learning activities provided to each group. Groups E1 and E2 received the three learning activities that comprise the Codification Pedagogy; while group C1 received the one learning activity, the *Assignment Presentation*, because the *Assignment Presentation* reduces the threat to validity by providing all the studies with the same learning environment. Using isomorphic learning environments reduces factors that could influence the study's results.

| | **E1** | **E2** | **C1** |
|---|---|---|---|
| **Pedagogy Activities** | •Assignment Design<br>•Questioning Activity<br>•Design Strategy Activity | •Assignment Design<br>•Questioning Activity<br>•Design Strategy Activity | •Assignment Design |
| **Semester** | February 2017 | August 2018 | February 2019 |
| **Cohort Size** | 249 | 145 | 95 |
| **Instructor** | Instructor A | Instructor A | Instructor B |
| **Tutors** | 1 Tutor:15 Students | | |
| **Course Grading System** | | | |
| **Assignments** | 5 (10%) | 5 (10%) | 5 (10%) |
| **Projects** | 2 (5%) | 2 (5%) | 2 (5%) |
| **Quizzes** | 20 (5%) | 10 (5%) | 10 (5%) |
| **Exams** | 2 (10%) | 2 (10%) | 2 (10%) |
| **Workshops** | 10 (10%) | 10 (10%) | 10 (10%) |
| **Final Exam** | 1 (40%) | 1 (40%) | 1 (40%) |
| **Group Project** | | | |
| **Reports** | 2 (2%) | 2 (2%) | 2 (2%) |
| **Code** | 1 (12%) | 1 (13%) | 1 (13%) |
| **Statement** | – | 1 (1%) | 1 (1%) |
| **Presentation** | 1 (6%) | 1 (4%) | 1 (4%) |

TABLE 10.1 Description of Study Groups

Table 10.1 shows the course grading system for the three study groups, to show grading differences between them. Table 10.1 shows the grading system divided into individual and group assessments. The table presents the number of individual assessments given to the students over the semester, and the weight (%) the assessment has in the final course grade. The table shows the learning activities that comprise the group project, along with the project's contribution (%) to the students' final course grades. The table identifies five assignments used in the grading system. Within these five assignments are eight exercises that are part of the Codification Pedagogy. Section 4.3 provided background on how the exercises are references in

the studies, with Table 4.1 providing links to the Appendix that presents the exercises.

Between groups E1 and E2 are changes to the number of quizzes administered. Group E1 received 20 quizzes, while groups E2 and C1 received ten quizzes; yet for both groups, the quizzes were weighted equally in the grading system. The second change in the grading system occurred between group E1 and E2 studies. Starting with group E2, a new assessment, an individual statement (1%) was integrated into the group project. The individual statement altered the reporting for the group project.

### 10.2.2 Metacognitive Awareness Instrument

This section presents the development of the test instrument used to evaluate students' metacognitive awareness. The test instrument was developed using a Metacognitive Awareness Inventory (MAI) instrument. The Metacognitive Awareness Inventory measures changes in students' metacognitive awareness over the duration of the semester. The Metacognitive Awareness Inventory was designed to allow students to self-assess their SRL strategies, and was selected for this study because it has been shown in the field of Education to reliably measure skill growth (Akin and Abaci, 2007). The Metacognitive Awareness Inventory has been previously used in CS1 identify a correlation between the students' metacognitive awareness and Grade Point Averages (GPA) (Rum and Ismail, 2016), where the GPA includes the final course grade.

The original Metacognitive Awareness Inventory was developed by Schraw and Dennison (1994), consisting of 52 assessment prompts. The 52 prompts are classified into two metacognitive categories: *Knowledge of Cognition* and *Regulation of Cognition*. *Knowledge of Cognition* contains 17 prompts related to acquired knowledge about cognitive processes (van Velzen, 2016), divided into three subcategories: *Declarative* (8 prompts), *Procedural* (4 prompts), and *Conditional* (5 prompts) Knowledge. *Regulation of Cognition* contains 35 prompts related to metacognitve strategies that students use to ensure a goal has been achieved (van Velzen, 2016). This category contains five subcategories: *Planning* (7 prompts), *Information Management* (10 prompts), *Comprehension Monitoring* (7 prompts), *Debugging* (5 prompts), and *Evaluation* (6 prompts).

Metacognitive Awareness Inventory classifies the prompts into categories to identify skill growth in the different areas of metacognition. In developing the test instrument, the categories and subcategories were evaluated to determine prompts that are aligned with this thesis. Some of the categories were excluded from the development of the test instrument because the prompts are unrelated to the research, such as the prompts in the *Debugging* subcategory. Any Metacognitive Awareness Inventory subcategories that are unrelated to the pedagogical goals are excluded from the test instrument.

After researching the purpose of each subcategory, *Procedural* (4 prompts) in the *Knowledge of Cognition* category, and *Planning* (7 prompts) in the *Regulation of Cognition* category were selected for the test instrument. The *Procedural* subcategory relates to knowledge on how to implement learning procedures, to obtain knowledge through problem-solving, and to complete a process. The *Planning* category contains prompts related to goal setting, planning, and allocation of resources by the student prior to their learning.

Using the prompts from the *Procedural* and *Planning* subcategories resulted in 11 prompts for the test instrument. One prompt was removed to bring the tests to

ten prompts, providing one prompt for each factor. The following is the full list of prompts used in the test instrument.

P1. I try to use strategies that have worked in the past.

P2. I have a specific purpose for each strategy I use.

P3. I am aware of what strategies I use when I study.

P4. I find myself using helpful learning strategies automatically.

P5. I pace myself while learning in order to have enough time.

P6. I think about what I really need to learn before I begin a task.

P7. I ask myself questions about the material before I begin.

P8. I think of several ways to solve a problem and choose the best one.

P9. I read instructions carefully before I begin a task.

P10. I organise my time to best accomplish my goals.

A test-retest was used to examine the reliability and validity of the Metacognitive Awareness Inventory when translated to the Turkish language (Akin and Abaci, 2007). For the reliability and validity testing, the researchers administered the same instrument twice to correlate and evaluate the stability over time. The results from the Turkish instrument were compared to the English version, to ensure the translation had the same reliability and validity. The results showed a relativity high internal consistency (.95) over the entire 52 prompt inventory, concluding the instrument is valid and reliable.

The original Metacognitive Awareness Inventory uses 'True' (1 point) and 'False' (0 points) as response options to the survey. For this study, the 'True' and 'False' responses are replaced with a Likert scale, a rating scale that provides students with a range of five to seven levels that best match their opinion to the prompt (Allen and Seaman, 2007). The Likert scale can identify granular changes to students' metacognitive awareness of their skill usage. The Likert scale used in the test instrument has five levels. The levels are 'Strongly Agree' (7 points), 'Agree' (6 points), 'Neither Agree nor Disagree' (4 points), 'Disagree' (2 points), and 'Strongly Disagree' (1 point). See Appendix B.2.1 for the tests used in this study.

To measure changes in students' metacognitive awareness, the test instrument was administered as a pre-post test. For each study group, the pre-test was administered during the second week of the semester, while the post-test was administered three weeks before the final exams (week 14).

Group E1 was given two weeks to complete the pre-post tests, using Google Forms (Google LLC, 2019a), an online service for constructing and administering surveys. Group E1 responses were exported from Google Forms and imported into a Google Sheets spreadsheet (Google LLC, 2019b) for further analysis. Groups E2 and C1 received the in paper format during class because of the low participation rate from group E1 receiving the survey through an online service. The change from administering the survey from online to paper format is explained further in Section 10.4. Students were asked to complete the tests during class, with completed responses collected by the course educator. Test responses from groups E2 and C1 were transcribed in a spreadsheet, translating students' Likert level to a numeric value.

### 10.2.3 Academic Study Design

This section describes the second study method, a comparative analysis (Pickvance, 2001) study that measures the students' academic success, since there is a linear relationship between GPAs and study skills, where the GPA contains the final course grades for all courses taken during the semester. A previous study (Pepe, 2012) concluded a relationship between GPA and study skills, but this study was performed in a non-CS course and at a Turkish institution, where the assignments and culture is different than this study performed with CS programming assignments at an Australian institution. To determine the relationship between GPAs and study skills for the study presented in this chapter, the Introductory to Programming final course grade, assignment completion rate, and grades for eight completed assignments as measurements of academic success were used. Measuring academic success might determine students' skills usage, which can demonstrate whether the thesis achieves its pedagogical goals. Comparing the grades of the programming assignments might demonstrate how the pedagogy might influence students' completion of the assignments and how well they understood the problem.

The data for the comparative analysis was contained in the Canvas Learning Management System (LMS), where the educator recorded the students' final course grade and the tutors recorded grades for the eight practical programming assignments. At the end of the semester for each study group, students' final course grades and grades for the practical programming assignments were exported from the LMS and imported into a Google Sheets spreadsheet (Google LLC, 2019a) for further analysis.

## 10.3 Analysis

This section describes the analysis performed. Section 10.3.1 presents the comparative analysis performed using the test instrument. Section 10.3.2 presents the analysis for measuring academic success.

### 10.3.1 Test Instrument Analysis

This section describes the three data analysis approaches performed on students' responses to the test instrument. The first method was analysis of means (Boone Jr and Boone, 2012), identifying the central tendency for each prompt in the test instrument. The analysis of means was performed in a spreadsheet for the ten prompts in groups E1, E2, and C1 pre-post tests. The result of the analysis of means was a central tendency for the ten prompts. The central tendencies were placed in a spreadsheet to create a comparative matrix. The central tendencies for the study groups' pre-post tests were compared, to measure change in metacognitive awareness. The comparison results were classified into three categories: positive change in the students' metacognitive awareness, negative change in their awareness, and no change. Positive change means that the study group 'agreed' more with (felt positive about) the metacognitive skill usage, while negative change showed a trend towards disagreeing with using the skill. No change means the students' opinions on the skill usage did not change over the course of the semester.

The second analysis performed on the test responses was variable-oriented analysis (Bergman and Trost, 2006), a type of cross-case analysis to compare results across the groups. Variable-oriented analysis was performed on the ten prompts, to determine if the students' metacognitive awareness was influenced by the pedagogy. The

central frequencies for all test instruments were collated into a spreadsheet, to create a cross-table for comparison of each prompt across the groups. Any cross-table differences were further examined by evaluating the prompt statement, to determine how the statement relates to the pedagogical goals.

The final analysis performed was statistical analysis (Greasley, 2008), to determine if there was any statistical difference in the groups' responses. The statistical analysis was performed on the ten prompts for all test instruments, where the responses were exported from a spreadsheet and imported into IBM SPSS Statistics version 2.5 for analysis. Prompts with statistical significance were further evaluated, because the statistical differences might imply an outside influence on the students' awareness of metacognitive skill usage.

### 10.3.2 Analysis of Academic Success

This section describes the data analysis performed on the data related to students' academic success. The analysis was performed on the final course grade, and the grades and completion rates for the practical programming assignments. Analysis of means was performed on the final course grade for the students within the study groups. The results from the analysis of means allowed for comparison of the cohort's overall final course grades, to determine whether the pedagogy influenced the cohorts' final grades. Statistical analysis was applied to the final course data, to determine if there is any statistical differences in the data across the study groups, and whether the pedagogy might have influenced the students' overall grades. Statistical analysis was performed by importing the final course grades from a spreadsheet, and using IBM SPSS Statistics version 2.5.

To analyse academic success for the completion rate and grades of the programming assignments, time-series analysis was used. Time-series analysis is a method of statistical analysis that enables observation to be made on a variable at different points in time (Chatfield, 2004). Time-series analysis was performed on the grades and completion rates of eight practical programming assignments. The analysis constructs models that visualise the pedagogy's potential influence on the completion rates and grades. The time-series analysis was performed within a spreadsheet, using analysis of means for the grades and completion rates of eight assignments of the three study groups.

## 10.4 Metacognitive Awareness Results

This section reports on the results of analysing the metacognitive awareness data to evaluate change in students' metacognitive awareness. Table 10.2 shows the results from the means and statistical analysis for the three study groups. The table shows the number of students enrolled in the course at the time of the test was taken, such as $T_{E1}$=283 for E1's pre-test, and the percentage of students participating in each test, such as $A_{E1}$=99 for E1's pre-test. Results from the pre-tests contain responses from students who withdrew from the course. For example, group C1 shows 120 responses in the pre-test, but 95 students completing the course.

Table 10.2 shows group E1 with a low participation rate for the pre-post tests ($Pre_{E1}$=34.98%, $Post_{E1}$=8.84%) and was the reason the pre-post tests were changed from an online survey to paper format administered during class. This change was to encourage a higher participation rate from groups E2 and C1. Results verified that administering the tests in paper format increased student participation. For example, group E2 had a 92.55% participation rate for the pre-test, and a 99.31%

| | E1 | | | E2 | | | C1 | | |
|---|---|---|---|---|---|---|---|---|---|
| **Prompt** | **Pre** $T_{E1}$=283 $A_{E1}$=99 (34.98%) | **Post** $T_{E1}$=249 $A_{E1}$=22 (8.84%) | **Sig** | **Pre** $T_{E2}$=161 $A_{E2}$=149 (92.55%) | **Post** $T_{E2}$=145 $A_{E2}$=144 (99.31%) | **Sig** | **Pre** $T_{C1}$=235 $A_{C1}$=120 (51.06%) | **Post** $T_{C1}$=95 $A_{C1}$=73 (79.35%) | **Sig** |
| P1. Past strategies | 2.30 | 2.55 | ns | 5.92 | 6.36 | *** | 6.02 | 6.38 | * |
| P2. Purposeful strategy | 2.69 | 2.77 | ns | 5.38 | 5.79 | ** | 5.37 | 5.73 | * |
| P3. Strategy awareness | 2.44 | 2.64 | ns | 5.42 | 5.92 | *** | 5.28 | 5.55 | ns |
| P4. Strategy automatically | 2.83 | 2.77 | ns | 5.03 | 5.53 | ** | 4.78 | 5.23 | * |
| P5. Pace myself | 3.17 | 3.23 | ns | 5.24 | 5.17 | ns | 4.61 | 4.51 | ns |
| P6. Need to learn | 2.82 | 3.05 | ns | 5.46 | 5.52 | ns | 5.19 | 5.04 | ns |
| P7. Ask myself questions | 3.44 | 3.55 | ns | 4.92 | 5.32 | * | 4.51 | 4.85 | ns |
| P8. Choose best strategy | 2.97 | 3.45 | ns | 5.43 | 5.45 | ns | 5.15 | 5.34 | ns |
| P9. Read instructions first | 2.38 | 2.77 | ns | 5.81 | 5.85 | ns | 5.70 | 4.23 | * |
| P10. Organise my time | 3.16 | 3.32 | ns | 5.44 | 5.44 | ns | 4.53 | 4.55 | ns |

*** Significant at $p < 0.001$
** Significant at $p < 0.01$
* Significant at $p < 0.05$
ns Not Significant

TABLE 10.2 Means and Statistical Analysis Results for Pre-Post Tests

participation rate for the post-test. Group E1's low participation rate does not reflect the overall cohort's opinions on the metacognitive awareness prompts; therefore, the results from this group are not reported in the results. The remainder of this section presents an overview of the results, then further discusses observations made from the analysis.

Table 10.2 shows the central frequencies for the prompts, which helped to draw out further discussion on the changes. Table 10.2 also shows the study groups' t-tests from the statistical analysis. The table shows statistical significance for prompts P1, P2, P3, P4, P7, and P9. These results motivated further inspection on these prompts.

Figure 10.1 shows the results from the comparative analysis, demonstrating the change in central frequency for the study groups E2 and C1. The figure provides

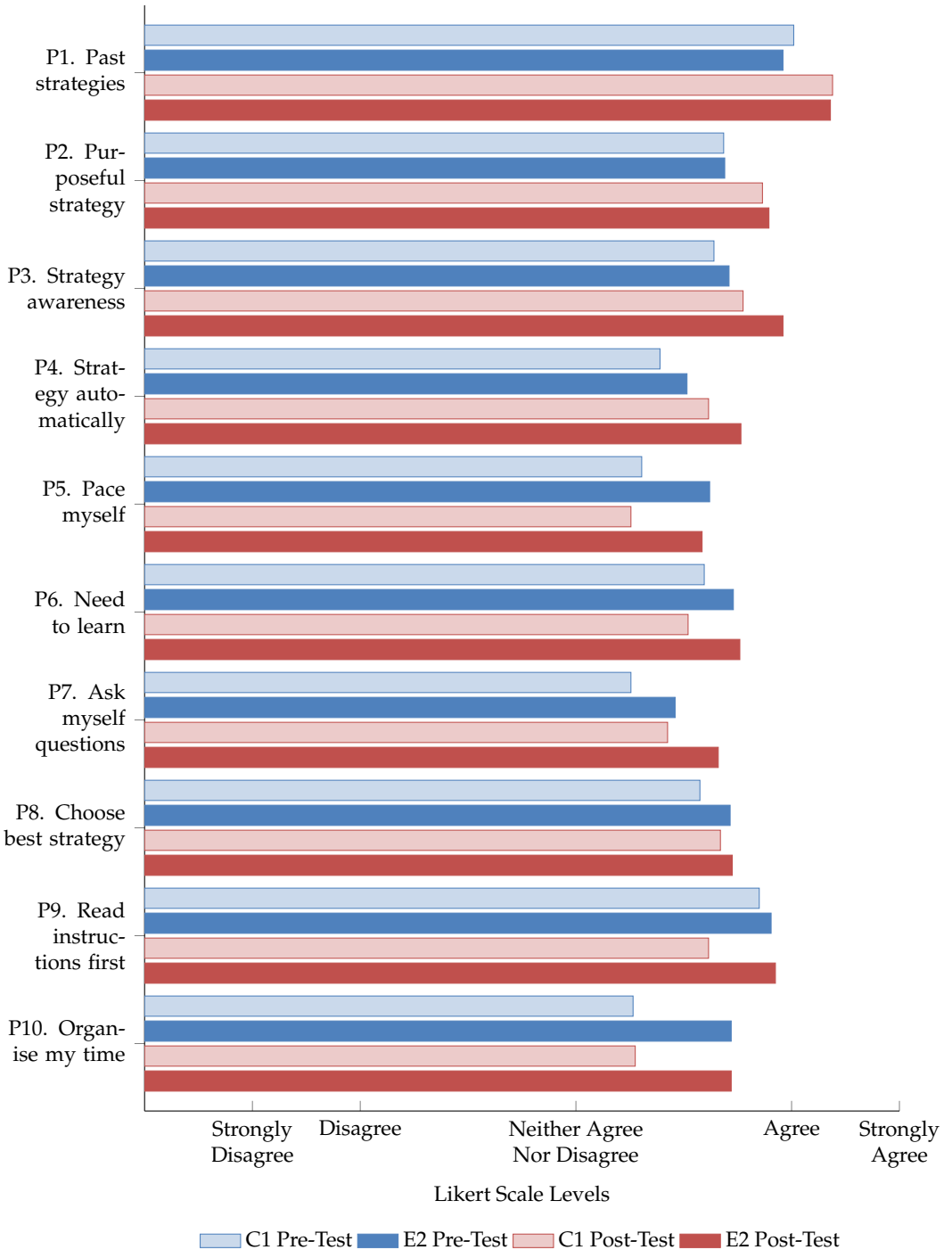



FIGURE 10.1 Comparing the Results from Groups E2 and C1 Test Instruments

an overview of the groups' changes in their metacognitive awareness. The results through this view provide comparisons for groups' E2 and C1 results. Section 10.4.1 presents the diverging opinions between the two groups. Section 10.4.2 shows skills that had an increase in both groups' metacognitive awareness. Section 10.4.3 discusses strategies that remain unchanged by the study groups.

### 10.4.1 Diverging Opinions on Metacognitive Skills

This section seeks to understand the diverging opinions between groups E2 and C1, to determine whether the Codification Pedagogy influenced these changes. This section presents observations made on Metacognitive Awareness Inventory (MAI) prompts that had diverging opinions from groups E2 and C1. The following four Metacognitive Awareness Inventory prompts show differences in opinions.

- **P2.** I have a specific purpose for each strategy I use.
- **P3.** I am aware of what strategies I use when I study.
- **P8.** I think of several ways to solve a problem and choose the best one.
- **P9.** I read instructions before I begin a task.

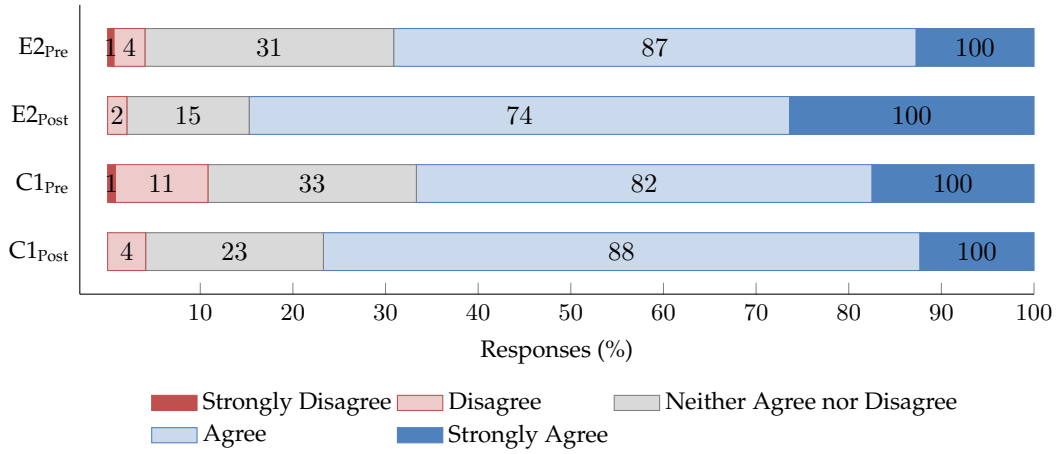



FIGURE 10.2 Groups E2 and C1 Results for P3: I am aware of what strategies I use when I study.

Figure 10.1 shows prompts P2 and P3 with similar responses to strategy usage. P3 is discussed in greater detail, since the two prompts relate to the skill usage. Figure 10.2 shows group E2 increasing their strong agreement ($\Delta$E2=13.65%) for prompt P3. Group C1 had a different opinion on this prompt than group E2. Group C1 decreased their strong agreement ($\Delta_{C1}$=-5.17%) on this prompt. The positive increase by group E2 may be the students' awareness of their motivation to initiate strategies during the learning process, engaging self-awareness that occurs during the Self-Regulated Learning (SRL) *forethought* stage (Zimmerman, 2000). *Forethought* is the first SRL stage that gives the student the opportunity to analyse the problems goals. The *Design Strategy Activity* study, presented in Chapter 9, demonstrated that the learning activity helped students through the *forethought* stage, by organising students' thoughts to approach the problem-solving process. Giving students the *Design Strategy Activity* might have helped them identify design strategies to use in the *forethought* stage, and taught them to apply design strategies to other problem-solving contexts. The response from group C1 to prompt P3 might imply their self-awareness is at the beginner SRL learner's level, where they might not have the awareness to identify the specific goals, but understand the general outcome for solving the problem.

Prompt P8 was another prompt that showed diverging opinions between groups E2 and C1, and is also related to selecting the best problem-solving strategy for a

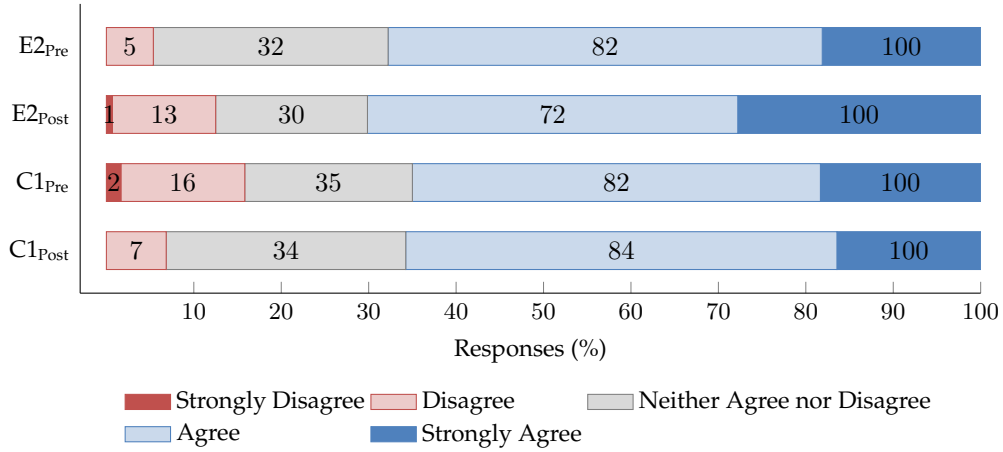



FIGURE 10.3 Groups E2 and C1 Results for P8: I think of several ways to solve a problem and choose the best one.

given situation. Figure 10.3 shows group E2 increasing ($\Delta_{E2}$9.66%) agreement to this prompt, while group C1 showed a reduction ($\Delta_{C1}$=-1.89%) in the students' agreement. This result might suggest group C1 placing less importance on the metacognitive skill, suggesting the Codification Pedagogy provided support for the organising and transforming SRL strategy for group E2, to recognise a need for correct strategies. When used in the CS space, students develop a plan prior to implementing a solution (Garcia, Falkner, and Vivian, 2018), which relates to identifying a strategy that works. Group E1's responses suggest that they are thinking about the problem-solving process before the implementation process, potentially helping them to develop their 'organising and transforming' SRL strategy earlier. Group C1 might be showing behaviours common to CS1 students, where they postpone thinking about the planning process until they begin coding (Falkner, Vivian, and Falkner, 2014).

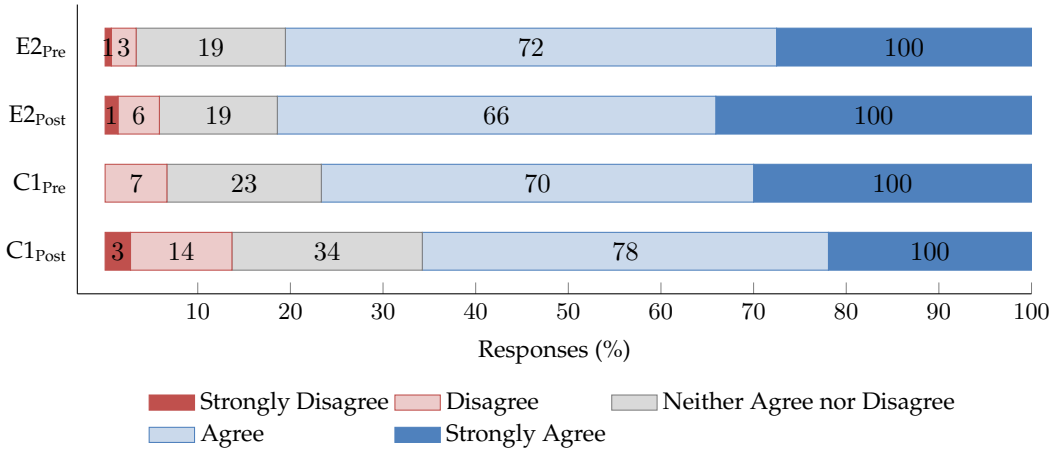



FIGURE 10.4 Groups E2 and C1 Results for P9: I read instructions carefully before I begin a task.

Prompt P9, shown in Figure 10.4, demonstrates groups E2 and C1 having similar central tendencies (E2=5.81, C1=5.70) about reading instructions carefully at the beginning of the semester; but their opinions differing at the end of the semester. Figure 10.4 shows E2 raising their strong agreement ($\Delta_{E2}$=6.56%), while C1 decreases theirs ($\Delta_{C1}$=-8.08%). The difference in opinion might be related to the *Questioning Activity*. In Chapter 8, the results showed students interacting with the *Questioning*

*Activity*, using cognitive levels that promote critical thinking skills to reflect on the problem, where reflection on text can positively influence the learner's reading comprehension (Facione, 2015). Reading comprehension involves students using skills with their internal knowledge, to improve their comprehension (Collins, Gambrell, and Pressley, 2002). By encouraging students to engage their critical thinking skills within the *Questioning Activity*, the activity might have encouraged the use of critical thinking skills for reading.

When combining 'Strongly Disagree' and 'Disagree' responses for group C1, there is an increase in disagreement ($\Delta_{C1}$=7.02%). Group C1's response could suggest that without the *Questioning Activity*, students were not encouraged read the assignment carefully, either through the activity promoting SRL strategy development, or the *Questioning Activity* making the problem more interesting. Though group C1 received the *Assignment Presentation* in the same scaffolded learning environment as E2, the *Assignment Presentation* activity might not have helped group C1 recognise the importance of reading the problem.

### 10.4.2 Increased Positive Opinions on Metacognitive Skills

This section discusses the prompt that shows positive gains from both groups E1 and C1. One Metacognitive Awareness Inventory prompt demonstrates similar responses from both study groups: *Prompt P1, I try to use strategies that have worked in the past*. Figure 10.5 shows both groups strongly agreeing, increasing ($\Delta_{E1}$=14.87%, $\Delta_{C1}$=15.38%) their opinion over the course of the semester. These findings are supported by prior work, where CS1 students use their prior knowledge to solve problems (Roll et al., 2007). These results demonstrate that students rely on past strategies, regardless of support from the pedagogy. The pre-test results in Chapter 9 also show students early in their learning being aware of using past problem-solving strategies. Section 9.7.1 shows students describing known strategies when encountering frustrating problem-solving situations. Perhaps more practice solving programming assignments would reinforce their opinion that it's helpful to use these skills in the problem-solving process.

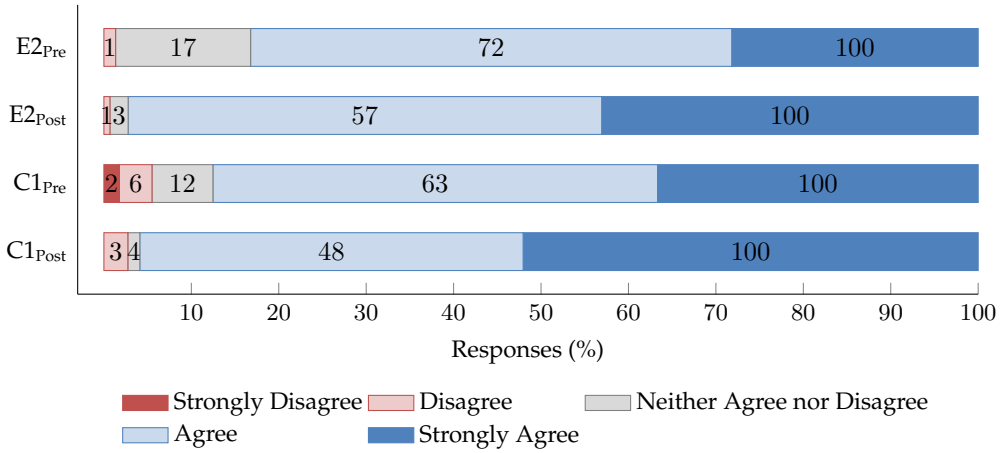



FIGURE 10.5 Groups E2 and C1 Results for P1: I try to use strategies that have worked in the past.

### 10.4.3 Unchanged Opinions on Metacognitive Skills

This section describes the Metacognitive Awareness Inventory prompt that did not change for both study groups. One Metacognitive Awareness Inventory prompt show the groups' opinions not changing over the duration of the semester. Figure 10.6 shows the distribution of Likert responses for *Prompt P10, I organise my time to best accomplish my goals*.

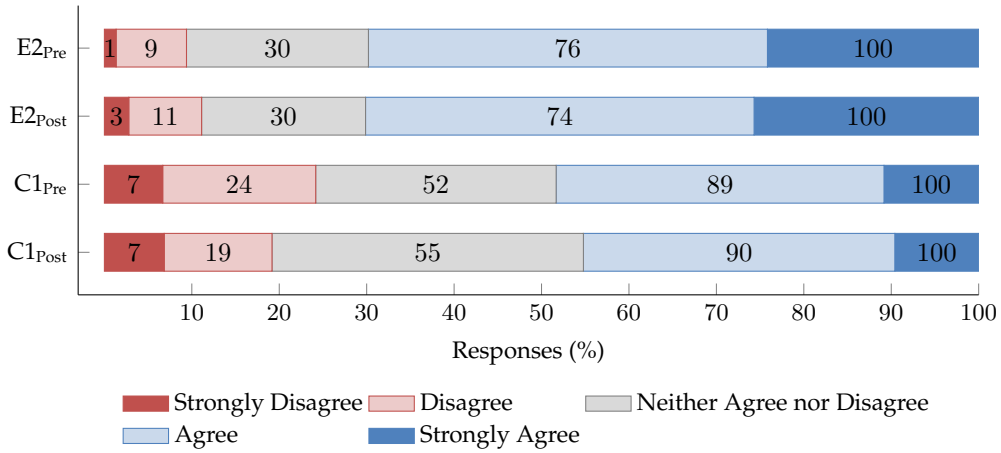



FIGURE 10.6 Groups E2 and C1 Results for P10: I organise my time to best accomplish my goals.

Prompt P10 had no change ($E2_{pre}$=5.44, $E2_{post}$=5.44) for group E2 and slight change ($C1_{pre}$=4.53, $C1_{post}$=4.55) from group C1, with an increase ($\Delta_{C1}$=8.12%) in the 'Neither agree nor disagree' option. The results suggest the Codification Pedagogy did not raise students' awareness in time management. The *Design Strategy Activity* is designed to encourage students to develop SRL strategies related to goal-setting and planning. Students have previously recognised time management as a key factor for success for goal-setting and planning (Falkner, Vivian, and Falkner, 2014); however, in this study, students did not identify the importance of time management. Perhaps students were not aware of their use of time management in this study, since students use their time effectively when engaging in SRL (Schraw, K.Crippen, and Hartley, 2006).

## 10.5 Academic Success Results

This section reports on the results from measuring academic success. Three data points are reported in this section: final course grade, and grades and completion rates for the practical programming assignments.

| | | | t-test Results | | | |
|---|---|---|---|---|---|---|
| **Group** | **Cohort Size** | **Final Course Grade (%)** | **SD** | **t** | **df** | **sig** |
| E1 | 249 | 87.33% | 8.57 | 11.79 | 343 | 0.0001 |
| E2 | 145 | 76.41% | 13.87 | 9.56 | 385 | 0.0001 |
| C1 | 95 | 68.30% | 28.44 | 4.59 | 238 | 0.0001 |

TABLE 10.3 Final Course Grade Comparisons for Study Groups

Table 10.3 presents the results from analysing the final course grades, the mean for the three groups, the number of students completing the course, and the results from the statistical analysis. Comparing the final course grade for the groups showed groups E1 (87.33%) and E2 (76.41%) performed better than the control group C1 (68.30%), where the difference between E1 and C1 is 19.03% and the difference between E2 and C1 is 8.11%. However, the cohort size and change in the instructor for group C1 might have influenced the final grade, since there are fewer students to discuss problems and the instructor might have changed the presentation of the learning materials.

There are other possible reasons for these findings. One potential reason is the small contribution (10%) the pedagogy had on the final grade, described in Section 10.2.1. Table 10.1 shows a variety of assessments that comprise the students' final course grades, and these other course assessments, such as workshops and quizzes, were not presenting using the assignment presentation developed with the pedagogy. The presentation of the other assessments might have encouraged students to use different behaviours, influencing the results from those activities.

Table 10.3 presents the results from the statistical analysis. The results shows statistical significance, suggesting outside influence for the final grades. A possible cause for these findings is the change made to the grading system between groups E1 and E2. As described in Section 10.2.1, the t-tests show significant differences between the study groups' final course grades. The statistical difference in the final grades might be due to the 10% difference in final course grades across groups. A reason for the difference between groups E1 and E2 might be the changes made to the grading system between the groups. As described in Section 10.1, Table 10.3 shows a reduction (50%) of quizzes starting with group E2, and the introduction of a new assessment, a personal statement for the group project. A reason for the differences between C1 and the other experiment groups, E1 and E2, might be the influence of the Codification Pedagogy.

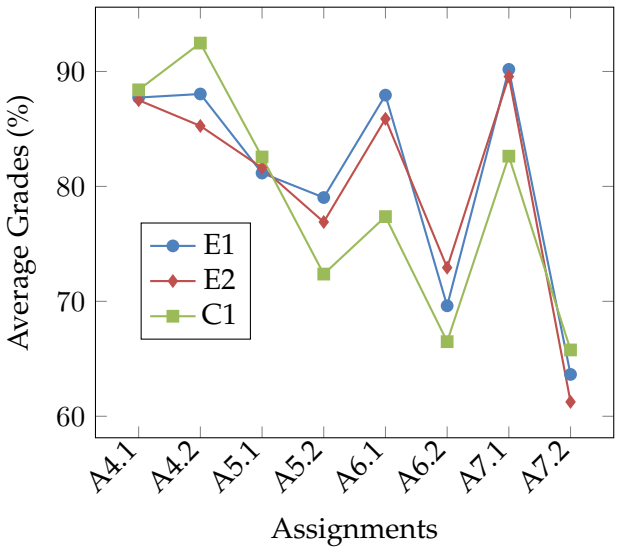



FIGURE 10.7 Overall Average of Programming Assignments

The second view of academic success examines the grades for the practical programming assignments containing the Codification Pedagogy. Figure 10.7 shows the average grade for the eight practical programming assignments, with the figure plotting the assignment averages over time. The figure does not show significant

improvements from groups E1 and E2 over group C1. Higher grades in the practical programming assignments would imply the complete solutions from group E1 and E2 contained more completed tasks. However, the results are not consistent enough across assignments to claim the pedagogy encouraged higher programming task completion.

Exposing study groups E1 and E2 to the *Questioning* and *Design Strategies* activities might have supported SRL strategies, such as goal-setting and planning (Zimmerman, 1989), which would have improved their ability to identify programming tasks. Though the findings presented in Section 10.4.1 supports the groups E1 and E2 are recognising tasks in the problem description, the support does not seem constant for all assignments. Inconsistent grades for the assignments might suggest an issue with the problem's context, hindering students' reading or program comprehension. In Section 10.4.1, group E2 shows higher agreement for reading instructions carefully before beginning the tasks, which can reduce students' misconceptions (Gick, 1986). Misconceptions might contribute to missed or incomplete programming tasks, potentially lowering the assignment grade. With the average grades showing inconsistent improvements, this might suggest the problem's context could be a factor.

The completion rate for practical programming assignments containing the Codification Pedagogy was also analysed, shown in Figure 10.8. The figure provides a visualisation of the assignments completed by groups E1, E2, and C1. This results show a higher completion rate for groups E1 and E2. Unlike the distribution of grades for the assignments, shown in Figure 10.7, Figure 10.8 shows the completion rate for groups E1 and E2 consistently higher (10.29%-19.35%) than group C1. These results suggest the learning activities in the Codification Pedagogy might have helped groups E1 and E2 persevere in completing the assignments. These results are strengthened by the findings from the metacognitive awareness study reported in Section 10.4.1. Section 10.4.1 reported on students exposed to the learning activities first having higher awareness using strategies to solve problems, and taking the time to find the best problem-solving strategy. These behaviours have been shown to be common among experts (Schoenfeld, 1992). Another potential reason for these findings is suggested by the results presented in Section 10.4.1, where E2 expressed higher awareness in reading instructions carefully, which could lead to better understanding and possibly more complete programming tasks, leading to a greater likelihood of completing the assignment.

The findings for the completion rate show all the groups had difficulties completing Assignment A7.2. Groups E1 and E2 had over 90% completion rate, except for Assignment A7.2 (E1=78.71%, E2=77.92%), while group C1 had a completion rate of 68.42% for this assignment. A potential reason for the low completion rate for Assignment A7.2 is the non-contextualisation of the problem description. Assignment A7.2 asked students to perform transformation operations on matrices. The assignment administered to group E1 and C1 asked students to transpose a 2D array (see Appendix A.2.4), while group E2 involved rotating a 2D array (see Appendix A.1.4). When comparing the problem description for Assignment A7.2 with the other practical programming assignments, Assignment A7.2 is the only problem without contextualisation. Contextualisation has been shown to help with students' motivation during problem solving (Lovellette et al., 2017), which could have contributed towards the students' completion of the earlier problems, or the higher workload towards the end of the semester might have reduced their participation in Assignment A7.2 (Rocca, 2010). Another potential reason for the lower completion rate for Assignment A7.2 is the higher use of mathematics skills to solve this problem. Matrix transformations use linear algebra concepts, and students tend not to transfer their

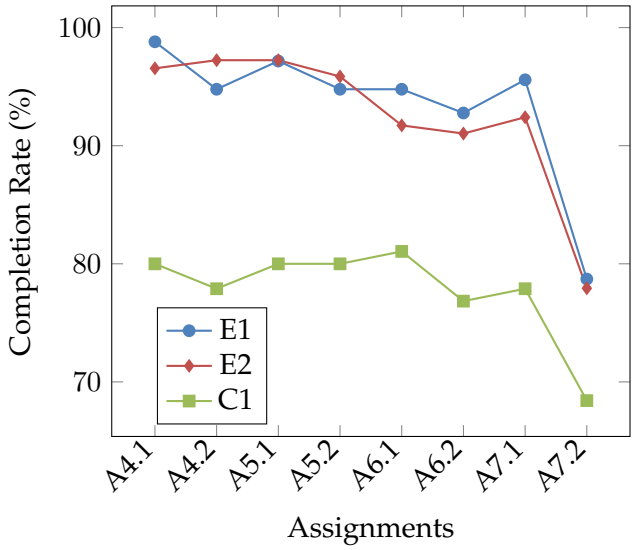



FIGURE 10.8 Programming Assignment Completion Rate

mathematical skills to other disciplines (Britton et al., 2007).

## 10.6 Summary

This chapter presented the final study, which examines the entire pedagogy. The study was conducted over three semesters, to draw comparisons between the study groups that received the learning activities to a control group that did not. This study compared students' metacognitive awareness, to determine how the pedagogy might have influenced their perception of metacognitive skills usage. This study also compared academic success, to determine whether the pedagogy had any influence in the students' learning gains. The results showed higher completion rates for groups receiving the learning activities, which might suggest that the learning activities help students develop skills for better understanding and solving the problem, a conclusion which is strengthened by the groups' responses to the Metacognitive Awareness Inventory also presented in this chapter.

This study was conducted over three semesters, gathering considerable quantitative data. Studying across cohorts provided the opportunity to compare groups receiving and not receiving the pedagogy. Comparison across cohorts helped to identify trends and to demonstrate the pedagogy's influence on the students' learning, such as the higher completion rate in the practical programming assignments.

There are limitations and threats to validity for this study. Group E1 has a different grading system than groups E2 and C1. Because two groups used the same grading system, this reduces the concerns about the results, but still raised as a treat to validity since group E1's grading system might have influenced the results for their final course grades. The different instructor for group C1 may have influenced the group's final course grade. A threat to validity is drawing comparisons between the pre-post tests' results, where the participation rate for the tests had different distributions between the compared groups. Group C1's participation rate (pre-test: 51.06%, post-test: 79.35%) was less than group E2 (pre-test: 92.55%, post-test: 99.31%). Future research can administer the tests to produce similar participation rate between the groups, to draw more accurate conclusions.

There are future research opportunities from this study. More research is needed to determine how students use the learning activities without submitting their answers. This research can strengthen the findings with higher completion rate in groups receiving the learning activities. Another future research opportunity can examine how the pedagogy helps students in other parts of the problem-solving process, such as the debugging process. This research could identify stages in the software development process that the students engage the learning activities, potentially finding students using different skills at certain stages within the same activity. This study showed no change in students' perception of time management skills. A future research opportunity is to include instructions in the *Design Strategy Activity* to encourage students to use strategic plans as guidelines for time management. Additional guidance might help raise students' awareness on time management. As stated in Section 10.5, other programming assessments, such as workshops and quizzes, were not presented using design treatments identified in Chapter 5. Future research opportunities could include applying the assignment design to all assessments, and measuring the influence the unified presentation has on the students' academic success.

# Chapter 11

# Conclusion

Designing a solution to a programming problem is a necessary part of the CS problem-solving process (Hoadley and Cox, 2009). However, many introductory programming (CS1) students do not always devote time to designing solutions to their programming problems (Pintrich, Berger, and Stemmer, 1987; Rist, 1995). Instead, CS1 students sometimes rely on past problem-solving approaches, and adopt Poor Learning Tendencies (PLTs) when these approaches fail (Baird and Northfield, 1995). Poor Learning Tendencies encourage students to focus on surface aspects of the problem (Adelson, 1984), resulting in misunderstandings (Collins, Brown, and Holum, 1991). This thesis investigates a pedagogy integrated into procedural CS1 assignments that promotes Good Learning Behaviours (GLBs) and discourages Poor Learning Tendencies. The pedagogy is designed using theories in program comprehension, critical thinking skills, and design knowledge and promotes Good Learning Behaviours by demonstrating the use of self-reflection and Self-Regulated Learning (SRL) strategies, to better understand the problem and to organise a plan prior to software implementation. The pedagogy is grounded in Cognitive Apprenticeship to support students early in their learning through a scaffolded learning environment. As students gain programming experience, the learning support within the pedagogy is reduced to encourage them to become independent learners.

In this thesis are six chapters presenting studies on the pedagogy. These studies examine the pedagogy's learning activities in depth, to determine how each activity contributes to the pedagogical goals: improve students' program comprehension and support their use of SRL strategies when designing a program solution. Chapter 5 collates assignment design treatments that supported the development of well-formed programming assignments. Chapter 6 presents a comparative study, comparing completed programming tasks between the assignment presentation developed in this thesis and other presentation approaches. Chapter 7 uses a qualitative study to gain more insight into students' perception of the assignment presentation. Chapter 8 evaluates the pedagogy's first intervention, a questioning activity designed to engage critical thinking skills. Chapter 9 discusses the influence the second intervention had on students' design knowledge, through the use of a design-based Parsons problem. Chapter 10 examines how the entire pedagogy influenced students' learning and their usage awareness of problem-solving skills. This chapter presents the contributions made by this thesis, along with concluding remarks about the pedagogy.

## 11.1 Contributions

This section presents the contributions made by the pedagogy. The contributions are outcomes from answering the research questions initially defined in the introduction, Chapter 1. Table 11.1 shows the research questions and hypotheses that

| Research Questions | Hypothesis |
|---|---|
| **Program Comprehension:**<br>**RQ1.1:** How does scaffolding the assignment presentation influence the student's ability to identify goals and subgoals necessary to complete a procedural programming problem?<br><br>**RQ1.2:** What presentation treatments within the programming assignments support students in their understanding of the programming problem? | **H1.1:** Scaffolding the assignment presentation will guide students in identifying the problem's goals and subgoals, providing them support in identifying a starting point for developing a programming solution.<br>**H1.2:** Itemising goals and subgoals as a treatment in the assignment presentation will help students identify and apply the goals and subgoals during the programming process. |
| **Critical Thinking Skills:**<br>**RQ2.1:** Does encouraging questions in an online CS1 learning environment promote the expected cognitive levels from students when answering the questions? | **H2.1:** Providing a questioning activity as an intervention to a programming assignment will help students internally reflect on and apply their knowledge when solving the current problem. |
| **Design Knowledge:**<br>**RQ3.1:** How do students use Parsons problems during the design process for solving CS1 procedural programming assignments?<br><br>**RQ3.2:** What Self-Regulated Learning (SRL) strategies are supported by Parsons problems used as a design-based intervention for programming assignments? | **H3.1:** Parsons problems will promote internal reflection, enabling the student to form a mental model on how the problem's plans interact with each other prior to developing a programming solution.<br>**H3.2:** The Parsons problem will promote organisation and planning SRL strategies that are necessary to solve the programming problem. |

TABLE 11.1 Summary of Research Questions and Hypotheses

guided the development of the pedagogy and the six studies. The research questions are divided into the pedagogy's three research areas: program comprehension, critical thinking skills, and design knowledge.

A goal of this thesis is to support students' learning and practice of design-based skills and strategies that help them better understand a problem and develop a programming plan. The hypothesis is the pedagogy would help students use Self-Regulated Learning (SRL) strategies. The results from the studies showed students using SRL strategies within the pedagogical activities. Treatments within the assignment presentation helped students identify the problem's goals by deconstructing the problem into subgoals and presenting them in list format. The treatment supported students' use of SRL strategies by enabling them to use the list to validate their solutions. The questioning activity encouraged students to analyse the assignment, promoting self-reflection that reduced misconceptions. The Parsons problem enabled students to practise SRL strategies when they were uncertain with the next steps in the problem-solving process and when they validated their solutions upon completion.

In Chapter 2, a description (See Figure 2.3) of the Codification Pedagogy was provided, layering the pedagogical goals over a problem-solving process model (Gick,

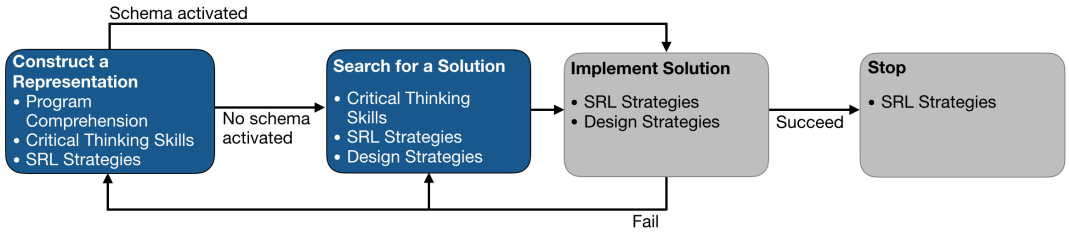



FIGURE 11.1 Revised Codification Pedagogy Workflow (Gick, 1986, p. 101)

1986). The purpose of layering was to help students model experts' behaviours during the problem-solving process. Figure 11.1 shows an adjusted version of the pedagogy's workflow. The solid blue problem-solving states were the focal states for the Codification Pedagogy, supporting the *Construct a Representation* and *Search for a Solution* states. The studies performed in this thesis demonstrated students using the pedagogy to support other states in the problem-solving process. Figure 11.1 shows the additional problem-solving states in grey, where students applied the pedagogical goals to solve the programming problems. Students used the pedagogy during the *Implement Solution* state when struggling to determine the next steps in the problem-solving process. During the *Implement Solution* state, students used the pedagogy to support their Self-Regulated Learning (SRL) and design strategies. Students also used (SRL) strategies during the *Stop* state, to validate their work upon completion.

By answering the research questions in Table 11.1, contributions were made to the field of Computer Science Education. Table 11.2 presents a brief summary of these contributions, which are discussed further in this section.

### 11.1.1 Assignment Design Framework

Chapter 5 described the research performed to collect assignment design treatments from peer-reviewed publications. These publications showed that these treatments helped students better understand CS1 programming assignments. This study collated the treatments into an assignment design framework, which was then applied to the programming assignments. Though the research did not investigate educators' use of the framework, it was designed to help guide them in constructing assignments that are appropriate to their students' abilities. Careful assignment design considerations appeared to influence students' software development process, supporting them when implementing and validating programming solutions.

Chapter 7 described a study using the assignment design framework, where interviews were conducted to get students' perspective on how the assignment presentation help in their understanding of the problem. This work contributes to the field of Computer Science Education by identifying CS1 students struggling when reading requirements in paragraph form. Results from the study showed students favouring the decomposition of programming goals and presenting them in list format. The decomposition of programming goals in list format helped students work through the subgoals, ensuring they were addressing all the requirements. Students also took advantage of the list format to validate their work, ensuring they successfully completed all the requirements. Identifying this presentation format can help future assignment design research, supporting students in transitioning towards reading software requirements specification documents.

| Contribution | Description |
|---|---|
| Assignment Design Framework | • Provides a CS1 assignment design presentation framework to assist in the scaffolding of assignment descriptions.<br>• Identifies that the decomposition of programming goals in list format helps students better identify the problem requirements and evaluate solving the problem at higher cognitive levels. |
| Instructional Question Framework | • Presents a framework to help educators construct questioning activities that will help students practise critical thinking skills appropriate to their cognitive levels.<br>• Demonstrates that the combination of lower and higher-order questions in a learning activity can help students generate answers that are appropriate to their cognitive level. |
| Design-based Parsons Problems | • Identifies three problem-solving approaches students take when organising plans in a design strategy activity. These approaches are: experimenting with the plan's order, ordering plans from the top down, and addressing easier plans first.<br>• Shows that the presence of a design-based Parsons problem as an intervention to a programming assignment can support students' use of SRL strategies to solve programming problems. These study finds the Parsons problems support the use of design and planning SRL strategies. |

TABLE 11.2 List of Research Contributions

### 11.1.2 Instructional Question Framework

Chapter 8 presented a study using instructional question types to encourage students to use critical thinking skills during the problem-solving process. To construct the questioning activity, the study identified 23 Instructional Question Types (IQTs) that engage students' cognitive levels. The study mapped the IQTs to Bloom's Taxonomy, resulting in a framework (See Table 8.2) that can guide educators in constructing learning activities. The study demonstrated a combination of low and high-order thinking questions as an intervention to a programming assignment can encourage low and high-order critical thinking skills, where the results showed the activity encouraged the desired cognitive levels (See Section 8.5). The activity can encourage students to self-reflect and apply their knowledge to the programming problem. The study showed the questioning activity helped students achieve cognitive levels appropriate for the course.

### 11.1.3 Design-Based Parsons Problems

Parsons problems were used to help students learn programming by arranging code fragments into a working program. This thesis is the first attempt to use Parsons problems during the design process. Chapter 9 presented a study that used Parsons

problems to support the practice of design knowledge by having students organise plans for solving programming problems. Results from the study showed Parsons problems supporting students' use of SRL strategies (See Table 9.6). Students used the tool to validate programming problems upon completion. The tool also helped struggling students to determine the next steps in the problem-solving process, such as helping them better understand how the plans work together to form a working a program (See Figure 9.6).

## 11.2 Threats to Validity

There are threats to validity for this research. One is the disparity between male and female students participating in the qualitative studies. Majority of the results come from male students, and the method of inviting participants through a class announcement in Canvas did not attract more female students. Another limitation is the small sample size in the qualitative studies, where the reported results came from four to six participants. However, the smaller sample size allowed for repeated and involved observational data collection that generated in-depth information on how the Codification Pedagogy helped them in their understanding and solving of procedural programming assignments. Another threat to validity is the change in the instructor during the studies that collected data for the control group. The instructor's teaching approach might have been different from the initial instructor, and could have influenced how the instructional material was presented. Another limitation is the Codification Pedagogy focused on procedural programming assignments and was not explored in other programming paradigms, such as functional paradigm. The pedagogy also focused on practical procedural programming environments and was not evaluated in other types of assessments, such as worked examples.

## 11.3 Future Work

This section presents future work for the Codification Pedagogy. Future work on individual learning activities was presented in the summary sections in the study chapters. This section discusses future research opportunities for the overall pedagogy.

The Codification Pedagogy was tested in the CS problem space. Future research opportunities can explore the use of the design process pedagogy benefiting Science, Technology, Engineering, and Mathematics (STEM) disciplines. Examining the learning activities in other disciplines can determine whether the activities provide similar support in promoting critical thinking and SRL strategies, or whether the activities promote the use of different SRL strategies. The guidance the Codification Pedagogy provides through the problem-solving process might also be beneficial to other STEM disciplines. The Codification Pedagogy allows for different activities to be contained and arranged, which other STEM disciplines could adapt for their own problem-solving process. Evaluating the pedagogy in other disciplines might also demonstrate the activities might be better suited for the CS problem space. For example, the *Design Strategy Activity* built on top of the Parsons problems tool might be relevant to the CS design process, but possible not applicable to other disciplines. Evaluating the learning activities in other disciplines could suggest replacing the Parsons problems with different design-based activities.

For this thesis, the Codification Pedagogy was evaluated within a Massive Open Online Course (MOOC). Future evaluation of the pedagogy in a classroom learning environment could determine whether the pedagogy can support aspects of collaborative design outside of the online learning environment. Applying in a collaborative environment might demonstrate how the pedagogy can support Co-Regulated Learning (CoRL) or Socially Shared Regulation of Learning (SSRL). Co-Regulated Learning is where 'group members take control of their own think, behavior, motivation, and emotion in the collaborative task' (Järvelä et al., 2016), while Socially Shared Regulation of Learning is where 'group members work together to regulate their collective cognition, behavior, motivation and emotions together in a synchronized and productive manner' (Järvelä et al., 2016).

As shown in Figure 11.1, the studies presented in this thesis demonstrate students using the learning activities to support them in the problem-solving process surrounding the planning and organising a solution. The results from this thesis showed students using the pedagogy throughout the software development process. Supporting other phases, such as debugging, in software development processes outside the design process was not the expectation for this thesis. Observing students using the pedagogy throughout the software development process was an unexpected outcome, and provides future research opportunities. The future research could help identify the learning activities that better support other software development processes, such as debugging.

## 11.4 Concluding Remarks

This research began with the intention to help CS1 students integrate the design process earlier in their learning. The methods to support the adoption of the design process formed a pedagogy designed to help CS1 students to learn and practise design-based skills for procedural programming assignments. Contained within the pedagogy are learning activities layered on top of a problem-solving process model (Gick, 1986). This instructional method is designed to help students engage certain skills and strategies through the different problem-solving states. By designing the pedagogy using a series of learning activities, the thesis demonstrates to the Computer Science Education community that individual learning activities can be combined with an agenda to present a broader learning objective. In the case of the Codification Pedagogy, the broader learning objective is helping CS1 students adopt Good Learning Behaviours to be used during the design process.

The thesis examines the pedagogy with six studies, evaluating the pedagogy's activities in detail. These studies are a combination of quantitative and mixed-methods qualitative studies. The thesis used a mix of qualitative and quantitative study methods that views the pedagogy through different lenses. The research spanned three semesters involving large sample sizes in the quantitative data to reduce errors in testing and identify behavioural trends, such as the pedagogy's influence with the completion rate of practical programming assignments. The qualitative studies used smaller sample sizes, allowing for repeated and involved observational data collection that generated in-depth information of the students' thinking processes while interacting with the pedagogy. Some of the study methods demonstrate how to get students more involved in the research process, giving students the opportunity to identify to educators areas in the learning process that are a challenge. The study methods also demonstrate how to collect students' cognitive

processes and experiences using the pedagogy as another approach to identify areas that are a challenge for students. With the students' insights, educators can re-examine the challenging areas to help improve the students' learning processes and experiences. The study methods used in this thesis can serve as a guide for evaluating future CS pedagogies, where the depth and variety of data collection and analysis provided a comprehensive evaluation of the pedagogy. The study methods also included a variety of study method approaches to the pedagogy's learning activities. By evaluating individual activities, this thesis demonstrates how each activity contributes to the students' learning at the different problem-solving states.

This thesis investigates a pedagogy integrated into procedural CS1 assignments that promotes Good Learning Behaviours (GLBs) and discourages Poor Learning Tendencies. The pedagogy is designed using theories in program comprehension, critical thinking skills, and design knowledge and promotes Good Learning Behaviours by demonstrating the use of self-reflection and Self-Regulated Learning (SRL) strategies, to better understand the problem and to organise a plan prior to software implementation. The thesis demonstrates the pedagogy helping students better understand programming assignments, helping struggling students progress through the problem-solving process, and supporting their use of SRL strategies. This research provides suggestions for future introductory procedural programming courses. Constructing the Codification Pedagogy with individual learning activities invites CS educators to try different instructional approaches best suited for their students' needs. Educators have the opportunity to evaluate the presentation order of the learning activities. For example, presenting the *Design Strategy Activity* before the *Questioning Activity* might encourage more self-reflection, possibly elevating students' cognitive levels when they answer the activity. Re-arranging the activities could result in different SRL strategies, or improved learning gains. Also, because the pedagogy was designed with individual activities that serve a specific purpose in the students' learning, substituting other activities within sections of the pedagogy could provide a framework for pedagogical experimental design.

# Appendix A

# Programming Assignments

## A.1 Pilot Group Codification Assignments

### A.1.1 Assignment 4

**Question 1**

**Assignment 4 Question 1: Products Sold (8 marks)**

In your programming assignment, you get to **practise your coding skills** using:

- **Variables**
- **Integers**
- **Loops**
- **If Then/Else**
- **Arrays**

**Program Description**

For this assessment, you will create a **program that averages products over a given amount of days**. Your program initially reads in an array, where an element in the array represent the number of products sold for one day. Your program processes each day from the array until your program encounters the value -789 in the array. The number -789 signals your program to stop reading products sold from the array, and proceed to calculate the average (arithmetic mean) of the products sold. Once the average is calculated, display the results on the canvas as:

*In 'n' days, the average products sold is 'u' units.*

The variable *'n'* represents the number of days you program reads in from the array. The variable *'u'* represents the average units sold over those days.

**Hints**

- Negative units is not a valid value for your program. If there is a negative number of units sold, your program should instead replace that negative value to 0 (zero).
- Here is an example list of products sold in 5 days: (15,0,-53,5,-789,2), with 15 units, 0 units, 0 units, and 5 units. Please note that -53 is an invalid number of units sold, so your program converts the 3rd day to 0 units.

- The formula for calculating the average is:
  Average = (x1 + x2 + ... + xn)/n
- For more information on averaging (arithmetic mean), go to the website:
  `http://mathsisfun.com/data/mean-machine.html`

**Assignment 4 Question 1: Planning Help**

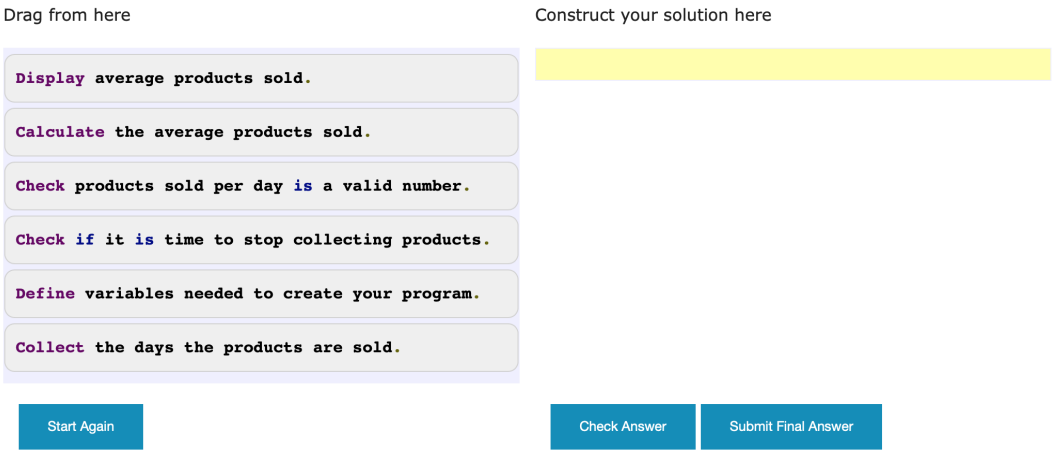
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What do you already know about collecting and organising data for products sold that might help you complete this assignment?
2. Breaking down the problem into smaller tasks will help you complete the assignment. How would you go about breaking down this problem into smaller tasks?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



**Question 2**

**Assignment 4 Question 2: Football League List (7 marks)**

In your assignment, you get to **practise your coding skills** using:

- **Variables**
- **Integers**
- **Loops**
- **If Then/Else**
- **Arrays**

**Program Description**

Your program will create a table that displays a list of football teams and their corresponding statistics, such as the one below:

| Team Name | Statistics |
|---|---|
| AFC Bournemouth | 2 |
| Arsenal | 5 |
| Brighton and Hove Albion | 3 |
| Burnley | 3 |
| Chelsea | 5 |
| Crystal Palace | 1 |
| Everton | 2 |
| Huddersfield Town | 3 |
| Leicester City | 2 |
| Liverpool | 3 |
| Manchester City | 8 |
| Manchester United | 6 |
| Newcastle United | 4 |
| Southampton | 3 |
| Stoke City | 2 |
| Swansea City | 2 |
| Tottenham Hotspur | 6 |
| Watford | 4 |
| West Bromwich Albion | 2 |
| West Ham United | 2 |

**Hints:**

- Your football list needs to be created with the following of names in your program:

```
String[] teamNames = { "AFC Bournemouth", "Arsenal",
                       "Brighton and Hove Albion",
                       "Burnley", "Chelsea",
                       "Crystal Palace", "Everton",
                       "Huddersfield Town",
                       "Leicester City", "Liverpool",
                       "Manchester City",
                       "Manchester United",
```

```
"Newcastle United",
"Southampton", "Stoke City",
"Swansea City",
"Tottenham Hotspur",
"Watford",
"West Bromwich Albion",
"West Ham United" };
```

- Create a second array to hold teams' statistics. Statistics should be assigned randomly, but with biased towards a mean stat of 5 with a standard deviation of 3. View the RandomGaussian tutorial on Processing.org to see how you can achieve this.
- The **Team Name** column should be left-aligned.
- The **Statistics** column should be center-aligned. The rows of the table should be filled based on the following:
  - Bottom (0-1) - yellow
  - Low (2-3) - cyan
  - Middle (4-5) - green
  - High (6-7) - purple
  - Lead (8-10) - red

**Assignment 4 Question 2: Planning Help**

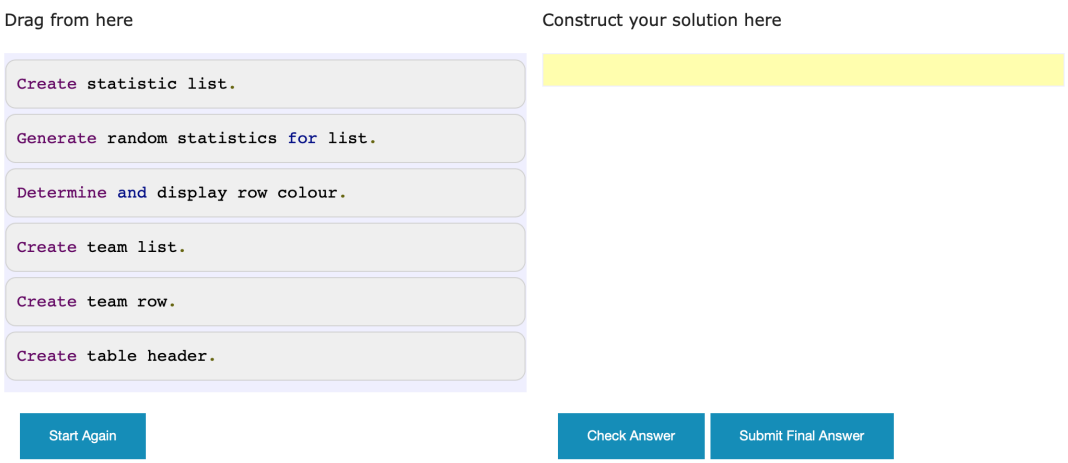
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. You might not be familiar with football, but you might already know about sports and competition. How can your current knowledge about sports and competition help you better understand this problem?
2. A part of this assignment is to colour code a team's row based on their statistics. What do you think is a good approach to assigning the colour?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.


Drag from here
Construct your solution here
Create statistic list.
Generate random statistics for list.
Determine and display row colour.
Create team list.
Create team row.
Create table header.
Start Again
Check Answer
Submit Final Answer

### A.1.2 Assignment 5

**Question 1**

**Assignment 5 Question 1: Hue Function (6 marks)**

In your assignment, you get to **reuse programming components** from previous programs, like for loops and functions.

**Program Description**

Below is code that draws a number of lines to form grayscale hue. The hue starts at the dark 0 (black) and lightens until it reaches **haltValue**. You should copy this code into the Processing IDE to see how it works. Try passing in different values to the **displayHue** function to see how the output changes.

```
void setup(){
 size(400,400);
 background(255);
 noLoop(); //This prevents draw() from running
           //more than once.
}

void draw(){
 //Call our function, draw a gradient from white to black
 displayHue(255);
}
// displayHue - Displays a grey hue scale, the scale
// starts at 0 (black) and lightens until it reaches
// haltValue.
void displayHue(int haltValue){
 for(int i = 0; i <= haltValue; i++){
   stroke(i);
   line(10,i,60,i);
 }
}
```

For your assignment, you will need to copy the above code and extend the **displayHue** function to have the following syntax:

```
  void displayHue(float x, float y, boolean o,
int hv, int s, int c, float d)
```

Where the input parameters will change the output of **displayHue** in accordance with the following table:

| Parameter | Description Possible Values | Possible Values |
|---|---|---|
| x | X-coordinate of the top left corner of the gradient. | |
| y | Y-coordinate of the top left corner of the gradient. | |
| o | Determines if the gradient is drawn vertically or horizontally. | True - if the gradient is drawn vertically (darkest at top) False - if the gradient is drawn horizontally (darkest at left-side) |
| hv | The halt value of the gradient (gradient will start at colour 0 and go to hv). | 0 to 255 |
| s | The opacity of the gradient. | 0 to 255 |
| c | Determines the colour of the hue. | 0 - grayscale 1 - yellow 2 - cyan 3 - purple |
| d | The width or height of the gradient (width if drawn vertically, height if drawn horizontally). | |

If any of the input parameters are invalid, your **displayHue** function should display a warning, and not draw the gradient. See above for the valid values. Use a **for loop** to call **displayHue** several times to design a pattern. The pattern should be produced by modifying the hue attributes - halt value, width/height, opacity, orientation, or colour. To achieve full marks for this part please ensure you create a pattern with at least 5 hues, and alter at least 2 of the hues attributes in your pattern.

**Assignment 5 Question 1: Planning Help**

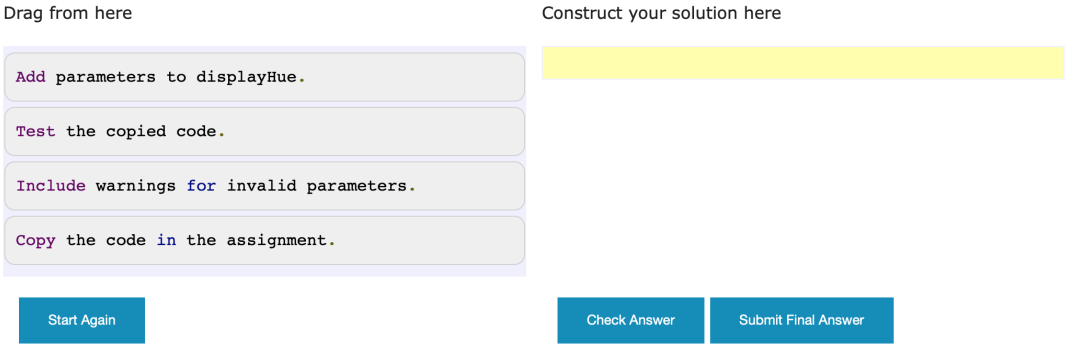
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What ideas do you have for the patterns your program will need to generate?
2. How do you think you will need to alter **displayHue** to create these patterns?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.


Drag from here
Add parameters to displayHue.
Test the copied code.
Include warnings for invalid parameters.
Copy the code in the assignment.
Start Again
Construct your solution here
Check Answer
Submit Final Answer

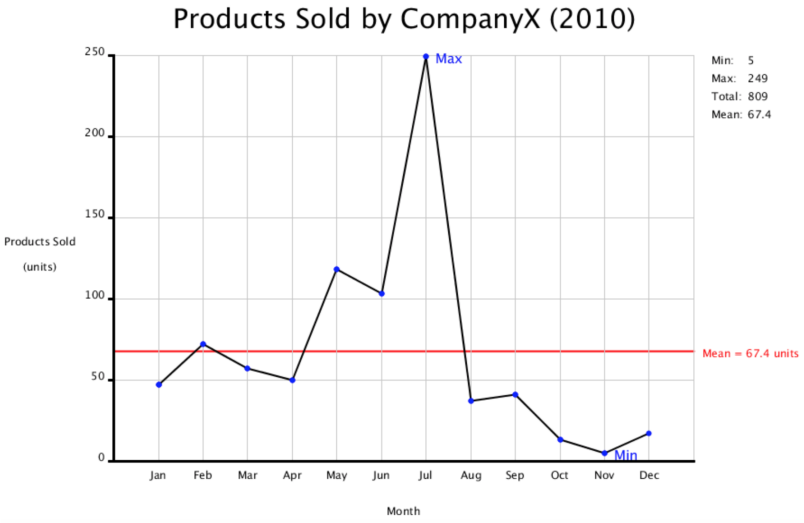
**Question 2**

**Assignment 5 Question 2: Products Sold Graph (14 marks)**

There are **two parts** to this assignment. This is the second part.

**Program Description**

Create a program that produces a product sold graph like the following:




You should build upon the following skeleton code to produce the product sold graph.

```
void setup(){
  size(900,600);
  background(255);
  noLoop(); // Only call the draw() function once
}

void draw(){
  // Array of products sold per month for CompanyX in 2010.
  int[] companyX2010 = {47, 72, 57, 50, 118, 103, 249,
                        37, 41, 13, 5, 17};

 // Draw the products sold graph for CompanyX, 2010.
  drawProductGraph(companyX2010,"CompanyX",2010);
}

// drawProductGraph - draws a product sold graph for a
// particular location in a particular year
// product:  each element in the product array represents
//           the total products sold for the
//           respective month
// location:  the location of the product sold
// year: the year of the products were sold
void drawProductGraph(int[] product,
```

```
                        String location, int year){}

// minimum - returns the minimum value in an int array
int minimum(int[] array){}

// maximum - returns the maximum value in an int array
int maximum(int[] array){}

// total - returns the total of a int array
int total(int[] array){}

// mean - returns the mean of a int array
float mean(int[] array){}
```

The specific requirements for the graph are as follows:

- Title: "Products Sold by company (year)"
- X-Axis:
  - Label: "Month"
  - Scale: One entry for every month of the year, Jan through to Dec.
  - Gridlines: Light grey vertical gridlines for each month.
- Y-Axis:
  - Label: "Products Sold (units)"
  - Scale: Should start at 0 and go to just above the maximum product sold, choose an appropriate scale interval so that it is clear and easy to read. The scale interval should change depending on the maximum products sold (if max is 250, an interval of 50 would be appropriate, if max is 45, an interval of 10 would be appropriate).
  - Gridlines: Light grey horizontal gridlines for each scale interval.
- Values
  - A blue dot should be positioned on the graph to represent the products sold for each month.
  - Connect the blue dots with black lines.
  - Display the text "Min" next to the minimum product sold.
  - Display the text "Max" next to the maximum product sold.
  - Display the mean value as a red line, with "Mean = value" next to the line.
- Statistical data
  - Displayed to the right of the graph.
  - Display the min, max, total and mean of the year's sold products.

Your **drawProductGraph** function should also work with different product sold data, test your code with the following data:

```
// Test data - 2
int[] companyW1995 = {72, 14, 16, 24, 28, 32, 110,
                      55, 198, 362, 579, 1674};
```

```
// Test data - 3
int[] companyY2005 = {3, 25, 42, 85, 81, 119, 183,
                      70, 96, 2, 8, 18};

// Test data - 4
int[] companyZ2016 = {53, 44, 51, 18, 40, 39, 56,
                      100, 67, 24, 6, 48};
```

**Assignment 5 Question 2: Planning Help**

*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

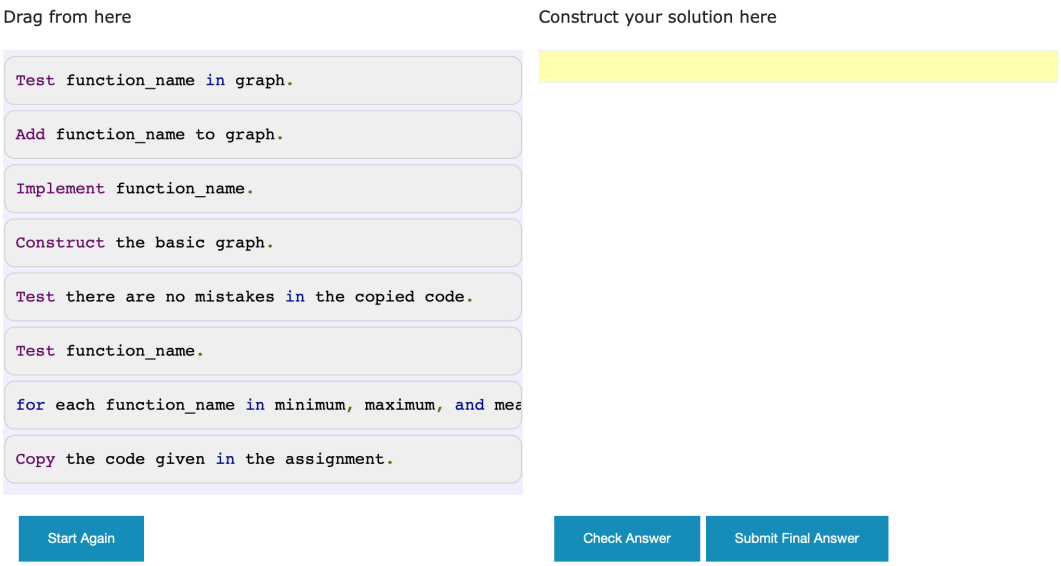
Student ID: (Required) _____

1. Review the graph in the assignment. How does this graph help you understand the problem?
2. There are a few separate programming parts to this assignment. Describe your plan on how you will program these parts.

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.

These planning tasks describe higher level task descriptions typical of how larger programs are planned. This approach enables you to have a broad view of what needs to get solved without being overwhelmed with coding details. Some sections are repeatable tasks for different functions, represented by a loop over these functions written in pseudocode.



### A.1.3 Assignment 6

**Question 1**

**Assignment 6 Question 1: Coin in the Well (7 marks)**

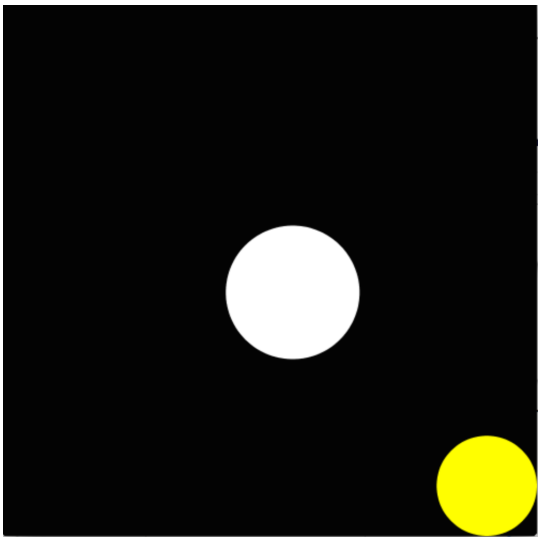
There are **two parts** to this assignment. This is the first of the two parts.

**Program Description**

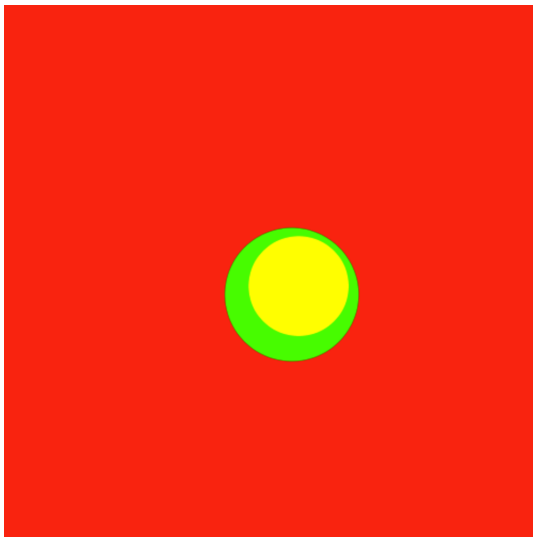
Create a program that allows the user to move a **yellow coin** into a **wishing well** using the arrow keys on the keyboard. Your program should meet the following specifications:

- The coin starts in the lower right hand corner.
- The wishing well should be at a random location, it cannot overlap the coin's beginning position.
- The wishing well's diameter must be scalable with the the sketch size.
- The coin's diameter should be 75% of the hole's diameter.
- The coin changes position when the arrow keys are pressed.
- The coin cannot go off the edge of the sketch.
- When the coin is completely or partially **outside** the wishing well the background should be black and the wishing well should be white.
- When the coin is completely **inside** the wishing well the background should be changed to red and the wishing well becomes green.
- Include a function named **isWishMade()** that returns true if the coin is completely within the wishing well, false otherwise.

Start of the program:



When coin is completely inside the wishing well:



**Assignment 6 Question 1: Planning Help**

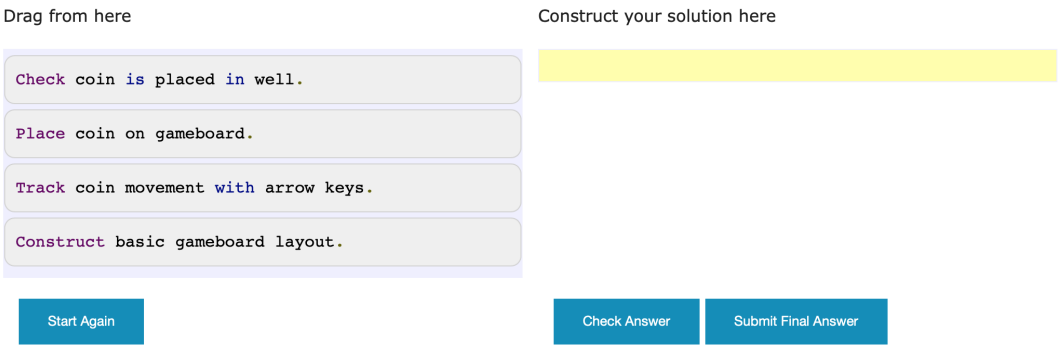
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What have you already learned to help you move the coin around on the canvas?
2. You might need additional information to help you solve the question. What are your plans for seeking out additional information?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



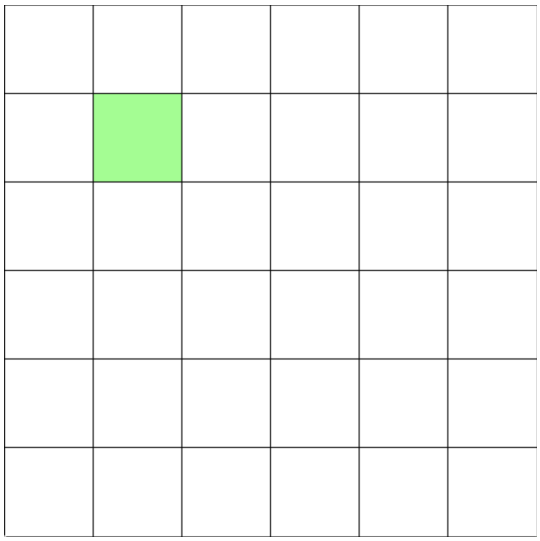
**Question 2**

**Assignment 6 Question 2: Interactive Grid (13 marks)**

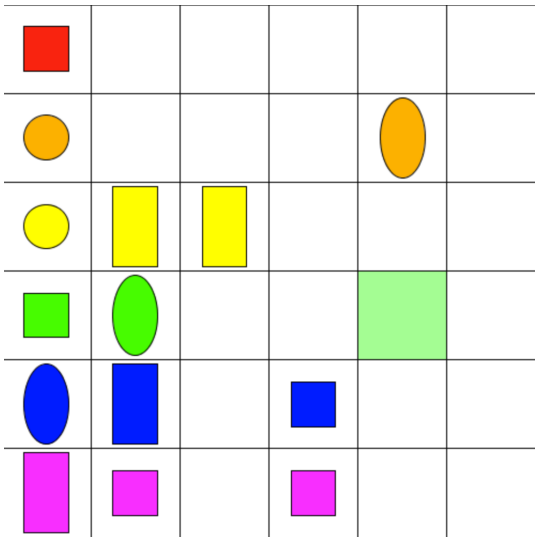
Create a program that allows the user to place a random shape in each square of a grid. Once all of the squares in the grid have a shape placed within them the grid can be reset by the user. Your program should meet the following specifications:

- The grid should have 6 rows and 6 columns.
- The grid should scale with the width and height of the sketch (the sketch should always be square - equal width and height).
- The grid should be made up of white squares with black outlines.
- When the mouse is over a square that does not have a shape placed within it, it should highlight that square in green.
- When the mouse is clicked on a square that does not have a shape placed within it, it should place a random shape within it. The available shapes to choose from are a circle, oval, rectangle or square.
- The fill colour of each shape will be determined by the row that they are being placed in:
  - Row 1 – Red
  - Row 2 – Orange
  - Row 3 – Yellow
  - Row 4 – Green
  - Row 5 – Blue
  - Row 6 – Purple
- Once a square has had a shape placed within it, it cannot be highlighted or have another shape placed within it until the entire grid is reset.
- Once the entire grid is filled with shapes, the next mouse click will reset the grid back to white, allowing the user to fill the grid over again.
- When resetting the grid back to white you must reset the squares one by one. For example, you could first reset square (0,0) back to white, and then square (0,1) back to white, and then square (0,2) back to white, and so on. You can choose the order in which the squares are reset.
- Use an appropriate frame rate, such that the resetting animation is easily visible.

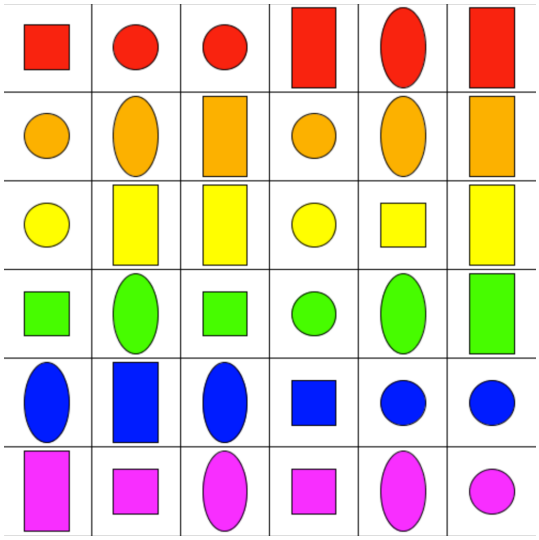
Beginning state of the program:



Intermediate stage:



Completed stage:



**Assignment 6 Question 2: Planning Help**

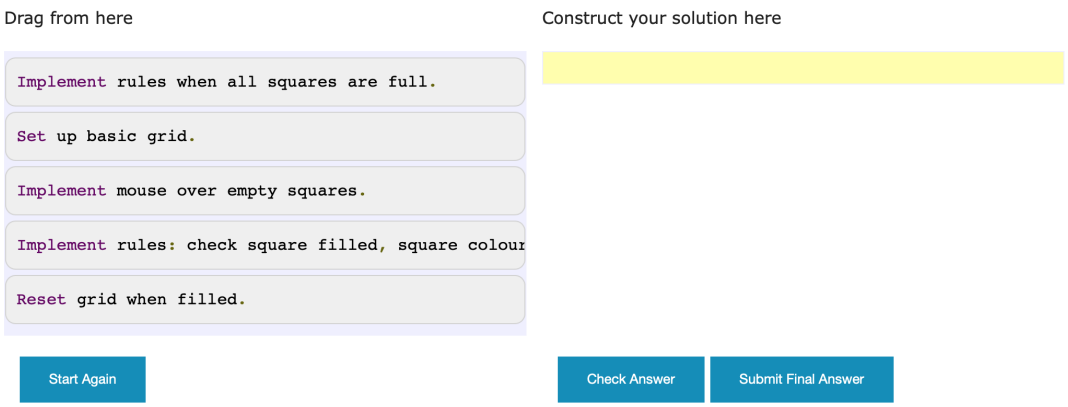
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



### A.1.4 Assignment 7

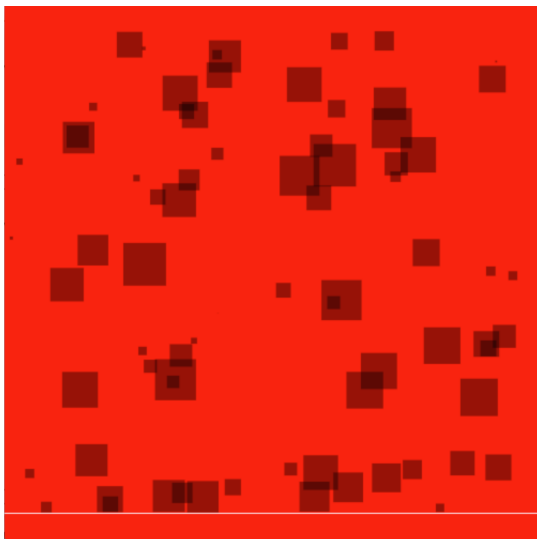
**Question 1**

**Assignment 7 Question 1: Dragging Squares (8 marks)**

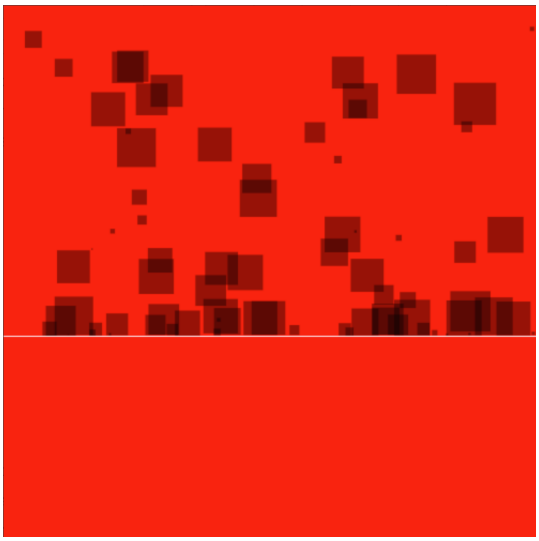
**Program Description**
Create a program that animates a number of randomly positioned squares being dragged down the screen. Your program should meet the following specifications:

- The canvas should have a width and height of 500, and have a red background.
- Your program should have a frame rate of 30.
- There should be 75 squares displayed on the screen with the following properties:
  - x and y coordinates should be random.
  - The side lengths should be random between 1 and 40.
  - The fill should be black, with an opacity of 100.
  - The entire square must be on the canvas initially.
- A white horizontal line will sweep up the canvas from the bottom to the top of the canvas.
- As the horizontal line sweeps up it should drag any square that it touches with it.
- Once the horizontal line has reached the top of the canvas the animation should reset with new random squares, moving from the top to the bottom of the screen.

The images below show the animation in action.
Moving up:



Moving up:



Moving down:

**Assignment 7 Question 1: Planning Help**

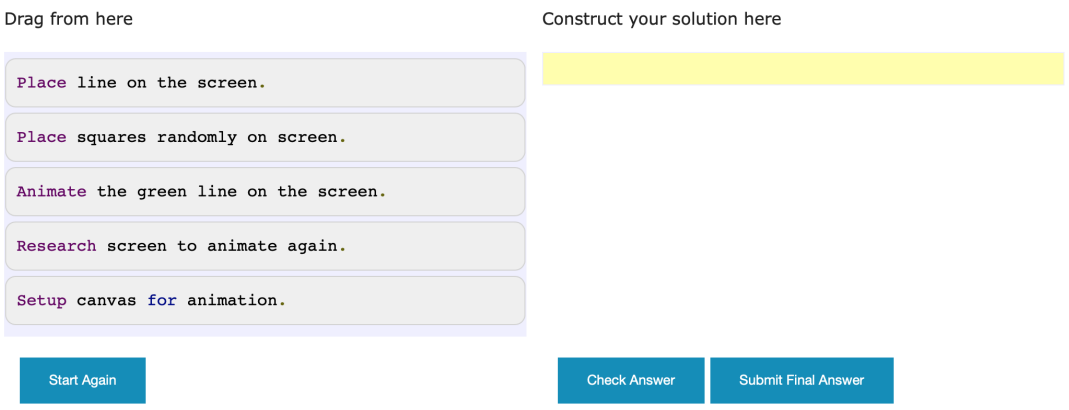
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



**Question 2**

**Assignment 7 Question 2: Rotation Function (8 marks)**

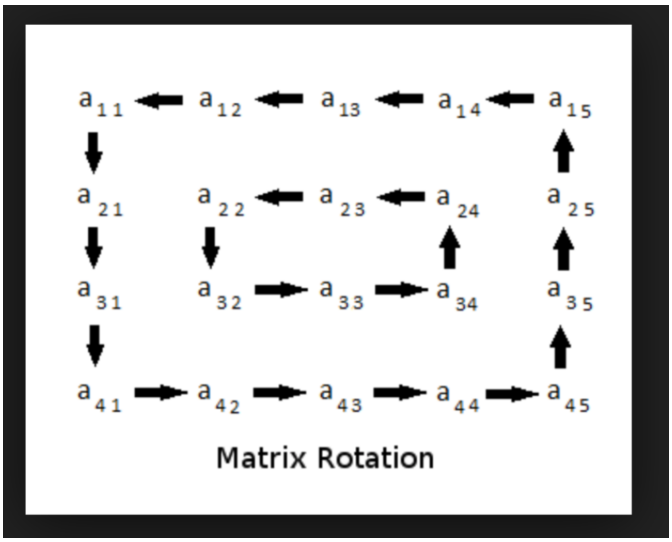
**Program Description**

Create a function name **rotateArray** that takes a 2D integer array as a parameter and rotates the array's entries in a anti-clockwise fashion by one position like the following image.

This algorithm needs to work for a 4x5 matrix dimension. Your program should meet the following specifications:

- Your function should print an error message to the console and not perform the rotation if the array passed in is empty, or if it does not adhere to the 4x5 matrix dimension.
- Print the array that is being rotated, both before and after the **rotateArray** function is called. This information should be printed to the console.
- You need to test your **rotateArray** function with at least 4 different matrices, including edge cases. Your test matrices need to be placed in your program for to receive credit.
- Your solution must use loops to rotate the elements of the 2D array.

**Hints:**




- For the above diagram, you need to be familiar with the mathematical use of subscripts to represent array elements. If not, visit:

  http://northstar-www.dartmouth.edu/doc/idl/html_6.2/Understanding_Array_Subscripts.html

- Example Input:

```
[ [ 1,  2,  3,  4,  5],
  [ 6,  7,  8,  9, 10],
  [11, 12, 13, 14, 15],
  [16, 17, 18, 19, 20] ]
```

Output from rotation:

```
[ [ 2,  3,  4,  5, 10]
  [ 1,  8,  9, 14, 15]
  [ 6,  7, 12, 13, 20]
  [11, 16, 17, 18, 19] ]
```

**Assignment 7 Question 2: Planning Help**

*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s). When you are done with your answers, please click the "Submit Final Answer" button.

Student ID: (Required) _____

1. How are you going to try to better understand this problem?
2. What are your plans for trying to solve this problem?

## A.2 Experiment 2 Group Codification Assignments

### A.2.1 Assignment 4

**Question 1**

**Assignment 4 Question 1: Averaging Rainfall (8 marks)**

In your programming assignment, you get to **practise your coding skills** using:

- **Variables**
- **Integers**
- **Loops**
- **If Then/Else**
- **Arrays**

**Program Description**

**Create a program that averages rainfall for a certain number of days.** Your program first receives the rainfall from an array. The days are processed one day at a time from the array until an array entry is number -99999. The -99999 is a signal to stop accepting rainfall information. Then your program should calculate the Arithmetic Mean (Average) of the rainfall. Lastly, it should display the result on the screen in the following format:

*The average rainfall for 'n' days is 'x' mm.*

The variable *'n'* represents the number of days you program reads in from the array. The variable *'x'* is the average rainfall over those days.

**Hints**

- Do not accept negative rainfall, since this does not exist in the real world. If a negative rainfall is provided, convert that day's rainfall to 0.
- Here is an example rainfall list for 5 days:
  (15,0,-53,5,2,-99999), with rainfall 15 mm, 0 mm, 0 mm, 5 mm, and 2 mm. Please note that -53 is an invalid rainfall value, so your program converts the 3rd day to 0 mm.
- The formula for calculating the average is:
  Average = (x1 + x2 + ... + xn)/n
- For more information on averaging (arithmetic mean), go to the website:
  `http://mathsisfun.com/data/mean-machine.html`

**Assignment 4 Question 1: Planning Help**

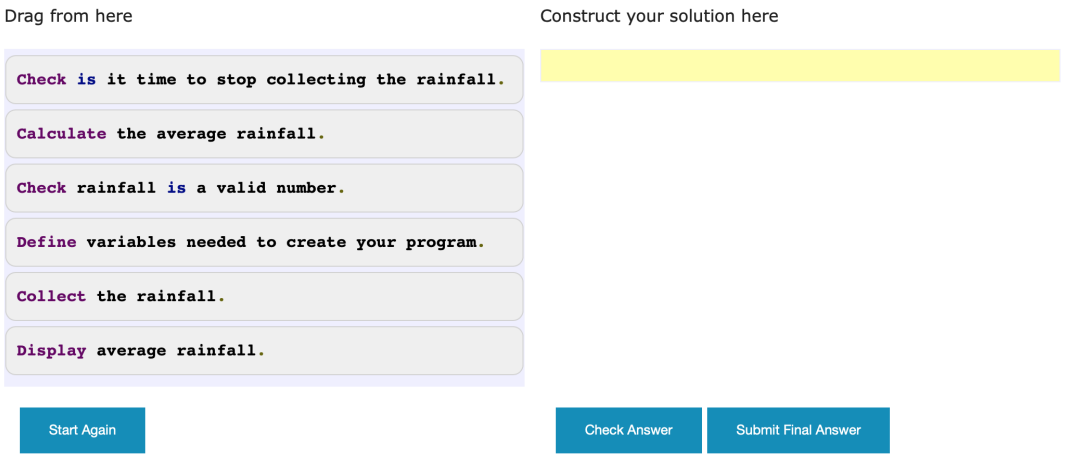
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What do you already know about collecting and organising data for averaging rainfall that might help you complete this assignment?
2. Breaking down the problem into smaller tasks will help you complete the assignment. How would you go about breaking down this problem into smaller tasks?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



**Question 2**

**Assignment 4 Question 2: Class Grade List (7 marks)**

In your assignment, you get to **practise your coding skills** using:

- **Variables**
- **Integers**
- **Loops**
- **If Then/Else**
- **Arrays**

**Program Description**

Your program will create a table that displays a list of students and their corresponding grades, such as the one below:

| Student Name | Grade |
|---|---|
| Beaubier, Jean-Paul | 76 |
| Bishop, Lucas | 68 |
| Cassidy, Sean | 38 |
| Drake, Robert | 34 |
| Frost, Emma | 94 |
| Grey, Jean | 58 |
| Gutherie, Paige | 55 |
| Howlett, James | 54 |
| LeBeau, Remy | 61 |
| Lee, Jubliation | 87 |
| Marie, Anna | 88 |
| McCoy, Hank | 51 |
| Monroe, Ororo | 51 |
| Moonstar, Danielle | 99 |
| Pryde, Kitty | 81 |
| Rasputin, Piotr | 21 |
| Summers, Scott | 84 |
| Wagner, Kurt | 60 |
| Worthington, Warren | 73 |
| Xavier, Charles | 71 |

**Hints:**

- Your class grade list needs to be created with the following of names in your program:
  Use the following array of names in your program:

```
String[] studentNames =
    {"Beaubier, Jean-Paul", "Bishop, Lucas",
     "Cassidy, Sean", "Drake, Robert",
     "Frost, Emma", "Grey, Jean",
     "Guthrie, Paige", "Howlett, James",
```

```
"LeBeau, Remy", "Lee, Jubilation",
"Marie, Anna", "McCoy, Hank",
"Monroe, Ororo", "Moonstar, Danielle",
"Pryde, Kitty", "Rasputin, Piotr",
"Summers, Scott", "Wagner, Kurt",
"Worthington, Warren", "Xavier, Charles"};
```

- Create a second array to hold the students' grades.
- The **Student Name** column should be left-aligned.
- The **Grade** column should be center-aligned.
- The rows of the table should be filled based on the following:
  - Fail (0-49) - red
  - Pass (50-64) - purple
  - Credit (65-74) - cyan
  - Distinction (75-84) - green
  - High Distinction (85-100) - yellow
- Grades should be assigned randomly, but biased towards a mean grade of 60, with standard deviation of 20.
- View the RandomGuassian tutorial on Processing.org to see how you achieve random generation.

**Assignment 4 Question 2: Planning Help**

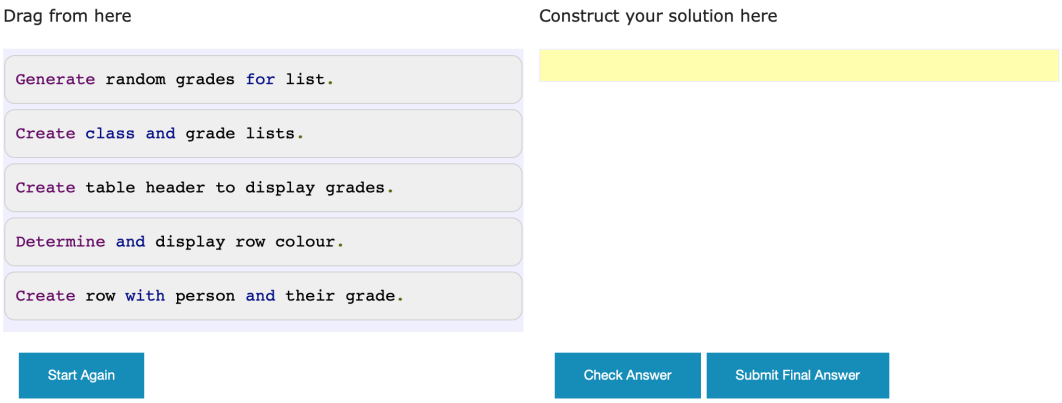
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. How can your current knowledge about classroom and grading help you better understand this problem?
2. A part of this assignment is to colour the row containing a student's information based on their grade. What do you think is a good approach to assigning the colour?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.


Drag from here
Construct your solution here
Generate random grades for list.
Create class and grade lists.
Create table header to display grades.
Determine and display row colour.
Create row with person and their grade.
Start Again
Check Answer
Submit Final Answer

### A.2.2 Assignment 5

**Question 1**

**Assignment 5 Question 1: Gradient Function (6 marks)**

In your assignment, you get to reuse programming components from previous programs, like for loops and functions.

**Program Description**

Below is code that draws a number of lines to form a grayscale gradient. The gradient starts at 0 (black) and lightens until it reaches **endValue**. You should copy this code into the Processing IDE to see how it works. Try passing in different values to the **drawGradient** function to see how the output changes.

```
void setup(){
  size(400,400);
  background(255);
  //Call our function, draw a gradient
  //from white to black
  drawGradient(255);
}

// drawGradient - Draws a grayscale gradient,
// the gradient starts at 0 (black) and lightens
// until it reaches endValue.
void drawGradient(int endValue){
  for(int i = 0; i <= endValue; i++){
    stroke(i);
    line(10,i,60,i);
  }
}
```

For your assignment, you will need to copy the above code and extend the **drawGradient** function to have the following parameters:

| Parameter | Description Possible Values | Possible Values |
|---|---|---|
| x | X-coordinate of the top left corner of the gradient. | |
| y | Y-coordinate of the top left corner of the gradient. | |
| vertical | Determines if the gradient is drawn vertically or horizontally. | True - if the gradient is drawn vertically (darkest at top) False - if the gradient is drawn horizontally (darkest at left-side) |
| endValue | The end value of the gradient . (gradient will start at colour 0 and go to endValue) | 0 to 255 |
| opacity | The opacity of the gradient. | 0 to 255 |
| col | Determines the colour of the gradient. | 0 - grayscale 1 - red 2 - green 3 - blue |
| widHei | The width or height of the gradient (width if drawn vertically, height if drawn horizontally). | |

If any of the input parameters are invalid, your **drawGradient** function should display a warning, and not draw the gradient. See above for the valid values. Use a **for loop** to call **drawGradient** several times, to design a pattern. The pattern should be produced by modifying the gradient attributes: end value, width/height, opacity, orientation (vertical/horizontal), or colour. To achieve full marks for this part, please ensure you create a pattern with at least 5 gradients, and alter at least 2 of the gradient attributes in your pattern.

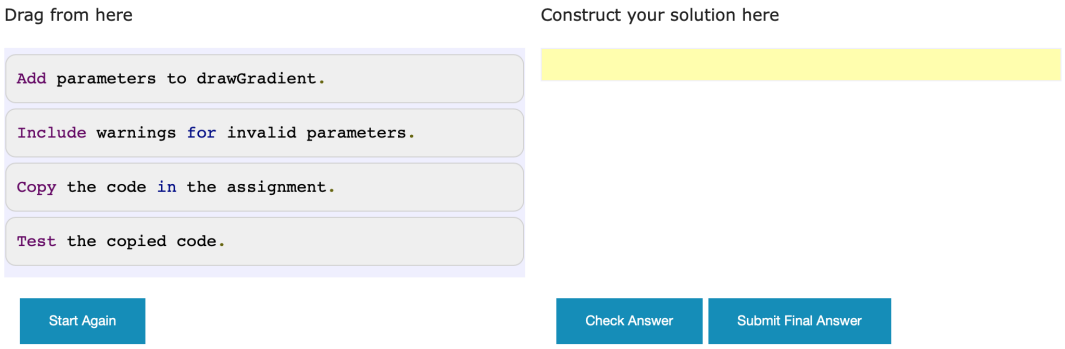
**Assignment 5 Question 1: Planning Help**

*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).
Student ID: (Required) _____

1. What ideas do you have for the patterns your program will need to generate?
2. How do you think you will need to alter drawGradient to create these patterns?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.


Drag from here
Add parameters to drawGradient.
Include warnings for invalid parameters.
Copy the code in the assignment.
Test the copied code.
Construct your solution here
Start Again
Check Answer
Submit Final Answer

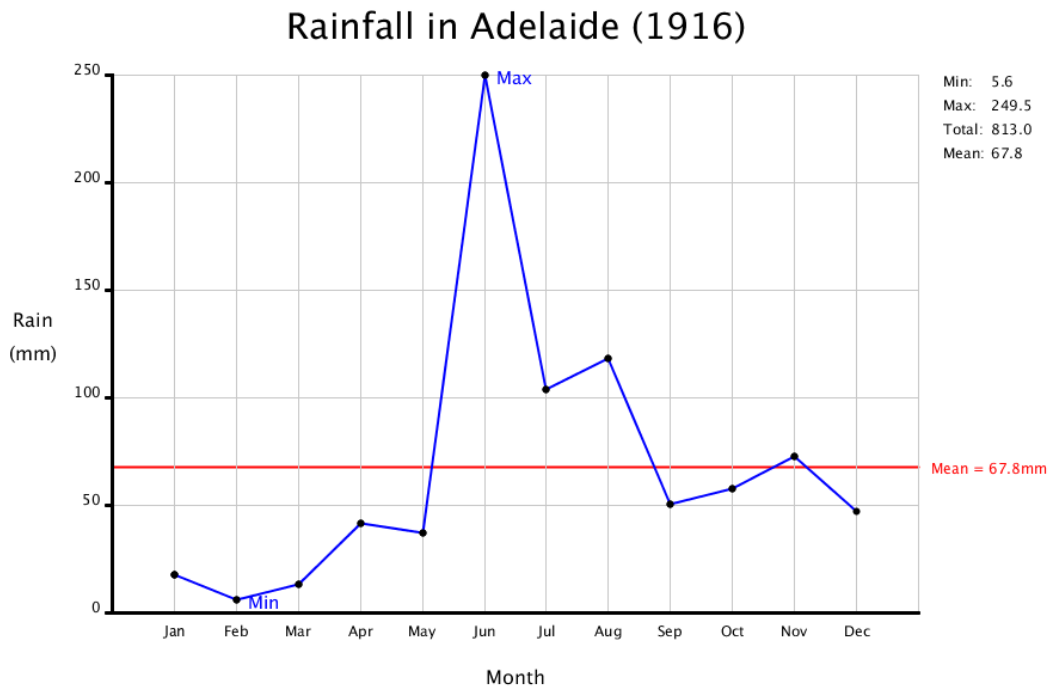
**Question 2**

**Assignment 5 Question 2: Rainfall Graph (14 marks)**

There are **two parts** to this assignment. This is the second part.

**Program Description**

Create a program that produces a rainfall graph like the following:




You should build upon the following skeleton code to produce the rainfall graph.

```
void setup(){
  size(900,600);
  background(255);
  // Array of rainfall per month for Adelaide in 1916.
  float[] adelaide1916 = {17.3,5.6,13.2,41.4,37.1,
                          249.5,103.4,118.1,50.1,
                          57.7,72.4,47.2};
  // Draw the rain graph for Adelaide, 1916.
  drawRainGraph(adelaide1916,"Adelaide",1916);
}

/*
drawRainGraph
Draws a rain graph for a particular location and year
rainfall: each element in the rainfall array
          represents the total rainfall for
          the respective month
```

```
location: the location of the rainfall
year: the year of the rainfall
*/
void drawRainGraph(float[] rainfall,
                   String location, int year){}

// minimum - returns the minimum value in a float array
float minimum(float[] array){}

// maximum - returns the maximum value in a float array
float maximum(float[] array){}

// total - returns the total of a float array
float total(float[] array){}

// mean - returns the mean of a float array
float mean(float[] array){}
```

**Please note** that you must implement the minimum, maximum, total and mean functions yourself, you should not be be making calls to any in-built Processing functions to assist with these functions. The specific requirements for the graph are as follows:

- Title: "Rainfall in location (year)"
- X-Axis:
  - Label: "Month"
  - Scale: One entry for every month of the year, Jan through to Dec.
  - Gridlines: Light grey vertical gridlines for each month.
- Y-Axis:
  - Label: "Rain (mm)"
  - Scale: Should start at 0 and go to just above the maximum rainfall, choose an appropriate scale interval so that it is clear and easy to read. The scale interval should change depending on the maximum rainfall (if max is 250, an interval of 50 would be appropriate, if max is 45, an interval of 10 would be appropriate).
  - Gridlines: Light grey horizontal gridlines for each scale interval.
- Values
  - A black dot should be positioned on the graph to represent the rainfall for each month.
  - Connect the black dots with blue lines.
  - Display the text "Min" next to the minimum rainfall.
  - Display the text "Max" next to the maximum rainfall.
  - Display the mean value as a red line, with "Mean = value" next to the line.
- Statistical data
  1. Displayed to the right of the graph.
  2. Display the min, max, total and mean of the year's rainfall.

Your **drawRainGraph** function should also work with different rainfall data, test your code with the following data:

```
// Test data - 2
float[] cairns1979 = {1674.4,579.4,362.4,198.4,
                      55.9,110.5,32.4,28.8,24.4,
                      16.7,14,72};

// Test data - 3
float[] perth2002 = {18.4,8.2,2.6,96,70.4,183.8,
                     119.8,81.2,85.6,42.2,25.8,3.8};

// Test data - 4
float[] melbourne1968 = {48.3,6.6,24.6,67.2,100,
                         56.9,39.7,40,18.3,51.5,
                         44.5,53.6};
```

**Assignment 5 Question 2: Planning Help**

*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

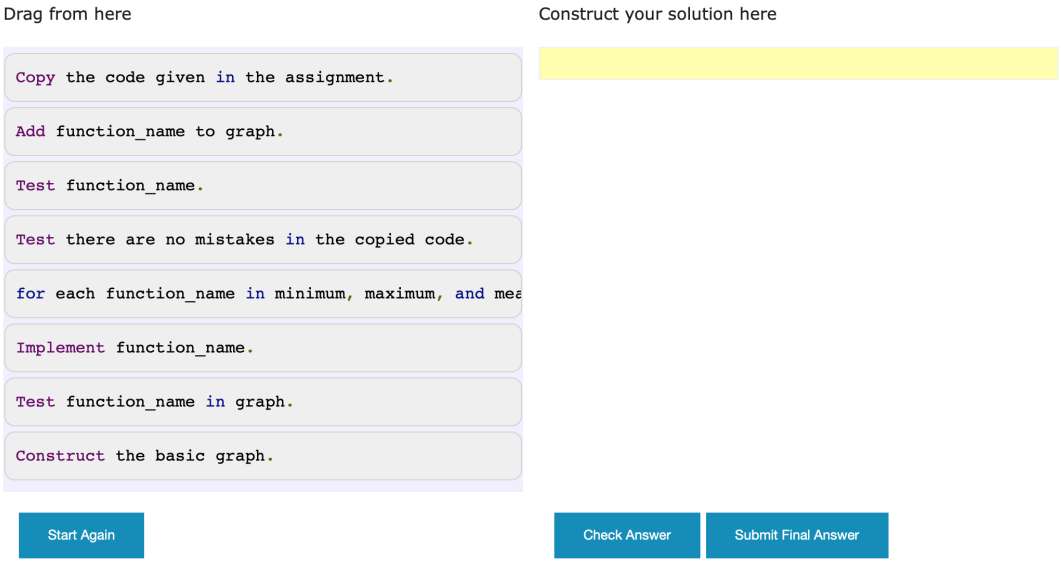
Student ID: (Required) _____

1. Review the graph in the assignment. How does this graph help you understand the problem?
2. There are a few separate programming parts to this assignment. Describe your plan on how you will program these parts.

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.

These planning tasks describe higher level task descriptions typical of how larger programs are planned. This approach enables you to have a broad view of what needs to get solved without being overwhelmed with coding details. Some sections are repeatable tasks for different functions, represented by a loop over these functions written in pseudocode.


Drag from here
Construct your solution here
Copy the code given in the assignment.
Add function_name to graph.
Test function_name.
Test there are no mistakes in the copied code.
for each function_name in minimum, maximum, and mea
Implement function_name.
Test function_name in graph.
Construct the basic graph.
Start Again
Check Answer
Submit Final Answer

### A.2.3 Assignment 6

**Question 1**

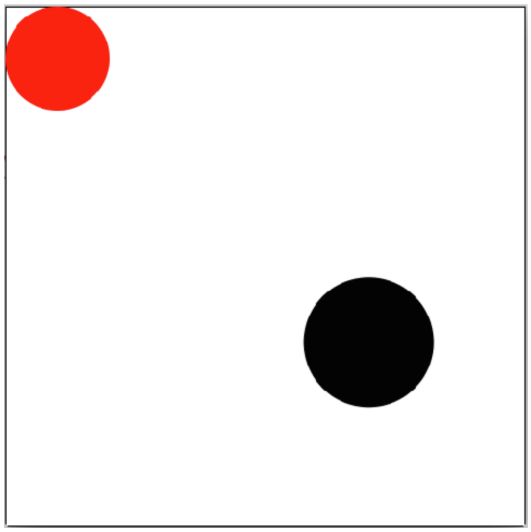
**Assignment 6 Question 1: Hole in One (7 marks)**

There are **two parts** to this assignment. This is the first of the two parts.

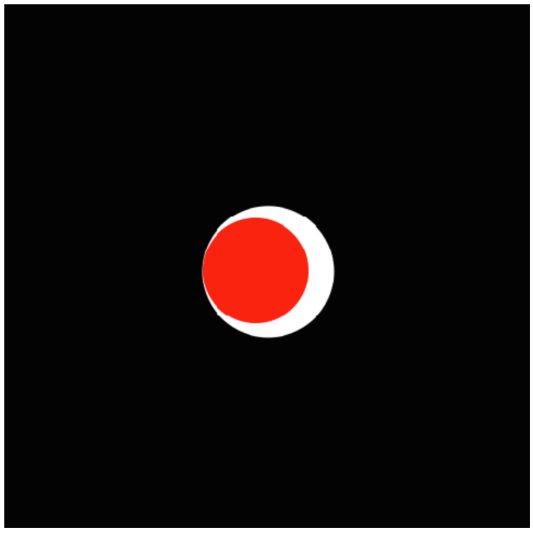
Create a program that allows the user to move a **red ball** into a **hole** using the arrow keys on the keyboard. Your program should meet the following specifications:

- The ball starts in the upper left hand corner.
- The hole should be at a random location, it cannot overlap the ball's beginning position.
- The hole's diameter must be scalable with the sketch size.
- The ball's diameter should be 80% of the hole's diameter.
- The ball changes position when the arrow keys are pressed.
- The ball cannot go off the edge of the sketch.
- When the ball is **completely or partially outside** the hole the background should be white and the hole should be black.
- When the ball is **completely inside** the hole the background should be changed to black and the hole becomes white.
- Include a function named **isBallinHole()** that returns true if the ball is completely within the hole and false otherwise.

Example of display when ball is not in the hole:



Example of display when ball is in the hole:



**Assignment 6 Question 1: Planning Help**

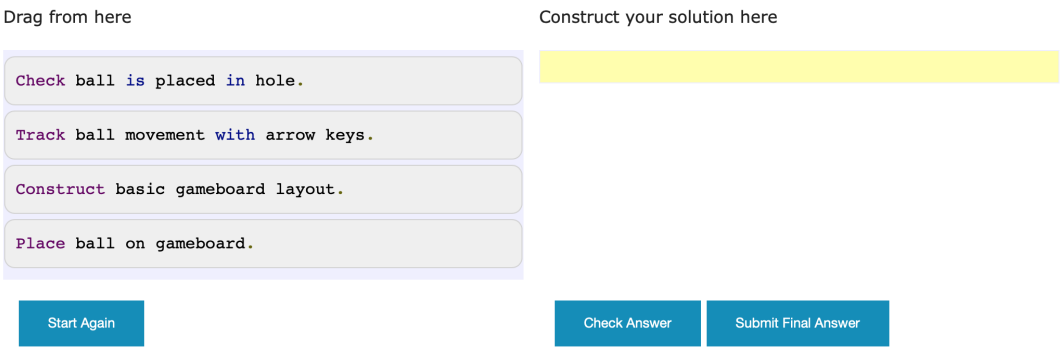
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What have you already learned to help you move the ball around on the canvas?
2. You might need additional information to help you solve the question. What are your plans for seeking out additional information?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



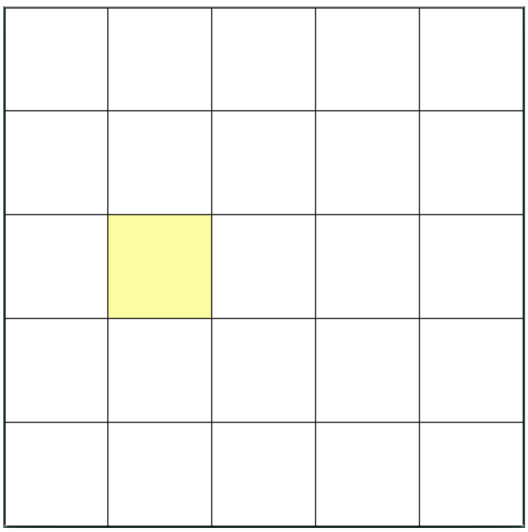
**Question 2**

**Assignment 6 Question 2: Interactive Grid (13 marks)**

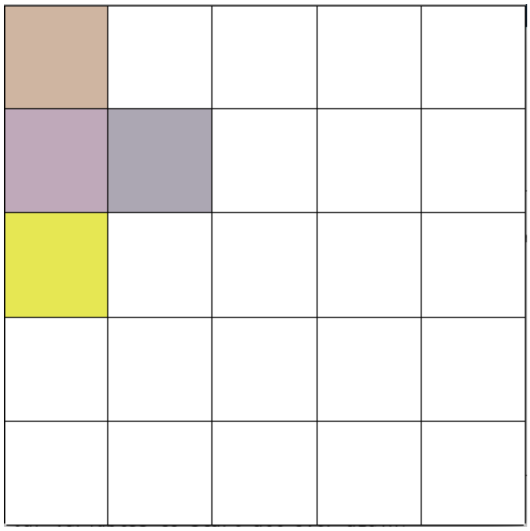
Create a program that allows the user to fill each square in a grid with random colours. Once all of the squares are filled, the grid can be reset by the user. Your program should meet the following specifications:

- The grid should have 5 rows and 5 columns.
- The grid should scale with the width and height of the sketch (the sketch should always be square - equal width and height).
- The grid should be made up of white squares with black outlines.
- When the mouse is over a square that has not been filled it should highlight that square in yellow.
- When the mouse is clicked on a square that has not been filled yet, it should place a random colour within it. (Note: The squares have a decreased opacity in the example below to soften the colours, but this is not required.)
- Once a square has been filled with a random colour it cannot be highlighted or filled again until the entire grid is reset.
- After the last square is filled, the next mouse click will reset the entire grid back to white, allowing the user to fill the grid over again.
- When resetting the grid back to white you must reset the squares one by one. For example you could first reset square (0,0) back to white, and then square (0,1) back to white, and then square (0,2) back to white, and so on. You can choose the order in which the squares are reset.
- Use an appropriate frame rate, such that the resetting animation is easily visible.

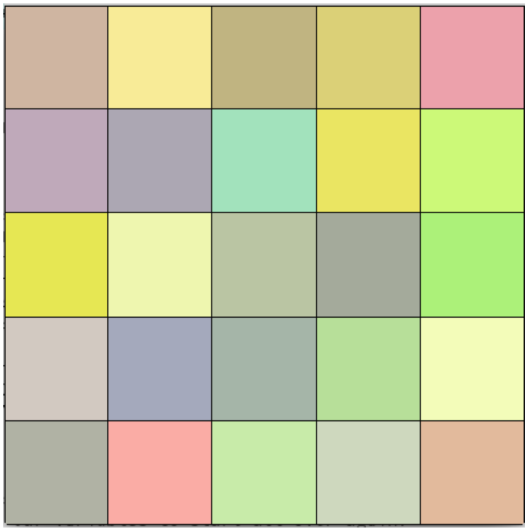
Example of the display before any square is selected:



Example of the display after the user has clicked on some of the squares:



Example of the display after the user has clicked on all of the squares, but before resetting the grid:



**Assignment 6 Question 2: Planning Help**

*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

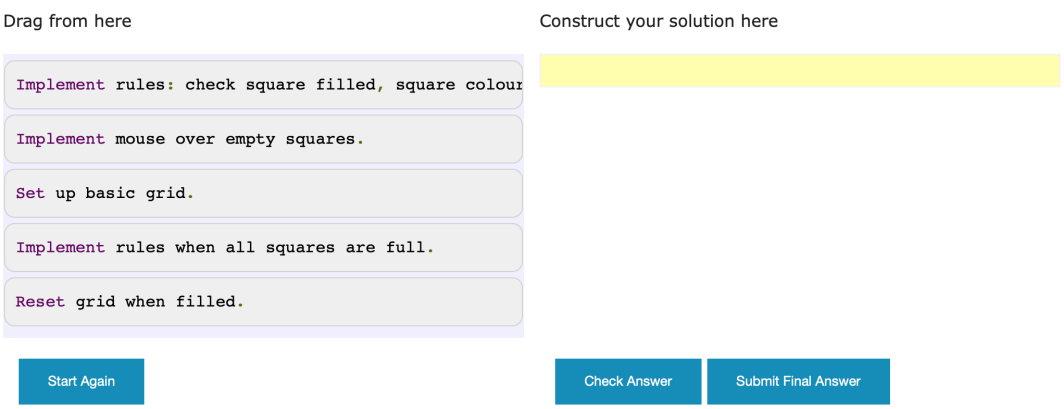
Student ID: (Required) _____

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column

to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



### A.2.4 Assignment 7

**Question 1**

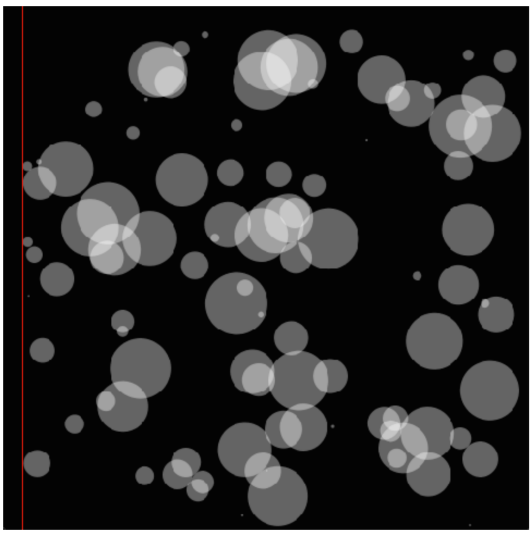
**Assignment 7 Question 1: Dragging Circles (7 marks)**

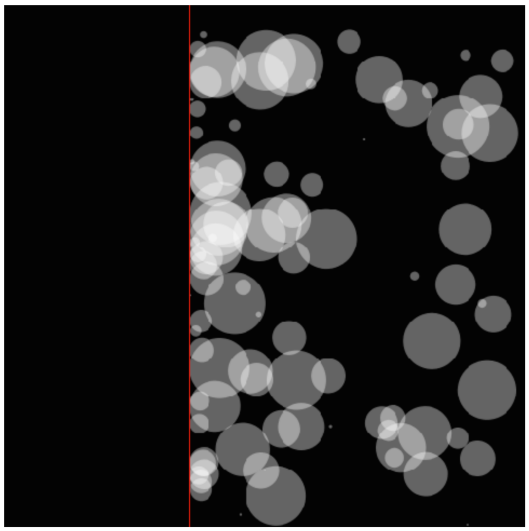
**Program Description**

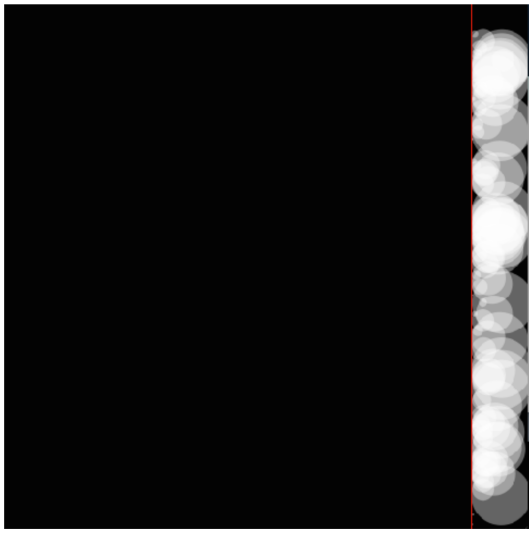
Create a program that animates a number of randomly positioned circles being dragged across the screen. Your program should meet the following specifications:

- The canvas should have a width and height of 500, and have a black background.
- Your program should have a frame rate of 30.
- There should be 100 circles displayed on the screen with the following properties:
  - x and y coordinates should be random.
  - The diameters should be random between 1 and 30.
  - The fill should be white, with an opacity of 100.
  - The entire circle must be on the canvas initially.
- A red vertical line will sweep across the canvas from the left side to the right side.
- As the vertical line sweeps across it should drag any circles that it touches with it.
- Once the vertical line has reached the end of the canvas the animation should reset with new random circles, moving from the right to the left of the screen.

The images below show the animation in action.







**Assignment 7 Question 1: Planning Help**

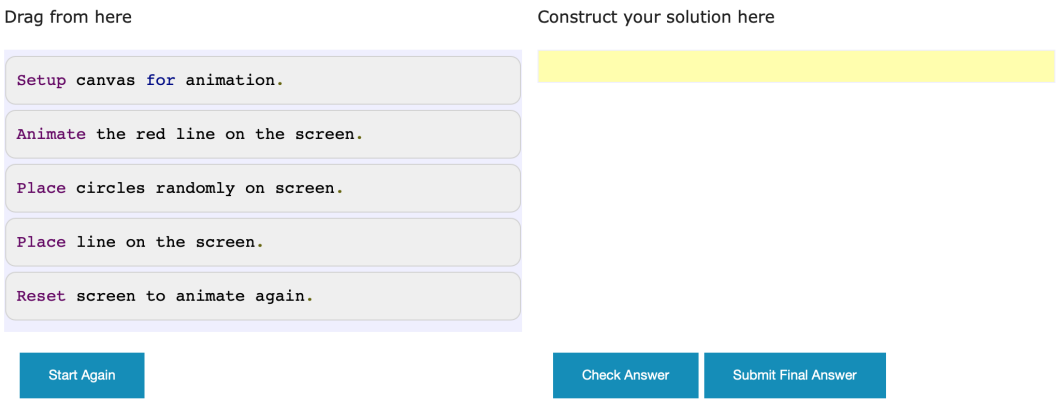
*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s).

Student ID: (Required) _____

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

*Instructions:* This section will help you organise the tasks needed to construct your program. The order of the tasks is the order you can use to develop your programming solution. Move the tasks from the left hand column to the right in the order you feel is the best way to program your solution.

You can check your work using the '*Check Answer*' button. When are done planning, please click '*Submit Final Answer*'.



**Question 2**

**Assignment 7 Question 2: Transpose Function (7 marks)**

**Program Description**

Create a function named **transpose** that takes a 2D integer array as a parameter and returns the transpose of that array. Your function should return an empty array if the array passed in is empty or if it contains arrays of varying lengths. Print the results of your transpose function either to the console or the canvas.

You should test your transpose function with at least 4 different matrices, paying particular attention to edge cases. Your test matrices should be placed in your program code for ease of marking.

You can find more information about transposing here:

`https://www.mathsisfun.com/algebra/matrix-introduction.html`

**Assignment 7 Question 2: Planning Help**

*Instructions:* The following two questions will help you think about the programming assessment and apply it to what you already know. This will help you identify the problem's goal(s). When you are done with your answers, please click the "Submit Final Answer" button.

Student ID: (Required) _____

1. How are you going to try to better understand this problem?
2. What are your plans for trying to solve this problem?

## A.3 Questioning Activities

### A.3.1 Pilot Group Questioning Activities

**Assignment 4 Question 1: Products Sold**

1. What do you already know about collecting and organising data for products sold that might help you complete this assignment?
2. Breaking down the problem into smaller tasks will help you complete the assignment. How would you go about breaking down this problem into smaller tasks?

**Assignment 4 Question 2: Football League List**

1. You might not be familiar with football, but you might already know about sports and competition. How can your current knowledge about sports and competition help you better understand this problem?
2. A part of this assignment is to colour code a team's row based on their statistics. What do you think is a good approach to assigning the colour?

**Assignment 5 Question 1: Hue Function**

1. What ideas do you have for the patterns your program will need to generate?
2. How do you think you will need to alter **displayHue** to create these patterns?

**Assignment 5 Question 2: Products Sold Graph**

1. Review the graph in the assignment. How does this graph help you understand the problem?
2. There are a few separate programming parts to this assignment. Describe your plan on how you will program these parts.

**Assignment 6 Question 1: Coin in the Well**

1. What have you already learned to help you move the coin around on the canvas?
2. You might need additional information to help you solve the question. What are your plans for seeking out additional information?

**Assignment 6 Question 2: Interactive Grid**

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

**Assignment 7 Question 1: Dragging Squares**

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

**Assignment 7 Question 2: Rotation Function**

1. How are you going to try to better understand this problem?
2. What are your plans for trying to solve this problem?

### A.3.2 Pilot Group Questioning Activities

**Assignment 4 Question 1: Averaging Rainfall**

1. What do you already know about collecting and organising data for averaging rainfall that might help you complete this assignment?
2. Breaking down the problem into smaller tasks will help you complete the assignment. How would you go about breaking down this problem into smaller tasks?

**Assignment 4 Question 2: Class Grade List**

1. How can your current knowledge about classroom and grading help you better understand this problem?
2. A part of this assignment is to colour the row containing a student's information based on their grade. What do you think is a good approach to assigning the colour?

**Assignment 5 Question 1: Gradient Function**

1. What ideas do you have for the patterns your program will need to generate?
2. How do you think you will need to alter drawGradient to create these patterns?

**Assignment 5 Question 2: Rainfall Graph**

1. Review the graph in the assignment. How does this graph help you understand the problem?
2. There are a few separate programming parts to this assignment. Describe your plan on how you will program these parts.

**Assignment 6 Question 1: Hole in One**

1. What have you already learned to help you move the ball around on the canvas?
2. You might need additional information to help you solve the question. What are your plans for seeking out additional information?

**Assignment 6 Question 2: Interactive Grid**

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

**Assignment 7 Question 1: Dragging Circles**

1. What do you already know about this topic?
2. How will you apply what you already know to solve this question?

**Assignment 7 Question 2: Transpose Function**

1. How are you going to try to better understand this problem?
2. What are your plans for trying to solve this problem?

# Appendix B

# Interviews and Tests

## B.1 Interviews

### B.1.1 Student Interview

**Interview Questions**

*Instructions*: This interview gives you the opportunity to provide feedback on your experiences working on the programming assignment. Please fill out the questions below.

1. What are three things that stand out in your mind when going through the programming assignment. Why did these things stand out?
2. Did you find yourself stuck in any situation and if so, how were you able to get unstuck?
3. What materials provided to you help you understand the problem? Materials can include assignment text description, the question activity, and the task activity proceeding the assignment text description. A sample of the materials is provided at the end of the survey.
4. Can you describe any positive or negative feelings generated during the session?
5. If you have any negative feelings during the activity, how did you overcome/try to overcome them? Please explain.
6. Please provide your opinion on each of the following parts of the study. Provide how the section was helpful, and how it could be improved. A sample of the materials is provided at the end of the survey.
   (a) Assignment Description:
   (b) The question activity:
   (c) The task activity:
7. Would you like to add any additional comments or feedback that was not covered in the questions?

**For the 2-4 interview sessions:**

1. Did you see any benefits from doing the question and task activities?
2. Did you have any suggestions on how to improve the question and task activities?

## B.2 Tests

### B.2.1 Student Pre- and Post-Test

**'Introduction to Programming' Homework Planning Survey**

This survey gives you the opportunity to think about how you go about doing your assignments. Your feedback will give us, the teachers, a better understanding of what you might need to successfully complete the assignments. Please take a moment to complete this survey.

Student ID: _____

| **Planning Statements** | **Strongly Agree** | **Agree** | **Neither Agree nor Disagree** | **Disagree** | **Strongly Disagree** |
|---|---|---|---|---|---|
| 1. I try to use strategies that have worked in the past. | □ | □ | □ | □ | □ |
| 2. I have a specific purpose for each strategy I use. | □ | □ | □ | □ | □ |
| 3. I am aware of what strategies I use when I study. | □ | □ | □ | □ | □ |
| 4. I find myself using helpful learning strategies automatically. | □ | □ | □ | □ | □ |
| 5. I pace myself while learning in order to have enough time. | □ | □ | □ | □ | □ |
| 6. I think about what I really need to learn before I begin a task. | □ | □ | □ | □ | □ |
| 7. I ask myself questions about the material before I begin. | □ | □ | □ | □ | □ |
| 8. I think of several ways to solve a problem and choose the best one. | □ | □ | □ | □ | □ |
| 9. I read instructions carefully before I begin a task. | □ | □ | □ | □ | □ |
| 10. I organise my time to best accomplish my goals. | □ | □ | □ | □ | □ |

### B.2.2 Usability Study Questionnaire

*Instructions*: This survey gives you the opportunity to provide background information on prior studies that might help you with learning Computer Science. Please fill out the questions below.

1. What strategies do you use in order to solve the problems? For example, what do you do when trying to solve a complex math problem?
2. Sometimes difficult problems can be frustrating. How do you overcome these negative emotions to try and solve the problem?
3. You are given a math homework assignment due in a week. Can you describe what you do during the week to complete the assignment.
4. How do you estimate your programming experience?

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Very inexperienced | ○ | ○ | ○ | ○ | ○ | Very experienced |

5. How do you estimate your programming experience compared to experts with 20 years of practical experience?

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Very inexperienced | ○ | ○ | ○ | ○ | ○ | Very experienced |

6. How do you estimate your programming experience compared to your class mates?

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Very inexperienced | ○ | ○ | ○ | ○ | ○ | Very experienced |

7. Besides Processing.js, how many additional languages do you know?
8. For how many years have you been programming?

### B.2.3 Rainfall Problem Survey

**Rainfall Problem - Survey**
We, the teachers, are investigating ways to improve how we present programming assignments, and your feedback will greatly help us. At the bottom of the survey, we provided the first part of Assignment 4 as reference. Please take a moment to complete the survey.

1. Rate your understanding of the Rainfall Problem statement.

- Completely understood
- Mostly understood
- Mostly did not understand
- Completely did not understand

2. Explain any problems you encountered when reading the Rainfall Problem statement.

3. Did the layout of the Rainfall Problem statement help you in creating your solution?

- Yes, the format was helpful
- No, the format was not helpful
- I did not notice

4. Explain how the Rainfall Problem statement's layout helped you create your solution.

# Bibliography


Adelson, Beth (1984). "When novices surpass experts: The difficulty of a task may increase with expertise". In: *Journal of Experimental Psychology: Learning, Memory, and Cognition* 10 (3), pp. 483–495.

Ahoniemi, Tuukka and Essi Lahtinen (2007). "Visualizations in preparing for programming exercise sessions". In: *Electronic Notes in Theoretical Computer Science (ENTCS)* 178, pp. 137–144.

Akin, Ahmet and Ramazan Abaci (2007). "The validity and reliablity of the Turkish version of the Metacognitive Awareness Inventory". In: *Educational Sciences: Theory and Practice* 7 (2), pp. 671–678.

Aleven, V., J. Sewall, O. Popescu, F. Xhakaj, D. Chand, R. Baker, Y. Wang, G. Siemens, C. Rosé, and D. Gasevic (2005). "The beginning of a beautiful friendship? Intelligent tutoring systems and MOOCs". In: *Artificial Intelligence in Education, Vol. 9112 of the Series Lecture Notes in Computer Science*, pp. 525–528.

Aleven, V., J. Sewall, O. Popescu, M. Ringenberg, M. van Velsen, and S. Demi (2016). "Embedding intelligent tutoring systems in MOOCs and e-learning platforms". In: *ITS '16: Proceedings of the 13th International Conference on Intelligent Tutoring Systems*, pp. 409–415.

Alexander, Christopher, Sara Ishikawa, and Murray Silverstein (1977). *A pattern language: Towns, buildings, construction*. Oxford University Press.

Alexiou, A. and F. Paraskeva (2010). "Enhancing self-regulated learning skills through the implementation of a e-portfolio tool". In: *Procedia Social and Behaviorial Science* 2, pp. 3048–3054.

Allen, Elaine and Christopher A. Seaman (2007). "Likert scales and data analysis". In: *Quality Progress*. Vol. 40. University of New York, pp. 64–65.

An, Yun-Jo and Li Cao (2014). "Examining the effects of metacognitive scaffolding on students' design problem solving and metacognitive skills in an online environment". In: *MERLOT: Journal of Online Learning and Teaching* 10 (4), pp. 552–568.

Anderson, J.R. (2013). *The architecture of cognition*. Hove, East Sussex, United Kingdom: Psychology Press.

Anderson, L. and D.A. Krathwohl (2001). *Taxonomy for Learning, Teaching and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives*. New York: Longman.

Atman, C.J., J.R. Chimka, K.M. Bursic, and H.L. Nachtmann (1999). "A comparison of freshman and senior engineering design processes". In: *Design Studies* 20, pp. 131–152.

Baert, M. (2019). *SimpleScreenRecorder*. https://www.maartenbaert.be/simplescreenrecorder. [Online; accessed 11-Feb-2019].

Baird, J.R. and J.R. Northfield (1995). *Learning from the PEEL experience*. 2nd. Peel Publications.

Barab, S. and K. Squire (2004). "Design-based research: Putting a stake in the ground". In: *Journal of the Learning Series* 13 (1), pp. 1–14.

Barker, Lecia J., Charlie McDowell, and Kimberly Kalahar (2009). "Exploring factors that influence computer science introductory course students to persist in the major". In: *ACM SIGCSE Bulletin* 41.2, pp. 282–286.

Bateson, A.G., R.A. Alexander, and M.D. Murphy (1987). "Cognitive processing differences between novice and expert computer programmers". In: *International Journal Man-Machine Studies* 26, pp. 649–660.

Beck, K. and C. Andres (2004). *Extreme programming explained: Embrace change*. 2nd ed. Addison-Wesley Professional.

Bergin, S., R. Reilly, and D. Traynor (2005). "Examining the role of self-regulated learning on introductory programming performance". In: *ICER '05: Proceedings of the First Annual Conference on International Computing Education Research*, pp. 81–86.

Bergman, Lars R. and Kari Trost (2006). "The person-oriented versus the variable-oriented approach: Are they complementary, opposites, or exploring different worlds?" In: *Person-Centered and Variable-Centered Approaches to Longitudinal Data* 52.3, pp. 601–632.

Biggs, J. B. and K. F. Collis (1982). "The SOLO Taxonomy (Structure of the Observed Learning Outcome)". In: *Evaluating the Quality of Learning*. Ed. by Allen J. Edward. New York, New York, pp. 237–242.

Bishop-Clark, Catherine (1995). "Cognitive style, personality, and computer programming". In: *Computers in Human Behavior* 11 (2), pp. 241–260.

Bloom, Benjamin S. (1956). *Taxonomy of Educational Objectives: The Classification of Educational Goals*. New York: Longmans, Green, pp. 201–207.

Boekaerts, Monique (1996). "Self-regulated learning at the junction of cognition and motivation". In: *European Psychologist* 1, pp. 100–112.

Boekaerts, Monique (2011). "Emotions, emotion regulation, and self-regulation of learning". In: *Handbook of Self-Regulation of Learning and Performance*. Ed. by Zimmerman B. J. and Schunk D. H. New York, NY: Routledge, pp. 408–425.

Bonar, Jeffrey and Elliot Soloway (1985). "Preprogramming knowledge: A major source of misconceptions in novice programmers". In: *Human-Computer Interaction* 1 (2), pp. 133–161.

Boone Jr, Harry N. and Deborah Boone (2012). "Analyzing Likert data". In: *Journal of Extension* 50 (2), pp. 1–5.

Booth, A., D. Papaioannou, and A. Sutton (2012). *Systematic approaches to a successful literature review*. Sage Publications Ltd.

Bouvier, Dennis, Ellie Lovellette, John Matta, Bedour Alshaigy, Brett A. Becker, Michelle Craig, Jana Jackova, Robert McCartney, and Kate Sanders (2016). "Novice programmers and the problem description effect". In: *ITiCSE '16: Proceedings of the 2016 ITiCSE Working Group Reports*, pp. 103–118.

Boyer, Kristy Elizabeth, William Lahti, Robert Phillips, Michael D. Wallis, Mladen A. Vouk, and James C. Lester (2010). "Principles of asking effective questions during student problem solving". In: *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, pp. 460–464.

Briggs, Tom (2005). "Techniques for active learning in CS courses". In: *Journal of Computing Sciences in Colleges* 1 (2), pp. 156–165.

Britton, Sandra, Peter New, Andrew Roberts, and Manjula Sharma (2007). "Investigating Students' Ability to Transfer Mathematics". In: *Transforming a University. The Scholarship of Teaching and Learning Practice*. Ed. by In Angela Brew & Judyth Sachs (Eds.) Sydney: Sydney University Press, pp. 127–140.

Buckley, Jim and Chris Exton (2003). "Bloom's Taxonomy: A framework for assessing programmers' knowledge of software systems". In: *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, pp. 165–174.

Buckley, Michael, Helene Kershner, Kris Schindler, Carl Alphonce, and Jennifer Braswell (2004). "Benefits of using socially-relevant projects in computer science and engineering education". In: *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pp. 482–486.

Carbone, Angela, John Hurst, Ian Mitchell, and Dick Gunstone (2000). "Principles for designing programming exercises to minimise poor learning behaviours in students". In: *ACSE '00 Proceedings of the Australasian conference on Computing education*, pp. 26–33.

Carbone, Angela, John Hurst, Ian Mitchell, and Dick Gunstone (2001). "Characteristics of programming exercises that lead to poor learning tendencies: Part II". In: *ITiCSE '01: Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 93–96.

Caspersen, Michael E. and Jens Bennedsen (2007). "Instructional design of a programming course – A learning theoretic approach". In: *Proceedings of the 3rd International Workshop on Computing Education Research*, pp. 111–122.

Castro, F. and K. Fisler (2017). "Designing a multi-faceted SOLO taxonomy to track program design skills through an entire course". In: *Koli Calling '17: Proceedings of the 17th Koli Calling International Conference on Computing Education Research*, pp. 10–19.

Chalk, P. (2001). "Scaffolding learning in virtual environments". In: *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 85–88.

Chatfield, C. (2004). *The analysis of time series: An introduction*. 6th ed. Boca Raton, FL: Chapman & Hall CRC Press.

Chi, M. T. H., R. Glaser, and E. Rees (1982). "Expertise in problem solving". In: *Advances in the psychology of human intelligence*. Ed. by R. J. Sternberg. Vol. 1. Hillsdale, NJ: Lawrence Erlbaum Associates, Inc, pp. 7–76.

Chi, Michelene T. H., Nicholas de Leeuw, Mei hung Chiu, and Christian LaVancher (1994). "Eliciting self-explanations improves understanding". In: *Cognitive Science* 18, pp. 439–477.

Chmiel, Ryan and Michael C. Loui (2003). "An integrated approach to instruction in debugging computer programs". In: *Proceedings of AEE/IEEE in Education* 3, pp. 1–6.

Collins, Gambrell, and Pressley (2002). *Comprehension strategy instruction that works*. Sundance, pp. 1–7.

Collins, Allan (1985). "Teaching reasoning skills". In: *Thinking and Learning Skills*. Ed. by & R. Glaser S. Chipman J. Segal. Vol. 2. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 579–586.

Collins, Allan, John Seely Brown, and Ann Holum (1991). "Cognitive apprenticeship: Making thinking visible". In: *American Educator* 15 (3), pp. 6–11.

Collins, Allan and Albert Stevens (1983). "A cognitive theory of inquiry teaching". In: *Instructional-design theories and models*. Ed. by C.M. Reigeluth. Norwood, NJ: Lawrence Erlbaum Associates, pp. 203–230.

Conati, Cristina and Kurt Vanlehn (2000). "Toward computer-based support of metacognitive skills: A computational framework to coach self-explanation". In: *International Journal of Artificial Intelligence in Education*, pp. 398–415.

Crandall, B., G. Klein, and R.R. Hoffman (2006). *Working minds: A practitioner's guide to cognitive task analysis*. MIT Press.

Creswell, J.W. (2012). *Educational research: Planning, conducting, and evaluating quantitative and qualitative research*. Educational Research: Planning, Conducting, and Evaluating Quantitative and Qualitative Research. Pearson.

Dadic, T., S. Stankov, and M. Rosic (2008). "Meaningful learning in the tutoring system for programming". In: *Proceedings of the ITI 2008 30th International Conference on Information Technology Interfaces*, pp. 483–488.

de Raadt, Michael, Mark Toleman, and Richard Watson (2004). "Training strategic problem solvers". In: *InRoads - The SIGCSE Bulletin* 36 (2), pp. 48–51.

de Raadt, Michael, Richard Watson, and Mark Toleman (2009). "Teaching and assessing programming strategies explicitly". In: *ACE '09: Proceedings of the Eleventh Australasian Computing Educating Conference*, pp. 45–54.

Denny, Paul, Andrew Luxton-Reilly, and Beth Simon (2008). "Evaluating a new exam question: Parsons problems". In: *Proceeding of the Fourth international Workshop on Computing Education Research*, pp. 113–124.

Détienne, F. (2002). "Software design: Theoretical approaches". In: *Software design—Cognitive aspects*. Ed. by Frank Bott. Berlin, Heidelberg: Springer-Verlag. Chap. 3, pp. 21–41.

Dillon, J.T. (1984). "The classification of research questions". In: *Review of Educational Research* 54, pp. 327–361.

diSessa, Andrea (2014). "The construction of causal schemes: Learning mechanisms at the knowledge level". In: *Cognitive Science* 38 (5), pp. 795–850.

Dreyfus, H.L. and S.E. Dreyfus (1986). *Mind over machine: The power of human intuitive expertise in the era of the computer*. New York: Free Press.

Dropbox, Inc. (2019). *Dropbox*. Available at https://www.dropbox.com. [Online; accessed 01-Nov-2019].

Ebrahimi, Alireza (1994). "Novice programmer errors: Language constructs and plan composition". In: *International Journal Human-Computer Studies* 41 (4), pp. 457–480.

El-Zakhem, Imad H. (2016). "Socratic programming: An innovative programming learning method". In: *International Journal of Information and Education Technology* 6 (3), pp. 247–250.

Ericsson, K. A. and H. A. Simon (1993). *Protocol Analysis: Verbal Reports as Data*. A Bradford Book, London: The MIT Press.

Esser, Frank and Rens Vliegenthart (2017). *Comparative research methods*. Ed. by Jörg Matthes, Christine S. Davis, and Robert F. Potter. John Wiley & Sons Inc, pp. 1–22.

Facione, Peter (2015). "Critical thinking: What it is and why it counts". In: *Insight Assessment*, pp. 1–27.

Falkner, Katrina, Rebecca Vivian, and Nickolas Falkner (2014). "Identifying computer science self-regulated learning strategies". In: *ITiCSE '14: Proceedings of the 2014 ACM Conference on Innovation and Technology in Computer Science Education*, pp. 291–296.

Falkner, Katrina, Claudia Szabo, Rebecca Vivian, and Nickolas Falkner (2015). "Evolution of software development strategies". In: *ICSE '15: Proceedings of the 37th International Conference on Software Engineering*, pp. 243–252.

Fee, Samuel B. and Amanda M. Holland-Minkley (2010). "Teaching computer science through problems, not solutions". In: *Computer Science Education* 20 (2), pp. 129–144.

Feldman, Todd and Julie Zelenski (1996). "The quest for excellence in designing CS1/CS2 assignments". In: *SIGCSE '96: Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, pp. 319–323.

Fincher, S., M. Petre, J. Tenenberg, K. Blaha, T. Chen, D. Chinn, S. Cooper, A. Eckerdal, H. Johnson, R. McCartney, A. Monge, J. Moström, K. Powers, M. Ratcliffe, A. Robins, D. Sanders, L. Schwartzman, B. Simon, C. Stoker, A. Tew, and T. VanDeGrift (2004). "A multi-national, multi-institutional study of student-generated software designs". In: *Koli Calling '04: Proceedings of the 4th Koli Calling International Conference on Computing Education Research*, pp. 20–27.

Fioravanti, Maria Lydia and Ellen Francine Barbosa (2016). "A systematic mapping on pedagogical patterns". In: *2016 IEEE Frontiers in Education Conference (FIE)*, pp. 1–9.

Fisler, Kathi (2014). "The recurring rainfall problem". In: *ICER '14: Proceedings in the tenth annual conference on International Computing Education Research Conference*, pp. 35–42.

Fisler, Kathi and Francisco Enrique Vicente Castro (2017). "Sometimes, rainfall accumulates: Talk-alouds with novice functional programmers". In: *ICER '17: Proceedings of the 2017 ACM Conference on International Computing Education Research Conference*, pp. 12–20.

Fisler, Kathi, Shriram Krishnamurthi, and Janet Siegmund (2016). "Modernizing plan-composition studies". In: *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 211–216.

Fry, Ben and Casey Raes (2018). *Processing.js*. Available at http://processingjs.org, version 1.4.8. [Online; accessed 20-Feb-2019].

Gal-Ezer, Judith, Dvir Lanzberg, and Daphna Shahak (2004). "Interesting basic problems for CS1". In: *ACM SIGCSE Bulletin*, pp. 275–275.

Garcia, Rita, Katrina Falkner, and Rebecca Vivian (2018). "Systematic literature review: Self-Regulated learning strategies using e-learning tools for computer science". In: *Computers & Education* 123, pp. 150–163.

Garner, S. (2007). "A program design tool to help novices learn programming". In: *ASCILITE '07: Proceedings of Annual Conference of the Australasian Society for Computers in Learning in Tertiary Education*, pp. 321–324.

Gerdes, Alex, Johan Juering, and Bastiaan Heeren (2012). "An interactive functional programming tutor". In: *ITiCSE '12: Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, pp. 250–255.

Ghefaili, Aziz (2003). "Cognitive apprenticeship, technology, and the contextualization of learning environments". In: *Journal of Educational Computing, Design Online Learning*, pp. 1–27.

Gibbs, Graham R. (2007). "Thematic coding and categorizing". In: *Analyzing Qualitative Data*. London: SAGE Publications Ltd. Chap. 4, pp. 38–56.

Gick, Mary L. (1986). "Problem-solving strategies". In: *Educational Psychologist* 21 (1&2), pp. 99–120.

Ginat, David (2003). "The novice programmers' syndrome of design-by-keyword". In: *ITiCSE '03: Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 154–157.

Ginat, David (2008). "Design disciplines and non-specific transfer". In: *Informatics Education - Supporting Computational Thinking. ISSEP 2008. Lecture Notes in Computer Science*. Vol. 5090. Berlin, Heidelberg: Springer, pp. 87–98.

Godwin-Jones, R. (2010). "Emerging technologies: New developers in web browsing and authoring". In: *Language Learning & Technology* 14 (1), pp. 9–15.

Google LLC (2019a). *Google Forms*. Available at https://www.google.com/forms/about/. [Online; accessed 04-Oct-2019].

Google LLC (2019b). *Google Sheets*. Available at https://www.google.com/sheets/about/. [Online; accessed 04-Oct-2019].

Graesser, Arthur and Natalie Person (1994). "Question asking during tutoring". In: *American Educational Research Journal* 31 (1), pp. 104–137.

Greasley, Pete (2008). *Quantitative Data Analysis with SPSS*. 1st. Milton Keynes, UK: Open University Press.

Greeno, J.G. (1980). "Trends in the theory of knowledge for problem solving". In: *Problem Solving and Education: Issues in Teaching and Research*. Ed. by D.T. Tuma & F. Reif (Eds.) Hillsdale, NJ: Lawrence Baum Associates Inc., pp. 9–23.

Greeno, J.G. and H.A. Simon (1988). "Problem solving and reasoning". In: *Stevens' handbook of experimental psychology: Perception and motivation*. Ed. by G. Lindzey & R. D. Luce (Eds.) R. C. Atkinson R. J. Herrnstein. Oxford, England: John Wiley & Sons, pp. 589–672.

Guzdial, M., L. Hohmann, M. Konneman, C. Walton, and E. Soloway (1998). "Supporting programming and learning-to-program with an integrated CAD and scaffolding workbench". In: *Interactive Learning Environments* 6 (1-2), pp. 143–179.

Guzdial, Mark (2010). "Does contextualized computing education help?" In: *ACM Inroads* 1.4, pp. 4–6.

Haddaway, N.R., P. Woodcock, B. Macura, and A. Collins (2015). "Making literature reviews more reliable through application of lessons from systematic reviews". In: *Conservation Biology* 29 (6), pp. 1596–1605.

Harms, K., J. Chen, and F. Kelleher (2016). "Distractors in Parsons problems decrease learning efficiency for young novice programmers". In: *ICER '16: Proceeds of the 2016 ACM Conference on International Computing Education Research*, pp. 241–250.

Hashim, Khairuddin and Nurul Naslia Khairuddin (2009). "Software engineering assessments and learning outcomes". In: *SEPADS '09: Proceedings of the 8th WSEAS International Conference on Software engineering, parallel and distributed systems*, pp. 131–134.

Hausmann, Robert G.M. and Michelene T.H. Chi (2002). "Can a computer interface support self-explaining? Cognitive training". In: *The 42nd ACM Technical Symposium on Computer Science Education* 7 (1), pp. 4–14.

Hazzan, Orit, Yael Dubinsky, Larisa Eidelman, and Victoria Sakhnini (2006). "Qualitative research in computer science education". In: *ACM SIGCSE Bulletin* 38 (1), pp. 408–412.

Hoadley, C. and C. Cox (2009). "What is design knowledge and how do we teach it". In: *Educating Learning Technology Designers*. Routledge. Chap. 2, pp. 19–34.

Howard, R.A., C.A. Carver, and W.D. Lane (1996). "Felder's learning styles, Bloom's Taxonomy, and the Kolb learning cycle: Tying it all together in the CS2 course". In: *SIGCSE '96: Proceedings of the Twenty-Fifth SIGCSE Technical Symposium in Computer Science Education*, pp. 227–231.

Hu, M., M. Winikoff, and S. Cranefield (2012). "Teaching novice programming using goals and plans in a visual notation". In: *ACE '12: 14th Australian Computing Education Conference*, pp. 43–52.

Hume, Gregory, Joel Michael, Allen Rovick, and Martha W. Evens (1996). "Hinting as a tactic in one-on-one tutoring". In: *Journal of the Learning Sciences* 5 (1), pp. 23–47.

Järvelä, S., P. Kirschner, A. Hadwin, H. Järvenoja, J. Malmberg, M. Miller, and J. Laru (2016). "Socially shared regulation of learning in CSCL: understanding and prompting individual- and group-level shared regulatory activities". In: *International Journal of Computer-Supported Collaborative Learning* 11 (3), pp. 263–280.

Jeffries, R., A.A. Turner Polson, and M.E. Atwood (1981). "The processes involved in designing software". In: *Cognitive Skills and their Acquisition*. Ed. by J.R. Anderson. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 255–283.

Jin, Wei (2008). "Pre-programming analysis tutors help students learn basic programming concepts". In: *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* 40 (1), pp. 276–280.

Joughin, Gordon (2010). "The hidden curriculum: a critical review of research into the influence of summative assessment on learning". In: *Assessment & Evaluation in Higher Education* 35 (3), pp. 335–345.

Kaasbøll, Jens (1998). "Teaching critical thinking and problem defining skills". In: *Education and Information Technologies* 3 (2), pp. 101–117.

Karavirta, Ville, Petri Ihantola, Juha Helminen, and Mike Hewner (2019). *js-parsons*. Available at https://github.com/js-parsons/js-parsons. [Online; accessed 23-Aug-2019].

Kerry, T. (1987). "Classroom questions in England". In: *Questioning Exchange* 1 (1), pp. 32–33.

Khairuddin, Nurul Naslia and Khairuddin Hashim (2008). "Application of Bloom's Taxonomy in software engineering assessments". In: *ACS '08: Proceedings of the 8th conference on Applied computer science*, pp. 66–69.

Kinnunen, P. and B. Simon (2010). "Experiencing programming assignments in CS1: The emotional toll". In: *ICER '10: Proceedings of the Sixth Annual Conference on International Computing Education Research*, pp. 77–85.

Kinnunen, Päivi and Beth Simon (2011). "My program is ok – am I? Computing freshmen's experiences of doing programming assignments". In: *Computer Science Education*, pp. 1–28.

Kizilcec, R., M. Pérez-Sanagustín, and J. Maldonado (2017). "Self-regulated learning strategies predict learner behavior and goal attainment in massive open online courses". In: *Computers & Education* 104, pp. 18–33.

Knobelsdorf, Maria, Christoph Kreitz, and Sebastian Böhne (2014). "Teaching theoretical computer science using a cognitive apprenticeship approach". In: *SIGCSE '14: Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pp. 67–72.

Köppe, Christian and Leo Pruijt (2014). "Improving students' learning in software engineering education through multi-level assignments". In: *CSERC '14: Proceedings of the Computer Science Education Research Conference*, pp. 57–62.

Kramer, J. (2007). "Is abstraction the key to computing?" In: *Communication of the ACM* 50, pp. 36–42.

Kuittinen, Marja and Jorma Sajaniemi (2004). "Teaching roles of variables in elementary programming courses". In: *ITiCSE '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pp. 57–61.

Kussmaul, Clifton L. (2008). "Scaffolding for multiple assignment projects in CS1 CS2". In: *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 873–876.

Lakanen, Antti-Jussi, Vesa Lappalainen, and Ville Isomöttönen (2015). "Revisiting rainfall to explore exam questions and performance on CS1". In: *Koli Calling '15: Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pp. 40–49.

Lane, Chad and Kurt VanLehn (2003). "Coached program planning: Dialogue-based support for novice program design". In: *SIGCSE '03: Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pp. 148–152.

Lane, Chad and Kurt VanLehn (2005). "Teaching the tacit knowledge of programming to novices with natural language tutoring". In: *Computer Science Education* 15 (3), pp. 183–201.

Layman, Lucas, Laurie Williams, and Kelli Slaten (2007). "Note to self: Make assignments meaningful". In: *SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, pp. 459–463.

Le, Nguyen-Thinh and Nico Huse (2016). "Evaluation of the formal methods for the Socratic method". In: *ITS '16: International Conference on Intelligent Tutoring Systems*, pp. 68–78.

Lister, Raymond, Beth Simon, Errol Thompson, Jacqueline L. Whalley, and Christine Prasad (2006). "Not seeing the forest for the trees: Novice programmers and the SOLO Taxonomy". In: *ITiCSE '06: Proceedings on the 11th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 118–122.

Lorenzen, Torben, Lee Mondshein, Abdul Sattar, and Seikyung Jung (2012). "A code snippet library for CS1". In: *ACM Inroads* 3 (1), pp. 41–45.

Lovellette, Ellie, John Matta, Dennis Bouvier, and Roger Frye (2017). "Just the numbers: An investigation of contextualization of problems for novice programmers". In: *SIGCSE '17: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, pp. 393–398.

Marcus, Nadine, Martin Cooper, and John Sweller (1996). "Understanding instructions". In: *Journal in Educational Psychology* 88 (1), pp. 49–63.

Margulieux, Lauren E. and R. Catrambone (2014). "Improving program solving performance in computer-based learning environments through subgoal labels". In: *L@S '14: Proceedings of the first ACM conferences Learning@ scale conference*, pp. 149–150.

Marshall, Catherine and Gretchen B. Rossman (1999). *Designing Qualitative Research*. 3rd. London: Sage Publications.

Mayer, Richard (1983). *Thinking, problem solving, condition*. New York: Freeman.

Mead, Jerry, Simon Gary, John Hamer, Richard James, Juha Sorva, Caroline St. Clair, and Lynda Thomas (2006). "A cognitive approach to identifying measurable milestones for programming skill acquisition". In: *ITiCSE '06: Working group reports on ITiCSE on Innovation and Technology in Computer Science Education*, pp. 182–194.

Merriënboer, Jeroen J.G.Van and Fred G.W.C. Paas (1990). "Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice". In: *Computers in Human Behavior* 6 (3), pp. 273–289.

Metcalfe, Janet and Arthur P. Shimamura (1994). *Metacognition: Knowing about knowing*. Cambridge, MA, US: The MIT Press.

Morrison, Briana B., Lauren E. Margulieux, and Mark Guzdial (2015). "Subgoals, context, and worked examples in learning and computing problem solving". In: *ICER '15: Proceedings of the eleventh annual International Conference on International Computing Education Research*, pp. 21–29.

Morrison, Briana B., Lauren E. Margulieux, Barbara Ericson, and Mark Guzdial (2016). "Subgoals help students solve Parsons problems". In: *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, pp. 42–47.

Muller, Orna (2005). "Pattern oriented instruction and the enhancement of analogical reasoning". In: *ICER '05: Proceedings of the 2005 International Workshop on Computing Education Research*, pp. 57–67.

Nelson, Greg L. (2017). "Comprehension-first pedagogy and adaptive, intrinsically motivated tutorials". In: *ICER '17: Proceedings of the 2017 ACM Conference on International Computing Education Research*, pp. 287–288.

Nelson, Greg L., Benjamin Xie, and Andrew J. Ko (2017). "Comprehension first: Evaluating a novel pedagogy and tutoring system for program tracing in CS1". In: *ICER '17: Proceedings of the 2017 ACM Conference on International Computing Education Research*, pp. 42–51.

Nelson, L. (1970). *Progress and Regress in Philosophy: From Hume and Kant to Hegel and Fries*. Vol. 1. Oxford: Basil Blackwell.

Newell, A. and H.A. Simon (1972). *Human Problem Solving*. Englewood Cliffs, NJ: Prentice-Hall.

Nicholls, J. (1992). "Students as educational theorists". In: *Student Perceptions in the Classroom*. Ed. by D.H. Schunk and J.L. Meece. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 267–286.

Nielsen, Rodney D., Jason Buckingham, Gary Knoll, Ben Marsh, and Leysia Palen (2008). "A taxonomy of questions for question generation". In: *Proceedings of the 1st Workshop on Question Generation Shared Task and Evaluation Challenge*, pp. 25–26.

Object Management Group (OMG) (2017). *What is UML?* http://www.uml.org/what-is-uml.htm.

Oliver, Dave, Tome Dobele, Myles Greber, and Tim Roberts (2004). "This course has a Bloom rating of 3.9". In: *ACE '06: Proceedings of the 8th Australian conference on Computing education*, pp. 227–231.

Ormerod, T. C. and J. Ridgway (1999). "Developing task design guides through cognitive studies of expertise". In: *ECCS '99: European Conference on Cognitive Science*, pp. 401–410.

Ormerod, Thomas C. (2004). "Chapter 3. Planning and ill-defined problems". In: *The Cognitive Psychology*. Ed. by Robin Morris and Geoff Ward, pp. 53–70.

Paas, Fred, Alexander Renkl, and John Sweller (2004). "Cognitive load theory: Instructional implications of the interaction between information structures and cognitive architecture". In: *Instructional Science* 32 (1–2), pp. 1–8.

Paris, S. and J. Turner (1994). "Situated motivation". In: *Student Motivation, Cognition and Learning: Essays in Honor of Wilbert J. McKeachie*. Ed. by P.R. Pintrich, D.R. Brown, and C.E. Weinstein. Hillsdale, NJ: Lawrence Erlbaum Associates, pp. 213–237.

Parsons, Dale and Patricia Haden (2006). "Parson's programming puzzles: A fun and effective learning tool for first programming courses". In: *ACE '06: Proceedings of the 8th Australian conference on Computing education*, pp. 157–163.

Paul, Richard and Linda Elder (2007). *Critical thinking: The art of Socratic questioning*. Dilton Beach, CA: The Foundation for Critical Thinking.

Pennington, Nancy (1987). "Comprehension strategies in programming". In: *Empirical Studies of Programmers: Second Workshop*, pp. 100–113.

Pepe, Kadir (2012). "A research of the relationship between study skills of students and their GPA". In: *Procedia - Social and Behavioral Sciences* (47), pp. 1048–1057.

Perkins, David and Fey Martin (1985). "Fragile knowledge and neglected strategies in novice programmers". In: *Papers presented at the first workshop on empirical studies of programmers*, pp. 213–229.

Perrig, Walter and Walter Kintsch (1985). "Propostitional and situational representations of text". In: *Journal of Memory and Language*, pp. 503–518.

Petersen, Andrew, Michelle Craig, and Daniel Zingaro (2011). "Reviewing CS1 exam question content". In: *SIGCSE '11: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pp. 631–636.

Pickvance, Christopher G. (2001). "Four varieties of comparative analysis". In: *Journal of Housing and the Built in Environment* 16, pp. 7–28.

Pillay, Nelishia (2003). "Developing intelligent programming tutors for novice programmers". In: *Inroads - The SIGCSE Bulletin* 35 (2), pp. 78–82.

Pintrich, P. (2000). "Multiple goals, multiple pathways: The role of goal orientation in learning and achievement". In: *Journal of Educational Psychology* 92 (3), pp. 544–555.

Pintrich, P.R., C.F. Berger, and P.M. Stemmer (1987). "Students' programming behavior in a Pascal course". In: *Journal of Research in Science Teaching* 24 (5), pp. 451–466.

Popovic, Vesna and Ben J. Kraal (2010). "Expertise in software design: Novice and expert models". In: *Proceedings of Studying Professional Software Design*, pp. 1–7.

Porter, Leo, Daniel Zingaro, and Raymond Lister (2014). "Predicting student success using fine grain clicker data". In: *ICER '14: Proceedings of the tenth annual conference on International Computing Education Research Conference*, pp. 51–58.

Powell, M. B., R. P. Fisher, and R. Wright (2005). *Investigative Interviewing*. New York, NY, US: The Guilford Press, pp. 11–42.

Prather, James, Raymond Pettit, Brett A. Becker, Paul Denny, Dastyni Loksa, Alani Peters, Zachary Albrecht, and Krista Masci (2019). "First things first: Providing metacognitive scaffolding for interpreting problem prompts". In: *SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pp. 531–537.

Qian, Yizhou and James Lehman (2017). "Students' misconceptions and other difficulties in introductory programming: A literature review". In: *ACM Transactions on Computing Education* 18 (1), pp. 1–24.

Ragonis, Noa (2012). "Type of questions - The case of computer science". In: *Olympiads in Informatics* 6, pp. 115–132.

Rajlich, Václav and Norman Wilde (2002). "The role of concepts in program comprehension". In: *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, pp. 271–278.

Ramalingam, Vennila, Deborah LaBelle, and Susan Wiedenbeck (2004). "Self-efficacy and mental models in learning to program". In: *ITiCSE '04: Proceedings of the 9th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 171–175.

Reitman, Walter R. (1965). *Cognition and thought. An information processing approach*. New York: John Wiley & Sons, Inc.

Ricken, Mathias (2005). *Assignments for an objects-first introductory software engineering curriculum*. Available at http://ricken.us/research/a4obj1st/a4obj1st.pdf. [Online; accessed 20-Feb-2020].

Riley, D.D. (1981). "Teaching problem solving in an introductory computer science class". In: *SIGCSE '81: Proceedings of the twelfth SIGCSE technical symposium on Computer science education*, pp. 244–251.

Rist, R.S. (1995). "Program structure and design". In: *Cognitive Science* 19, pp. 507–562.

Robillard, P.N. (1999). "The role of knowledge in software development". In: *Communications of the ACM* 42 (1), pp. 87–92.

Robins, A., J. Rountree, and N. Rountree (2003). "Learning and teaching programming: A review and discussion". In: *Computer Science Education* 13 (2), pp. 137–172.

Rocca, Kelly A. (2010). "Student participation in the college classroom: An extended multidisciplinary literature review". In: *Communication Education* 59 (2), pp. 185–213.

Roll, Ido, Vincent Aleven, Bruce M. Mclaren, and Kenneth R. Koedinger (2007). "Designing for metacognition—applying cognitive tutor principles to the tutoring of help seeking". In: *Metacognition and Learning* 2 (2), pp. 125–140.

Rombach, H.D. (1990). "Design measurement: Some lessons learned". In: *IEEE Software*, pp. 17–25.

Rum, Siti and Maizatul Ismail (2016). "Metacognitive awareness assessment and introductory computer programming course achievement at university". In: *The International Arab Journal of Information Technology* 13.6, pp. 667–676.

Ruocco, Anthony S. (2001). "Experiences in threading UML throughout a computer science program". In: *IEEE Transactions on Education* 46 (2), pp. 226–228.

Sanders, I. and C. Mueller (2000). "A fundamentals-based curriculum for first year computer science". In: *SIGCSE '00: Proceedings Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pp. 227–231.

Schaafstal, A.M. (1999). "Diagnostic skill in progress operation: A comparison between experts and novices". PhD thesis. University of Groningen, The Netherlands.

Schoenfeld, Alan H (1992). "Learning to think mathematically: Problem solving, metacognition, and sense making in mathematics". In: *NCTM Handbook of research on mathematics teaching and learning*. Ed. by D.A. Grouws. Macmillan, pp. 334–370.

Schraw, G. and R.S. Dennison (1994). "Assessing metacognitive awareness". In: *Contemporary Educational Psychology* 19, pp. 460–475.

Schraw, G., K.Crippen, and K. Hartley (2006). "Promoting self-regulation in science education: Metacognition as part of a broader perspective on learning". In: *Research in Science Education* 36 (1–2), pp. 111–139.

Schulte, Carsten, Teresa Busjahn, Tony Clear, James H. Paterson, and Ahmad Taherkhani (2010). "An introduction to program comprehension for computer science educators". In: *ITiCSE '10: Proceedings of the 2010 ITiCSE Working Group*, pp. 65–86.

Scott, T. (2003). "Bloom's Taxonomy applied to testing in computer science". In: *Proceedings of the 12th Annual CCSC Rocky Mountain Conference, Consortium for Computing Sciences in Colleges*, pp. 267–274.

Seppälä, Otto, Petri Ihantola, Essi Isohanni, Juha Sorva, and Arto Vihavainen (2015). "Do we know how difficult the rainfall problem is?" In: *Koli Calling '15: Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pp. 87–96.

Settle, Amber, Arto Vihavainen, and Craig S. Miller (2014). "Research directions for teaching programming online". In: *CSCE '14: Proceedings of the 10th International Conference on Frontiers in Education: Computer Science and Computer Engineering*, pp. 1–7.

Sharmin, Sadia, Daniel Zingaro, Lisa Zhang, and Clare Brett (2019). "Impact of open-ended assignments on student self-efficacy in CS1". In: *CompEd '19: Proceedings of the ACM Conference on Global Computing Education*, pp. 215–221.

Silbert, Jerry and Marcy Stein (1990). *Direct Instruction Mathematics*. 2nd. Columbus, OH: Merrill.

Simon (2013). "Soloway's rainfall problem has become harder". In: *LaTiCE '13: International Conference on Teaching and Learning in Computing and Engineering*, pp. 130–135.

Smith, P.L. and T.J. Ragan (1999). *Instructional design*. New York: John Wiley & Sons, Inc.

Smith III, J.P, A.A. diSessa, and J. Roschelle (1993). "Misconception reconceived: A constructivist analysis of knowledge in transition". In: *The Journal of the Learning Sciences* 3 (2), pp. 115–163.

Soloway, E., J. Bonar, J. Greenspan, and K. Ehrlich (1982). "What do novices know about programming?" In: *Directions in Human-Computer Interactions*, pp. 27–54.

Soloway, Elliot (1986). "Learning to program = Learning to construct mechanisms and explanations". In: *Communications of the ACM* 29 (9), pp. 850–858.

Soloway, Elliot, Jeffrey Bonar, and Kate Ehrlich (1983). "Cognitive strategies and looping constructs: An empirical study". In: *Communications of the ACM* 26 (11), pp. 853–860.

Sonnentag, Sabine (1998). "Expertise in professional software design: A process study". In: *Journal of Applied Psychology* 83 (5), pp. 703–715.

Spoher, J. and E. Soloway (1989). *Studying the novice programmer*. Hilldale, NJ. Lawrence Erlbaum, pp. 412–413.

Spohrer, James C. (1992). *MARCEL: Simulating the Novice Programmer*. Intellect Books.

Stevens, Albert L. and Allan Collins (1977). "The goal structure of a Socratic tutor". In: *ACM '77: Proceedings of the 1977 Annual Conference*, pp. 256–263.

Stevenson, Daniel E. and Paul J. Wagner (2006). "Developing real-world programming assignments for CS1". In: *ITiCSE '06: Proceedings of the 11th Annual Conference on Innovation and Technology in Computer Science Education*, pp. 158–162.

Stolcke, Andres, Klaus Ries, Noah Coccaro, Elizabeth Shriberg, Rebecca Bates, Daniel Jurafsky, Paul Taylor, Rachel Martin, Carol Van Ess-Dykema, and Marie Meteer (2000). "Dialogue act modeling for automatic tagging and recognition of conversational speech". In: *Computational Linguistics* 26 (3), pp. 339–373.

Sutcliffe, A.G. and N.A.M. Maiden (1992). "Analysing the novice analyst: Cognitive models in software engineering". In: *International Journal Man-Machine Studies* 36, pp. 719–740.

Sweller, John (1988). "Cognitive load during problem solving: Effects on learning". In: *Cognitive Science* 12, pp. 257–285.

Sweller, John, Paul Ayres, and Slava Kalyuga (2011). "Instrinsic and extraneous cognitive Load". In: *Cognitive Load Theory*. Ed. by J. Michael Spector and Susanne P. Lajoie. Springer, pp. 57–69.

Sweller, John and Marvine Levine (1982). "Effects of goal specificity on means–ends analysis and learning". In: *Effects of goal specificity on means–ends analysis and learning* 8 (5), pp. 463–474.

Sweller, John, Jeroen J. G. van Merriënboer, and Fred G. W. C. Paas (1998). "Cognitive architecture and instructional design". In: *Educational Psychology Review* 10 (3), pp. 251–296.

Thomas, Anne (1993). *Study skills*. Vol. 36. 5. Oregon School Study Council.

Thompson, Errol, Andrew Luxton-Reilly, Jacqueline L. Whalley, Minjie Hu, and Phil Robbins (2008). "Bloom's Taxonomy for CS assessment". In: *ACE '08: Tenth Australasian Computing Education Conference*, pp. 155–161.

Thomson, S.B. (2011). "Sample size and grounded theory". In: *JOAGG: Journal of Administration & Governance* 5.1, pp. 45–52.

Tikhonova, Elena and Natalia Kudinova (2015). "Sophisticated thinking: Higher order thinking skills". In: *Journal of Language and Education* 1 (3), pp. 12–23.

Turkay, Cagatay, Erdem Kaya, Selim Balcisoy, and Helwig Hauser (2017). "Designing progressive and interactive analytics process for high-dimensional data analysis". In: *IEEE Transactions on Visualization and Computer Graphics* 23 (1), pp. 131–140.

van Dijk, T.A. and W. Kintsch (1983). *Strategies of discourse comprehension*. New York: Academic Press.

van Someren, Maarten W., Yvonne F. Barnard, and Jocobijn A.C. Sandberg (1994). *The think aloud method*. Academic Press.

van Velzen, J. (2016). *Metacognitive learning: Advancing learning by developing general knowledge of the learning process*. Switzerland: Springer International Publishing.

Veerasamy, Ashok Kumar, Daryl D'Souza, and Mikko-Jussi Laakso (2016). "Identifying novice student programming misconceptions and errors from summative assessments". In: *Journal of Education Technology Systems* 45 (1), pp. 50–73.

Venables, Anne, Grace Tan, and Raymond Lister (2009). "A closer look at tracing, explaining and code writing skills in the novice programmer". In: *ICER '09: Proceedings of the fifth international workshop on Computing education research workshop*, pp. 117–128.

Vihavainen, A., M. Paksula, and M. Luukkainen (2011a). "Extreme apprenticeship method in teaching programming for beginners". In: *SIGCSE '11: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pp. 93–98.

Vihavainen, A., M. Paksula, and M. Luukkainen (2011b). "Extreme apprenticeship method in teaching programming for beginners". In: *SIGCSE '11: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, pp. 93–98.

Vihavainen, Arto, Craig S. Miller, and Amber Settle (2015). "Benefits of self-explanation in introductory programming". In: *SIGCSE '15: Proceedings on the 46th ACM Technical Symposium on Computer Science Education*, pp. 284–289.

Villiger, Mark (1985). "The factual framework: Codification in past and present". In: *Customary International Law and Treaties*, pp. 63–113.

Violet, Simone (1991). "Modelling and coaching of relevant metacognitive strategies for enhancing university students' learning". In: *Learning and Instruction* 1, pp. 319–336.

Webster, Elizabeth A. and Allyson F. Hadwin (2014). "Emotions and emotion regulation in undergraduate studying: Examining students' reports from a self-regulated learning perspective". In: *Educational Psychology: An International Journal of Experimental Educational Psychology* 35 (7), pp. 794–818.

Weusijana, Baba Kofi A., Christopher K. Reisbeck, and Joseph T. Walsh Jr (2004). "Fostering reflection with Socratic tutoring software: Results of using inquiry teaching strategies with web-based HCI techniques". In: *ICLS '04: Proceedings of the 6th International Conference on Learning Sciences*, pp. 561–567.

Whalley, Jacqueline L., Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P.K. Ajith Kumar, and Christine Prasad (2006). "An Australasian study of reading and comprehension skills in novice programmers, using the Bloom and SOLO Taxonomies". In: *ACE '06: Eighth Australasian Computing Education Conference*, pp. 353–382.

Wilson, Judith D. (1987). "A Socratic approach to helping novice programmers debug programs". In: *SIGCSE '87: Proceedings of the 18th SIGCSE Technical Symposium on Computer Science Education*, pp. 179–182.

Winne, P. (2011). "A cognitive and metacognitive analysis of self-regulated learning". In: *Handbook of self-regulation of learning and performance*. Ed. by B. J. Zimmerman and D. H. Schunk. New York: Routledge, pp. 15–32.

Winslow, L.E. (1996). "Programming pedagogy – A psychological overview". In: *SIGCSE Bulletin* 28 (3), pp. 17–22.

Winterling, Vincent, Glen Dunlap, and Robert E. O'Neill (1987). "The influence of task variation on the aberrant behaviors of autistic students". In: *Education and Treatment of Children* 10 (2), pp. 105–119.

Wolfe, Joanna (2004). "Why the rhetoric of CS programming assignments matter". In: *Computer Science Education* 14 (2), pp. 147–163.

Wood, D., J.S. Bruner, and R. Gail (1976). "The role of tutoring in problem solving". In: *The Journal of Child Psychology and Psychiatry* 17 (2), pp. 89–100.

Yang, Ya-Ting C., Timothy J. Newby, and Robert L. Bill (2005). "Using Socratic questioning to promote critical thinking skills through asynchronous discussion forums in distance learning environments". In: *The American Journal of Distance Education* 19 (3), pp. 163–181.

Yoo, J., C. Pettey, S. Yoo, J. Hankins, C. Li, and S. Seo (2006). "Intelligent tutoring system for CS-I and II laboratory". In: *The Annual ACM Southeast Conference*, pp. 146–151.

Zimmerman, B. (1989). "A social cognitive view of self-regulated academic learning". In: *Journal of Educational Psychology* 81 (3), pp. 329–339.

Zimmerman, B. (2000). "Attaining self-regulation: A social cognitive perspective". In: *Handbook of Self-Regulation*. Ed. by M. Boekaets, M. Zeidner, and P.R. Pintrich. London, UK: Elsevier Academic Press, pp. 13–39.

Zimmerman, Barry J. and Manuel Martinez Pons (1986). "Development of a structured interview for assessing student use of self-regulated learning strategies". In: *American Educational Research Journal* 23 (4), pp. 614–628.

Zohar, Anat and Adi Ben David (2008). "Explicit teaching of meta-strategic knowledge in authentic classroom situations". In: *Metacognition and Learning* 3 (1), pp. 59–82.