12-2016

Students' explanations in complex learning of disciplinary programming

Camilo Vieira
Purdue University

STUDENTS' EXPLANATIONS IN COMPLEX LEARNING OF DISCIPLINARY PROGRAMMING

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Camilo Vieira

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2016

Purdue University

West Lafayette, Indiana

Graduate School Form
30 Updated 12/26/2015

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Camilo Vieira Mejia

Entitled
STUDENTS' EXPLANATIONS IN COMPLEX LEARNING OF DISCIPLINARY PROGRAMMING

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Alejandra J. Magana
Chair

Marisa E. Exter

R. Edwin Garcia

Senay Purzer

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): Alejandra J. Magana

Approved by: Kathryne A. Newton 12/6/2016

Head of the Departmental Graduate Program Date

Dedicated to my parents and to Charito.

ACKNOWLEDGMENTS

I would like to acknowledge my graduate committee for their contributions throughout the whole process, Dr. Magana for supporting me and believing in me, and the National Science Foundation for sponsoring this project under the award number EEC# 1329262.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABBREVIATIONS

CLT	Cognitive Load Theory
THERMO	Thermodynamics
CPMSE	Computation and Programming for Materials Scientists and Engineers
CK	Declarative Knowledge
PK	Procedural Knowledge
SK	Schematic Knowledge
TK	Strategic Knowledge
LK	Limited Knowledge

GLOSSARY

Intrinsic CL	Cognitive load that is inherent to the complexity of the learning material. It is given by the interaction of different elements. Everything that is supposed to be learned is an element. Learning materials are understood when all the elements and their interactions have been processed (Sweller & Chandler, 1994)
Extraneous CL	Cognitive load that is not beneficial to learning. This load is given by the way the information is presented. Hence, it can be modified by an appropriate or inappropriate instructional design (Sweller et al., 2011)
Worked-Example effect	The learners study worked examples before engaging in problem solving activities. The worked example effect focuses learners attention on the problem states (i.e. the different steps that are taken until a problem is solved) and prevents them from using a means-ends-analysis. The worked examples provide the learners with an expert solution to the problem (Van Merrienboer & Sweller, 2005)
Self-explanation effect	When learners engage in a self-explanation process of the examples to take an additional advantage from them (M. Chi et al., 1989)

ABSTRACT

Vieira, Camilo Ph.D., Purdue University, December 2016. Students' Explanations in Complex Learning of Disciplinary Programming. Major Professor: Alejandra J. Magana.

Computational Science and Engineering (CSE) has been denominated as the third pillar of science and as a set of important skills to solve the problems of a global society. Along with the theoretical and the experimental approaches, computation offers a third alternative to solve complex problems that require processing large amounts of data, or representing complex phenomena that are not easy to experiment with. Despite the relevance of CSE, current professionals and scientists are not well prepared to take advantage of this set of tools and methods. Computation is usually taught in an isolated way from engineering disciplines, and therefore, engineers do not know how to exploit CSE affordances.

This dissertation intends to introduce computational tools and methods contextualized within the Materials Science and Engineering curriculum. Considering that learning how to program is a complex task, the dissertation explores effective pedagogical practices that can support student disciplinary and computational learning. Two case studies will be evaluated to identify the characteristics of effective worked examples in the context of CSE. Specifically, this dissertation explores students explanations of these worked examples in two engineering courses with different levels of transparency: a programming course in materials science and engineering glass box and a thermodynamics course involving computational representations black box.

Results from this study suggest that students benefit in different ways from writing in-code comments. These benefits include but are not limited to: connecting individual lines of code to the overall problem, getting familiar with the syntax, learning effective algorithm design strategies, and connecting computation with their discipline. Students in the glass box context generate higher quality explanations than students in the black box context. These explanations are related to students prior experiences. Specifically, students with low ability to do programming engage in a more thorough explanation process than students with high ability. This dissertation concludes proposing an adaptation to the instructional principles of worked-examples for the context of CSE education.

CHAPTER 1. INTRODUCTION

Computational science and engineering (CSE) focuses on complex problem solving using a multidisciplinary approach including science, computer science, engineering and mathematics (EDUCATION et al., 2011). CSE's capacity to solve complex problems in several sectors has led some scholars to denominate CSE - along with theoretical and physical experimentation - as "the third pillar of science." (PITAC, 2005). This reflects CSE's vital contribution to the United States' scientific, economic, social, and national security goals.

National agencies such as National Science Foundation (NSF), the President's Information Technology Advisory Committee (PITAC), and the World Technology Evaluation Center, Inc. (WTEC), among others, have stressed the importance of introducing CSE as part of the engineering curricula. They argue that, by integrating CSE into the engineering curricula, professionals and scientists will be better prepared with the computational and disciplinary skills necessary to face increasingly complex problems in an evolving world (Chesnais, 2012; Glotzer et al., 2009; NSF, 2011; PITAC, 2005).

However, current undergraduate curriculum strategies frequently use computation as a restricted technical tool applied in an isolated way for the fundamentals in engineering (PITAC, 2005). This means that engineering students have disciplinary training and also computational training, but they do not have training on how to use computing in their discipline. One of the models that has been proposed to increase the exposure of students to computational concepts is to incrementally use small add-on courses to supplement conventional courses in mathematics, engineering, and science (Turner et al., 2002).

These curricular innovations involve at least two different ways in which users interact with computation: black box and glass box (Mayer, 1981). The black box approach uses computational tools without providing access to the underlying mechanisms. Meanwhile, with the glass box approach the learners have access to see, and often modify, these mechanisms. Students exposed to the black box approach often mention that they would prefer to have access to the underlying mechanisms. However, the glass box approach involves more complex learning material, which often overwhelms students Magana, Brophy, and Bodner (2010, 2012). These conflicting concerns related to the level of transparency provided to students is known as the transparency paradox. For instance, in preliminary work, initial offerings of a computational materials science and engineering course showed that, although the experience was relevant for the students, and increased their perceived ability to create and use computation, students found the content of the course to be time consuming with a high workload, and very challenging, in part because they not only had to learn programming but also mathematical modeling (Magana, Falk, & Reese Jr, 2013). Programming is a complex skill to learn (Mselle & Twaakyondo, 2012), since it involves many interacting elements (e.g., syntax, programming logic, the problem, how a computer works) to be considered at once. We hypothesize that glass box courses are even more complex, since they involve understanding the disciplinary and mathematical concepts that are being transformed into a computer program (Magana, Falk, & Reese Jr, 2013).

This project focuses on integrating computational tools and methods into engineering disciplinary courses. However, to successfully do that, the challenges regarding the complexity of programming courses must be addressed. Cognitive Load Theory (CLT) can support this form of complex learning (Van Merrienboer & Sweller, 2005). CLT establishes a cognitive architecture and a cognitive process to understand how learning occurs (Sweller et al., 2011). Using these components, CLT identifies instructional design guidelines considering the complexity of the learning material and the learner. One of the pedagogical strategies suggested by the CLT is called worked examples.

A worked example comprises a problem statement and an expert solution to the problem (Atkinson et al., 2000). This strategy can be useful for novice learners under certain conditions. One of those conditions is to engage the learners in an active exploration of the examples by a self-explanation process. However, this process has not been studied in the context of computational science and engineering within naturalistic settings.

1.1 Scope

The research process took place during different engineering courses between Fall 2013 and Spring 2016. One pilot study and two case studies were investigated aiming to contribute to the identification of effective pedagogical practices in computational science and engineering courses. First, a pilot case study explored an introductory programming course in the Purdue Polytechnic Institute (former College of Technology at Purdue University). The purpose of this pilot study was to examine the characteristics of effective programming worked examples and whether writing comments within the code can be an effective self-explanation approach. Thirty five students participated on this study during Fall 2013. Results from this pilot study are presented in Vieira, Yan, and Magana (2015)

During the spring semesters in 2015 and 2016, two different courses were investigated. First, a course at Johns Hopkins University called "Computation and Programming for Materials Scientists and Engineers" (CPMSE) was explored. Approximately 25 students enroll in this course that is offered every spring semester. The other course that was explored it "Thermodynamics of Materials" (THERMO) at Purdue University, which included four computational sciences and engineering modules. Approximately 60 sophomore Materials Engineering (ME) students enroll in this course every spring semester a first-year Materials Science and Engineering (MSE). The difference between these two courses is that THERMO is a disciplinary course that includes computational tools and methods to support student disciplinary learning (black box approach). On the other hand, CPMSE aims to introduce applied algorithmic thinking in the context of MSE problems (Magana, et. al. 2015): glass box. Two rounds of data collection took place for each of these cases, one during Spring 2015 and the other one during Spring 2016. Figure 1.1 summarizes the three case studies that were investigated for the purpose of this dissertation.

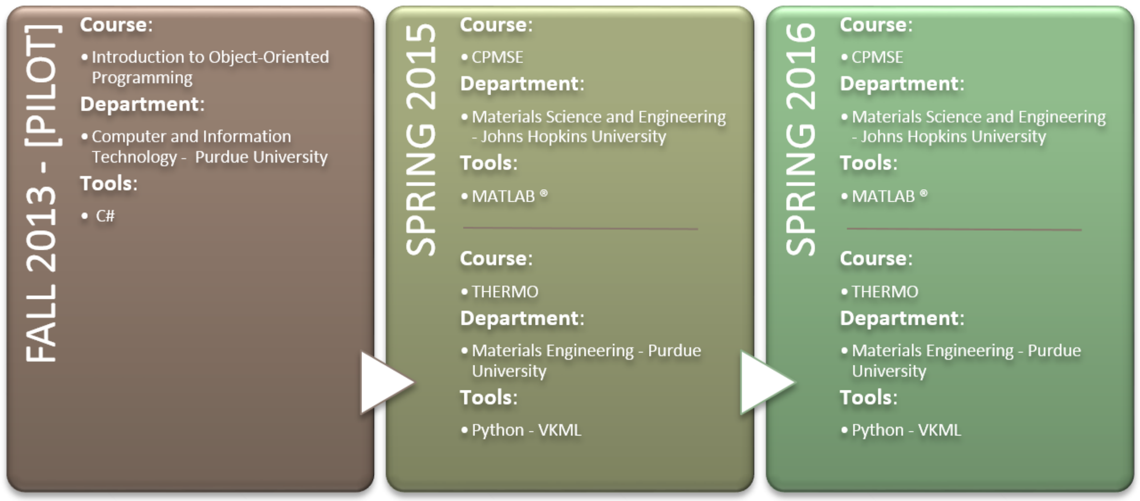

FALL 2013 - [PILOT]
Course:
• Introduction to Object-Oriented Programming
Department:
• Computer and Information Technology - Purdue University
Tools:
• C#
SPRING 2015
Course:
• CPMSE
Department:
• Materials Science and Engineering - Johns Hopkins University
Tools:
• MATLAB ®
Course:
• THERMO
Department:
• Materials Engineering - Purdue University
Tools:
• Python - VKML
SPRING 2016
Course:
• CPMSE
Department:
• Materials Science and Engineering - Johns Hopkins University
Tools:
• MATLAB ®
Course:
• THERMO
Department:
• Materials Engineering - Purdue University
Tools:
• Python - VKML

Figure 1.1. Timeline of the investigated courses

For the context of this study, we classified the two different contexts as glass box (CPMSE) and black box (THERMO). Computing education can be approached using different levels of transparency regarding the underlying mechanisms of the program (Mayer, 1981). In a glass box approach, students access to the actual code to manipulate the mathematical models that determine the behavior of the simulation. A black box approach to CSE education enables students to perform advanced experiments, but limits their understanding about the underlying mechanisms of the scientific phenomena (Resnick, Berg, & Eisenberg, 2000). However, although students often prefer to have access to these mechanisms, this level of transparency increases the complexity of the learning materials, which can overwhelm and frustrate students. This phenomenon has been denominated as the transparency paradox (Magana et al., 2010).

Although results can be expanded to other materials science and engineering (MSE) courses and engineering disciplines, additional evaluation processes are required to adapt the learning materials and procedures. The curricular intervention included the worked examples and the computational modules introduced in these courses. The worked examples involved the acquisition and application of MATLAB® and Python programming skills to solve disciplinary problems. Hence, the characteristics of students' explanations described as result of this inquiry process may only apply to these specific contexts.

1.2 Significance

Several studies have established that there is a lack of well-prepared engineers and scientists in the United States with the disciplinary and computational skills needed to approach global grand challenges (PITAC, 2005). As shown by one of the findings presented in the International Assessment of Research and Development in Simulation-Based Engineering and Science, this is a challenge and an impediment for this country to overcome. The finding states:

> "Continued progress and U.S. leadership in SBE&S (Simulation-Based Engineering and Science) and the disciplines it supports are at great risk due to a profound and growing scarcity of appropriately trained students with the knowledge and skills needed to be the next generation of SBE&S innovators. (Glotzer et al., 2009) "

The U.S. has a decreasing number of professionals in Computational Sciences compared with the European Union and Asia and there is a need for computing science education at all the levels (Chesnais, 2012; Glotzer et al., 2009; PITAC, 2005). The percentage of U.S. undergraduate students with a science and engineering degree is comparatively low: South Korea, 38%; France, 47%; China, 50%; Singapore, 67%; United States, 15% (Glotzer et al., 2009). This means the U.S. should not only increase its number of professionals in sciences and engineering but also existing students should be trained with knowledge and skills to compete and maintain leadership in innovation. This training and exposure to CSE skills usually come very late during the engineering career (i.e., during the graduate studies) (Magana & Mathur, 2012)

Despite this shortfall, current undergraduate curriculum strategies are frequently designed to prepare the next generation of engineers to use computation as a restricted technical tool applied in an isolated way for the fundamentals in engineering (PITAC, 2005). This proposal is focused upon the first stages of addressing this problem - by examining how we can better design curricula and pedagogies to prepare the next generation of scientists and engineers to use computation to compete in a global and continually evolving society. The significance and broader impacts for this project involve not only the identification of adequate approaches to prepare the next generation of MSE professionals, but in several engineering disciplines. By understanding the way learning occurs in integrated computational-disciplinary learning courses, new possibilities for better designing engineering curricula emerged. The use of such an innovative pedagogical practice along the research process offers the description and dissemination of new and better approaches to train new engineers with the disciplinary and computational skills required by the U.S to be a competitive nation. In addition, this approach can demonstrate not only an increased intention of students to continue participating in computing courses but an increased participation for female and minority students. Hence, although the project is designed to be applied in MSE, the practices and some of the modules can be applied to many other engineering disciplines having a direct impact on engineering curricula.

1.3 Research Question

The goal of this dissertation is to understand *students' explanations of worked examples for an integrated disciplinary-programming complex task*. To do so, the researcher explores the following derived research questions:

RQ1. What are affordances of in-code commenting self-explanation activities in the context of black box and glass box approaches to computational science and engineering?

RQ2. : What are the characteristics of students' explanations in a glass box and a black box approach to CSE education?

RQ3. How do the characteristics of students' self-explanations in glass box and black box approaches to CSE education relate to their ability to program?

1.4 Assumptions

The following assumptions are inherent to this study:

- Participants in this study answered honestly to the assessment instruments.
- Participants enrolled in the explored courses may have had previous computer programming experience.
- Participants enrolled in the thermodynamics course have previous knowledge in chemistry, calculus and computer programming.
- Participants completed the laboratory assignments including the examples self-explanations, pretest and posttest based on what they know.

1.5 Limitations

The limitations for this study include:

- This study takes place at a naturalistic context. Thus, students participating on this study may be influenced by elements of this course and other courses that were not part of the study.
- Students participating in this study were enrolled in specific courses of the MSE curricula at Purdue University and Johns Hopkins University. Thus, inferences from the study are applicable to student with similar characteristics. However, in order to generalize the findings follow-up studies should be carried out.
- The number of participants was given by the number of students enrolled in the courses that were explored. During the semester, as the students were able to drop the course, the study was vulnerable to incomplete procedures by some of the participants
- Students previous programming knowledge may affect their performance and responses to the study. Since there is no standardized computing curriculum for K12, it was expected that students would arrive with a broad range of experiences in this field.
- Some of the activities involved in this study correspond to homework assignments, and therefore the order in which students do a sequence of activities students (e.g. writing self-explanations and completing the performance tests). As a consequence, the conclusions relating students explanations and performance are only exploratory, and future experimental research is needed to validate them.

1.6 Delimitations

The delimitations inherent to this study are the following:

- This study is developed from a Cognitive Load Theory perspective. This means that learning is narrowed down to the cognitivist perspective. The author recognizes that there are other factors such as collaboration or motivation that might affect the learning process and student performance. These were not taken into account in the scope of the study.

- There might be different effective pedagogical approaches to support complex learning. This study did not compare the worked examples approach to other approaches nor to a control group. The aim of the study is not to evaluate the effectiveness of the worked examples approach. The purpose is to understand how students create and use their own explanations of worked examples, and what factors influenced them.

1.7 Summary

This chapter described the context of the study and the problem this dissertation is addressing. The next chapter provides a review of the literature in computational science and engineering education.

CHAPTER 2. LITERATURE REVIEW

2.1 Computational science and engineering (CSE)

Computational science and engineering (CSE) focuses on complex problem solving using a multidisciplinary approach including science, computer science, engineering and mathematics (EDUCATION et al., 2011). CSE's capacity to solve complex problems in several sectors has led it to be denominated - along with theoretical and physical experimentation - as "the third pillar of scientific inquiry" (PITAC, 2005, p. 1). National agencies such as National Science Foundation (NSF), the President's Information Technology Advisory Committee (PITAC), or the World Technology Evaluation Center, Inc. (WTEC), have stressed the importance of introducing CSE as part of the engineering curricula in order to enable future professionals and scientists to face the increasing complex problems in an evolving world (Chesnais, 2012; Glotzer et al., 2009; NSF, 2011; PITAC, 2005).

An example of this phenomenon is the establishment of a Materials Science and Engineering (MSE) sub-discipline: "computational materials science and engineering (CMSE)" (Magana, Falk, & Reese Jr, 2013). The sub-discipline emerged as a response to the need of computational tools to solve complex problems, simulate and predict materials' responses, and increase reliability using computer experiments (Hafner, 2000; Magana, Falk, & Reese Jr, 2013) . Another example is the introduction of computational science as one of the computer science disciplines according to the computer science curricula 2013 (Joint Task Force on Computing Curricula & Society, 2013; Sahami, Roach, Cuadros-Vargas, & LeBlanc, 2013). The computational science discipline comprises six topics: (1) Introduction to modeling and simulation, (2) Modeling and simulation; (3)

Processing; (4) Interactive visualization; (5) Data, information and knowledge; and (6) Numerical analysis.

In spite of these initial efforts, current undergraduate curriculum strategies frequently use computation as a restricted technical tool applied in an isolated way for the fundamentals in engineering (PITAC, 2005). One of the possible reasons for this phenomenon is that the responsibility of integrating CSE into disciplinary courses falls between the computing science department and the disciplinary departments, and none of them assume responsibility for it (NSF, 2011). The consequences are that the students do not learn how to apply computational practices to a real-world problem within their discipline (NSF, 2011). Hence, students might have the disciplinary training, and also computational training, but they may not have training on how to use computing in their discipline.

2.2 Learning Programming

Programming is a complex and hard skill to learn (Du Boulay, 1986; Mselle & Twaakyondo, 2012; Rogalski & Samurçay, 1990). Programming involves so many interacting elements to be learned at once, that overwhelms the cognitive capacity of novice programmers (Sweller et al., 2011). The purpose of the program, the language syntax and semantics, the programming logic, and new abstract data structures are some of the interacting elements that need to be considered (Du Boulay, 1986).

Programming itself can be seen as putting together different instructions that will solve a problem. Consequently, programming can be seen as a design process, which does not have a unique solution (i.e., a computer program), but instead has multiple solutions to achieve the end goal: performing a given task (Confrey, 1990; Soloway, 1986). In this context, programming is also an iterative process, where the programmer tries and refines a potential solution multiple times until the desired product (the execution of a given task) is reached. At the end, the programmer needs to understand and be able to explain why all the program parts together provide an accurate solution (Soloway, 1986).

A large multi-institutional project found that novice programmers tend not to have problems with individual programming instructions but with groups of instructions that have a given purpose (Lister, 2011; Lister et al., 2012; Whalley & Lister, 2009). These studies identified three levels of expertise in novice programmers (Lister, 2011). First, students are able to trace values in a programming code without understanding its whole purpose. Second, students are able to understand and describe the overall purpose of a program, but are unable to use it in a different context. The third level involves students able to make abstractions of the programming code, understanding its purpose, and use it in the appropriate contexts.

Novice programming learners focus on what each line of code does (Mselle & Twaakyondo, 2012), while experts identify an abstract explanation of the overall purpose of the code (Whalley & Lister, 2009). Expert programmers do not only know language syntax or semantics, nor do they only understand individual lines of code, but they also know solutions to common problems that can be used in different contexts (Soloway, 1986). As in the experiments about expertise with chess players (Bransford, Brown, & Cocking, 2000), expert programmers identify existing solutions and strategies, and apply these chunks of code to solve other problems (Mayer, 1981; Soloway, 1986). Consequently, novice programmers face more problems with transfer tasks that require from them to apply acquired knowledge to different contexts, than with understanding tasks such as tracing values (Whalley & Lister, 2009).

2.2.1 Misconceptions in programming

Students' misconceptions in programming have been studied for a long time. Several researchers argue that one of the main sources of programming misconceptions is the novices' lack of knowledge about the underlying mechanisms of the machines and the programming languages (Mselle & Twaakyondo, 2012). Many misconceptions are related to things that happen during execution-time and are not visible when writing a program (e.g., memory allocation) ("Difficulties in Learning and Teaching Programming Views of Students and Tutors", 2002; Sorva, 2013). Common misconceptions in programming deriving from the student lack of mental models about the computer mechanisms comprise student understanding of (Bayman & Mayer, 1983): (1) input-output commands, how the machine stores inputted values in memory, or where the data comes from; (2) transitions from one line to other one that is not the next one in the sequence of the program; (3) the equal sign, considered as an equation instead of an assignment; (4) the name of a variable compared to the value contained in that variable.

A second source has been denominated the 'superbug', which describes the fact that novice programmers tend to assume that the computer 'understands' human language beyond its capacity, and therefore, can infer instructions that are not explicit in the code (Kaczmarczyk, Petrick, East, & Herman, 2010; Pea, 1986; Pea, Soloway, & Spohrer, 1987). Pea 1986, for example, described three types of student misconceptions in programming known as language-independent bugs that fall under the umbrella of the superbug. First, students incorrectly assume that the computer understands human language, and therefore provide imprecise and limited instructions through the program. Second, learners misinterpret the order in which the instructions are executed. Novice programmers may think that all instructions are executed at the same time instead of sequentially. For instance, if the value of a variable is modified at certain point of the code, novices would expect that instructions using this variable at different points (even previous points) will be immediately affected by that change. Third, assuming that a computer program has its own intention to do something. In this case, students first see the code to identify the goal of the program, and then assume that the computer wants to do this and knows it at each instruction.

Pea et al. (1987) extended these three language-independent bugs to include students' misconceptions related to the use of metacognitive strategies and the language-dependent bugs. The language-dependent bugs relate to specificities of the language such as syntax, semantics, or memory management, which the students misinterpreted on the first place, or things that they know but are unable to apply it in a different context (i.e., transfer). Students' use of metacognitive strategies such as monitoring their own learning are very limited, and they often skip lines code and do not validate their work. Although these monitoring activities are usually overlooked, they can be promoted through self-explanations (Williams & Lombrozo, 2010) .

A third source of misconceptions relates to the fact that students make assumptions about whether certain elements can be used in other contexts or not (Fleury, 2000). For example, students consider the dot operator only applicable to invoke methods, although it can also be used for other purposes. Another example is that students often incorrectly use the abstraction of a loop to describe how a recursive method works (Sorva, 2013). In general, this source of misconceptions is related to the use of correct knowledge that is incorrectly applied in a broader domain, as students perform systematic errors by applying incorrect rules resulting in common patterns of mistakes (Confrey, 1990).

Several attempts have been made to create concept inventories that would enable educators and researchers to assess conceptual knowledge in programming (Taylor et al., 2014; Tew & Guzdial, 2011). One of the big challenges to successfully complete this task is that computer science is a dynamic field that not only changes technologies (e.g., computer architectures and programming languages) but also paradigms (e.g. functional programming to object oriented programming). All these changes involve different concepts that are usually challenging for students to learn. Nonetheless, these attempts provide useful information regarding students' common misconceptions within a given context. For example, Kaczmarczyk et al. (2010) presented the preliminary findings of a study to design a concept inventory for computer programming. Four themes emerged from their qualitative analysis on what students struggle with: (1) language properties and memory usage; (2) misunderstanding of while-loop; (3) lack of understanding of objects from the object oriented programming paradigm; and (4) inability to trace code linearly. In addition to these themes, it has also been identified that students usually believe that a variable can store more than one value at a given time, which is related to the fact that arrays are a complex concept to learn for beginner programmers (Taylor et al., 2014).

Similarly, Goldman et al. (2008, 2010) employed a Delphi study to identify the difficult and important topics to learn in computing courses. Specifically, they explored the courses of discrete mathematics, introductory programming, and logic. The most difficult and important topics among the ones identified for programming fundamentals are: (1) scope of the parameters; (2) design of procedures; (3) local vs. global variables; (4) inheritance; (5) pattern recognition and use; (6) recursion; (7) memory model; (8) decomposition of the problem in different functions; (9) design of a solution for a given problem; (10) debugging; and (11) test design. Table 2.1 summarizes the misconceptions and difficult concepts that were described in this section.

In the context of computational science, there are additional components that go beyond the programming skills. Additional misconceptions and difficulties can emerge due to student limited ability to make clear connections among the disciplinary phenomena, the mathematical models, and computational representations (Magana, Falk, & Reese Jr, 2013).

2.3 Supporting CSE Education

PITAC proposes to start addressing this gap with the design of individual courses including CSE concentrations. This would lead to developing computational concepts in students but also would involve and encourage faculty members to explore the capabilities of those concepts in their disciplines (PITAC, 2005). Also, several models have been proposed to increase the exposure of students to CSE such as (a) building programs from existing courses, creating concentrations in CSE; (b) introducing multidisciplinary team-taught project courses; (c) incrementally using small add-on courses to supplement conventional courses in mathematics, engineering, and science (Yaar, 2013); and (d) using a particular vehicle, such as computer graphics, to introduce key CSE ideas into regular courses (Turner et al., 2002).

Table 2.1.

Misconceptions and difficult concepts in computer programming

Source of misconceptions or difficult concept	**Misconception or Difficult Concept**
Lack of understanding underlying mechanisms	Input-output commands and memory management (Goldman et al., 2010; Kaczmarczyk et al., 2010; Pea et al., 1987)
	Non-linear sequence of program (Bayman & Mayer, 1983)
	Equal sign: equation vs. assignment (Bayman & Mayer, 1983)
	Name of a variable compared to the value in that variable.(Bayman & Mayer, 1983)
Superbug (Pea, 1986)	Assuming that the computer understands human language (Pea, 1986)
	The order in which the instructions are executed (Kaczmarczyk et al., 2010; Pea, 1986)
	Intentionality of the computer program (Pea, 1986)
	Language-dependent bugs: syntax and semantics (Pea et al., 1987)
	Lack of meta-cognitive strategies (monitoring learning) (Pea et al., 1987)
Systematic errors (Confrey, 1990)	Difficulty to identifying chunks of code with certain purpose (Mselle & Twaakyondo, 2012; Whalley & Lister, 2009)
	Use of correct knowledge that is incorrectly applied in a broader domain (Fleury, 2000)
Difficult and important concepts (relevant to this study)	Objects (Kaczmarczyk et al., 2010)
	Loops (Kaczmarczyk et al., 2010)
	Arrays (Taylor et al., 2014)
	Scope of Parameters (Goldman et al., 2010)
	Procedures (Goldman et al., 2010)
	Local and Global Variables (Goldman et al., 2010)

Magana, Vieira, Polo, Yan, and Sun (2013) surveyed engineering faculty to identify how engineering professors would integrate computation into disciplinary courses. Computation is used in disciplinary courses to support the solution of real world problems and facilitating complex calculations. Homework assignments, laboratory activities and projects were the preferred tasks to have students applying computation. However, there is no evidence in the study of professors providing scaffolding techniques for all these independent work activities (Magana et al., 2012). These complex learning activities that integrate computational tools into disciplinary contexts require pedagogical practices to avoid overwhelming students (Magana, Falk, & Reese Jr, 2013; Magana, Vieira, et al., 2013).

Similar areas may provide an insight into effective pedagogical approaches. However, the picture in areas such as computer science is not very encouraging either. Guzdial (2011) urged computing education researchers to develop better approaches to teach computer science. Although some reports say that there unemployment rate for technology-related jobs is raising, professionals are not well prepared for the needs of the job market. There is strong evidence that courses such as CS1 are not preparing students in the basics of programming and algorithm design (Guzdial, 2011). Moreover, a systematic literature review carried out by Radermacher and Walia (2013) listed programming and testing as two of the top ten knowledge deficiencies to meet employers' expectations.

Another example of the current status of educational practices on related to computing areas is presented by Exter (2014). Professionals in software design considered they are not well prepared in their formal education to be successful in their careers (Exter, 2014). The participants of a study comparing their work-force experiences to their educational experiences highlighted the importance of being exposed to real-world problem solving from the beginning of their undergraduate programs (Exter, 2014; Exter & Turnage, 2012). These problems should be several semester-long in order to simulate real world practices. The participants also discussed that learning a specific programming language is not very important. Instead, understanding programming logic and concepts provide students with relevant tools to understand other programming languages (Exter & Turnage, 2012).

2.3.1 Approaches for Complex Learning

Understanding what complex learning is and how this process can be supported, is an important step to effectively introduce CSE into the engineering curricula. Chapter 3 addresses these topics. Nevertheless, it is worth to mention some of the existing pedagogical approaches that have been evaluated to support CSE and computer science learning processes.

Congitive load theory (CLT) (Sweller et al., 2011) describes different pedagogical approaches to support complex learning. The goal free effect consists of allowing the students to interact with the available tools and get to different possible sub-goals. Thus, the students do not consider all the steps they would need to do in order to get to a specific final goal. The goal free effect has been evaluated in areas such as physics (C. S. Miller, Lehman, & Koedinger, 1999; Sweller, Mawer, & Ward, 1983; Wirth, Künsting, & Leutner, 2009), geometry (P. Ayres & Sweller, 1990; Sweller et al., 1983), and trigonometry (Owen & Sweller, 1985).

Other approaches described by the CLT are the worked example effect and the completion effect. The worked example effect refers to providing students with a problem statement, a step by step solution, and auxiliary representations. Hence, students explore an expert solution before engaging on problem solving on their own. The completion effect also uses a worked example but this time the example is only partially solved. The worked example effect has been evaluated in several areas such as: Computer Programming (Trafton & Reiser, 1994; Vieira et al., 2015); Mathematics (Carroll, 1994), Geometry (F. G. W. C. Paas & Van Merrinboer, 1994) and Physics (M. Chi et al., 1989) while he completion effect has been evaluated in domains such as modeling (Mulder, Bollen, de Jong, & Lazonder, 2016), Physics (Renkl, Atkinson, Maier, & Staley, 2002), Electrical circuits (Reisslein, Atkinson, Seeling, & Reisslein, 2006), Engineering (Moreno, Reisslein, & Ozogul, 2009), and Mathematics (Kalyuga & Sweller, 2004; Salden, Aleven, Schwonke, & Renkl, 2010; Schwonke, Renkl, Salden, & Aleven, 2011). An instance of the completion effect in supporting CSE was identified by Magana et al. (2012), where students highlighted the use of templates and blueprints as a useful scaffolding technique for implementing coding solutions. These effects and strategies are focused on supporting the design of specific instructional materials. However, it is not always clear how to integrate them in a classroom context (e.g., what is the role of the teacher) or when to use each approach (Van Merrienboer & Sluijsmans, 2009). 4C/ID is a model to effectively design educational programs for complex learning tasks (Van Merrienboer & Sluijsmans, 2009; Van Merriënboer, Clark, & De Croock, 2002). The model comprises four interrelated components that should be considered:

(1) *Learning tasks* that encourage the schema creation through the abstraction of the specific application of the task. Instructional designers should promote an inductive approach to the task to facilitate this creation process. These need to be presented as a progression of an increasingly task complexity (i.e., task classes) to avoid a cognitive overload in a novice student from the beginning (Van Merriënboer, Kirschner, & Kester, 2003). Within each "task class", the scaffolding provided to the learner should be gradually removed in order to develop students' expertise. For example, the first task can be supported with high level scaffolding such as worked examples; the second task will only need goal free or completion; and the last task may not have any explicit support.

Another example of these progressions was proposed by Lee and collaborators (2011). Use-modify-create is a three-stage progression used to introduce CSE concepts. The first activity intends to expose the student to use and become familiar with a computational tool or model. Once the student has acquired certain familiarity with the concepts, she/he starts making increasingly complex changes to the existing solution. Once the student has reached certain level of competence and confidence, she/he can be encouraged to create and refine new solutions for different ideas and needs.

(2) *Supportive information* that connects learners' existing schemata to what is required to complete the task. This information refers to the "theories" or the tools that are provided to the learner to enable her/him to complete a task. Part of this information comes from the progression of "task classes". Therefore, an earlier and less complex task will provide more supportive information than a more complex task. This supportive information should be presented using an "inductive-expository strategy" in which individual case-studies are provided to the learners so that they find relationships among them. Hence, this approach will empower the creation of abstract schemata in the long-term memory.

(3) *Just-in-time information* provided by the instructor. It includes the required rules, procedures and corrective feedback to enable the learner to complete the task. The delivery of this information is gradually faded since the learner only requires it at a very early stage, after which it becomes automatic. In a programming context, this information would include indications of how to create a new program or how to compile it and run it. This information can be delivered using demonstrations and instances in order to avoid memorizing activities for such information.

(4) *Part-task practice* for recurrent aspects of the learning task that require a high level of automaticity. The whole task learning process might not be enough practice to achieve the required automaticity level. Some examples of these aspects can be the multiplication tables or the scales in musical instruments. It is suggested that these part-task practices are only introduced after an initial exposure to the complex task context. Thus, the learner will have a complete view of how this part-task will support the complex learning.

2.4 Summary

CSE is a set of important skills and tools for a competitive global society. CSE uses the power of computation to solve complex problems in several disciplinary areas, which would not be possible to approach by using theoretical or experimental techniques. Nevertheless, current professionals are not well prepared to advance in their fields using computational methods and tools. The current engineering curricula include computation courses in an isolated way from disciplinary courses. Hence, students do not learn how to apply their computation knowledge to solve disciplinary problems.

Some initial approaches have been suggested to fill this gap in undergraduate education. Among others, the creation of CSE concentrations, the modification of existing courses to include computational modules, and the use tools such as computational visualization to support student disciplinary learning, are some alternatives.

Several pedagogical strategies for computer science and CSE education have been explored in this chapter. However, introducing computational concepts such as programming within a disciplinary course may add an additional layer of complexity to the learning process. Thus, this study explores the characteristics of worked examples as a pedagogical approach that could reduce the cognitive load on students while learning CSE concepts.

The next chapter describes what complex learning is, techniques to reduce the cognitive load, how can complex learning be pedagogically supported, and what are the considerations to implement these scaffolding strategies.

CHAPTER 3. THEORETICAL FRAMEWORK

3.1 Complex Learning

Complex learning refers to the acquisition, understanding, integration and coordination of concepts, procedures, and skills in order for these to be applicable to real-life experiences(Kester, Paas, & van Merriënboer, 2010; Van Merriënboer et al., 2003; van Merriënboer, Clark, & Croock, 2002). Although approaches such as project-based education or problem-based learning intend to provide the learner with these experiences, the task complexity can be such that learners are overwhelmed with the learning activity (Merril, 2002; Van Merrienboer & Sweller, 2005).

The complexity of these tasks lies on the high level of interactivity among the different elements that compose them (Sweller et al., 2011; Sweller, van Merriënboer, & Paas, 1998). In order to understand how the element interactivity affects a learning task, and how we can effectively help students overcome such complexity, it is necessary to examine the Cognitive Load Theory (Van Merrienboer & Sweller, 2005).

3.2 Cognitive Load Theory

The Cognitive Load Theory (CLT) intends to "explain the relationship between the human cognitive architecture, instructional design, and learning" (Moreno & Park, 2010a, p. 20). CLT uses the complexity of the information to be learned and the way humans process it to guide effective instructional design practices (Van Merrienboer & Sweller, 2005).

The main component of CLT is the cognitive architecture, which describes the way information is processed by humans. The cognitive architecture comprises a limited working memory and a vast long-term memory (F. Paas, Renkl, & Sweller, 2003). The first one is limited in size and time. In adults, depending on how it is measured (i.e., storing individual chunks, compound chunks, or processing chunks), the working memory is able to manage between three to five chunks of information (Cowan, 2001, 2010) or seven, plus or minus two (G. A. Miller, 1956). This variation is not important from an instructional design perspective, because what is relevant is the fact that it is limited (Sweller et al., 2011). The time constraint refers to the need of either processing or rehearsing the information we are working with, if we do not want to forget it. In this regard, there is a range of possible values going from two to 20 seconds (Cowan, 2001; Van Merrienboer & Sweller, 2005).

The long term memory stores information as schemata that can be retrieved by the working memory when needed. A schema is a conceptual structure that groups different elements as a unit, based on how these can be represented or used (Bransford et al., 2000; M. T. Chi, Glaser, & Rees, 1981). The schemata allow us to solve problems using approaches we know are effective. Thus, it is important to acquire and automate schemata in order to recognize problem types and actions to be taken in a particular situation (Sweller et al., 2011). The size and time constraints in the working memory only apply to information from the environment and not to information from the long-term memory (Sweller et al., 2011). In this case, the schemata can be organized such that they do not overload the working memory. In fact, Ericsson and Kintsch (1995) coined the term "long-term working memory" to suggest that there are two different working memories, one for the environmental information and another for the long-term memory schemata.

Information is processed through several stages within the cognitive architecture as depicted in Figure 3.1 (Wickens et al., 2015). Once the senses process the information from the environment, the perception process filters it and gives it a meaning based on our attention and our previous knowledge. Hence, not all the information processed by our senses is actually perceived. Instead, we select what to focus on, and we use prearranged schemata stored in the long term memory to give a meaning to the information. From the perception stage, it can go to either/both the working memory stage and/or the response selection stage. Reaction to an external stimulus will trigger the information to go directly to the response selection stage. However, whenever learning is going to happen, it must go through the working memory stage. Another way to look at the perception and working memory stages is as a single cognition stage (Wickens et al., 2015).

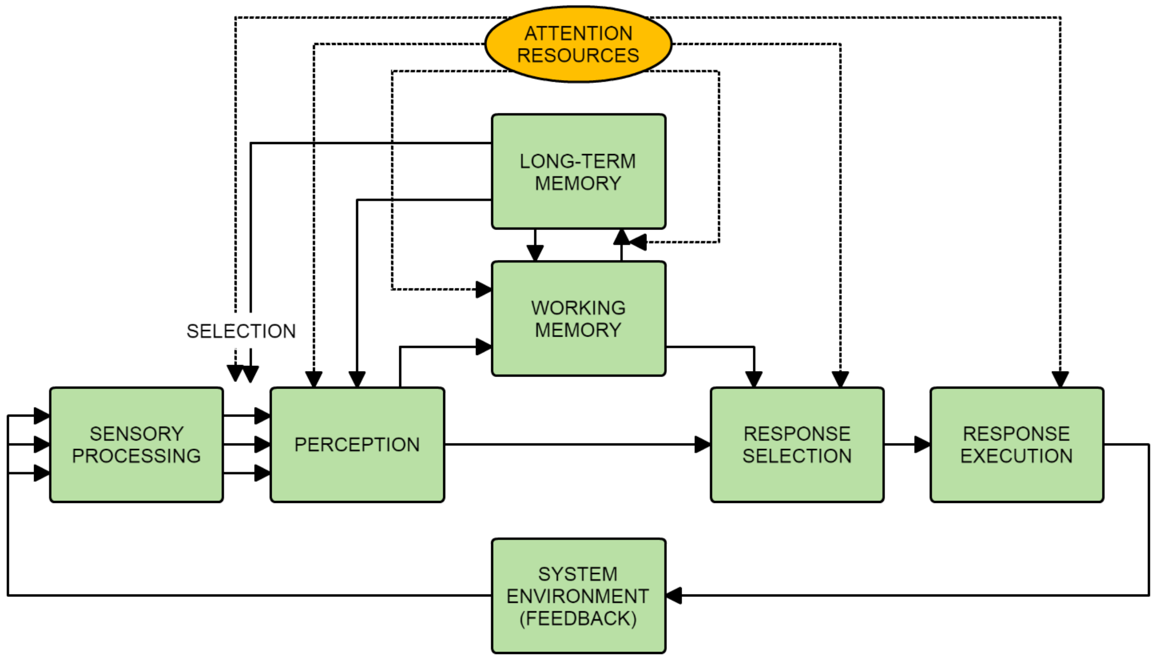


Figure 3.1. A model of human information processing stages (Wickens et al., 2015)

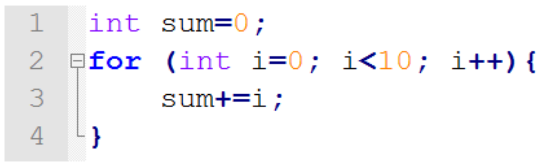
When learning occurs, the information in the working memory is transformed into a schema that is stored and automated into the long-term memory (F. Paas, Tuovinen, Tabbers, & Van Gerven, 2003). First, when the information gets to the working memory, it devotes its resources to reflect on it, bringing additional information (preexisting schemata) from the long term memory. The attention resources are also connected to the working memory to alert for any changes on the environment. At this point, the information becomes a schema that is going to be stored in the long-term memory. The automation process does not come to our mind right after studying a material. First the schema is acquired and, after some period of practice, the schema is automatized so that it can be invoked without a conscious action (Sweller et al., 2011).

3.3 Cognitive Loads

During a learning task, the student is exposed to different types of cognitive load in the working memory: intrinsic, extraneous, and germane load. The intrinsic load refers to the inherent complexity of the task or concepts to be learned (F. Paas, Renkl, & Sweller, 2003; F. Paas, Tuovinen, et al., 2003; Sweller et al., 2011). The only way to reduce this load would be by changing or reducing the concepts under study. The extraneous load is given by the way the information is presented. Hence, it can be modified by an appropriate or inappropriate instructional design (Sweller et al., 2011). The germane resources, also called germane load, are those that are beneficial to the learning process and therefore are oriented towards the intrinsic load. Germane resources are not given by the learning activity but by the working memory (Moreno & Park, 2010a). There are also extraneous resources that are used to deal with the extraneous load. The intrinsic load and the extraneous load are handled by the germane resources and extraneous resources. When the required germane and extraneous resources exceed the capacity of the working memory, a cognitive overload takes place.

The intrinsic cognitive load is given by the element's interactivity (Sweller et al., 2011). Everything that is supposed to be learned is an element. Learning materials are understood when all the elements and their interactions have been processed (Sweller et al., 2011; Sweller & Chandler, 1994). If the individual elements, in certain learning materials can be learned independently from each other, they have low interactivity and therefore low intrinsic cognitive load. The element interactivity can be increased by the instructional materials. A given instructional design that involves many different elements, which otherwise are not required to be processed simultaneously, will modify the element interactivity and the extraneous cognitive load. However, a material with very low element interactivity that is learned using an inadequate instructional design might not interfere with learning. The intrinsic load may be too small that the extraneous load does not exceed the working memory resource constraints.

An example for low element interactivity is learning new vocabulary (Sweller et al., 2011). A person can study and practice independent words as opposed to learning them all together. Learning a small set of words has a low intrinsic load. On the other hand, programming concepts such as loops are an example of high element interactivity. Consider a single loop that adds up all the numbers from one to nine as depicted in Figure 3.2. Besides the need of the learner to understand the goal of the program and the syntax and semantics of the programming language, there are several elements that interact within this for-loop clause. Consider line number 2: the variable i is initialized to zero; then, an upper limit has been set to 10; finally a one-by-one increment is established for variable i. The learner also needs to be aware that the for clause means 'iteratively repeat what is between the brackets'. She/he needs to process it all together to actually understand the loop concept and the function of the program.



```
1  int sum=0;
2  for (int i=0; i<10; i++){
3      sum+=i;
4  }
```

Figure 3.2. A model of human information processing stages (Wickens et al., 2015)

A distinct concept to element interactivity, regarding learning materials, is its difficulty. Some concepts can be difficult and have low element interactivity. In the vocabulary example, it can be difficult to learn a whole new set of vocabulary because it involves many individual words. Furthermore, learning materials with a small number of elements, but high element interactivity, can be also considered difficult concepts.

Different forms of learning can be used for different kinds of materials. While learning by rote can be useful for low element interactivity concepts, high element interactivity concepts would benefit more from a 'learning with understanding' approach (Sweller et al., 2011, p. 62). As mentioned before, the understanding is achieved when all the elements and their interactions are processed.

3.4 Measuring Cognitive Loads

Different approaches to measure cognitive load have been studied for several years (for a review see Ch. 6 Sweller et al. (2011)). Aside from the early attempts to measure it, this document will only describe the most used during the last few years: subjective measures, secondary tasks and physical measures. The subjective measures use learner reflections to assess the mental effort required to complete a task and the difficulty of the task.

Mental effort refers "to the cognitive capacity that is actually allocated to accommodate the demands imposed by the task" (F. Paas, Tuovinen, et al., 2003, p. 64). The subjective measure of mental effort assumes that a learner is able to assess how much mental effort he/she had to invest to complete a learning task. This assessment is carried out after the learning task using a nine-level Likert scale instrument from very, very low mental effort to very, very high mental effort. First presented by F. G. Paas (1992), there is evidence that this approach can objectively measure the cognitive load of a learning task (F. Paas, Tuovinen, et al., 2003).

(F. G. W. C. Paas & Van Merrinboer, 1994) propose a model to represent the cognitive load, which is comprised by causal factors and assessment factors. The causal factors include the learner, the task and the relation between them. Each of these elements has characteristics that interact with each other. The task characteristics include the format, the novelty, the complexity and the time pressure. The learner characteristics are the expertise, the age and the preferences (F. Paas, Tuovinen, et al., 2003; F. G. W. C. Paas & Van Merrinboer, 1994).

The assessment factors include the mental load, the mental effort and the performance. The mental load is inherent to the task and therefore, independent from the learner. On the other hand, the mental effort relates to how much cognitive resources a learner needs to assign to the task. Hence, this factor is not only affected by the task but also by the learner's previous knowledge and cognitive skills. The performance factor describes how the learners perform in a task. As with the mental effort, this factor is also influenced by the task, the learner, and the interaction among their characteristics (F. G. W. C. Paas & Van Merrinboer, 1994). The measurement of the cognitive load is difficult to determine because it has multiple variables that can be assessed and that can mitigate the effect of the others. For example, the amount of mental effort invested by the learner can influence her/his performance characteristics (F. G. W. C. Paas & Van Merrinboer, 1994). However, this would also increase the cognitive load.

Additional models have been built based on the subjective measure of mental effort. F. G. Paas and Van Merriënboer (1993) proposed a model to assess the efficiency of an instructional design based on the mental effort and the performance. They suggested that if the performance for two instructional materials were the same but the mental effort were less for one of them, that instructional material could be considered more efficient. A graphical representation of the efficiency construct is depicted in Figure 3.3. The mental effort goes along the x axis while the student performance goes along the y axis. A low mental effort together with a high score means high instructional efficiency, while a high mental effort with a low performance implies a low instructional efficiency.

The secondary task measurement employs an additional task to the learning activity to measure the cognitive load (Sweller et al., 2011). This additional task is not related to the learning materials. Instead, it is usually an external stimulus

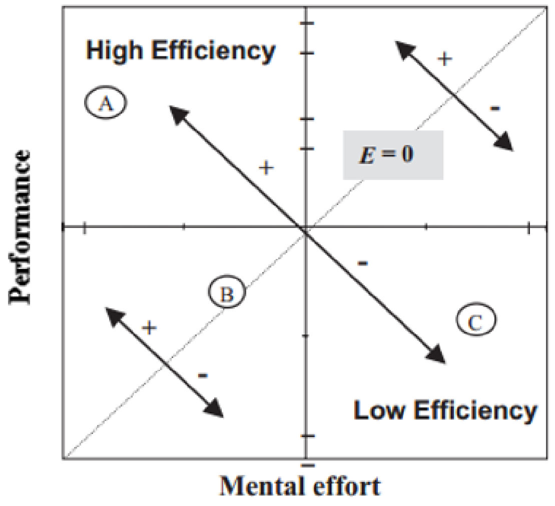


Figure 3.3. A model of human information processing stages (Wickens et al., 2015)

(e.g., a sound or a visual signal) to measure performance factors such as accuracy or response time (F. Paas, Tuovinen, et al., 2003). The assumption under this measurement is that when the learner i facing high cognitive load, the performance in the secondary task will decrease. Similarly, when the cognitive load is low, the working memory will have available resources to receive the external stimulus and respond to it.

In addition to these approaches, physical measures have also been used to try to calculate the cognitive load. Heart rate, pupil dilation and magnetic resonances are some examples of these (F. Paas, Tuovinen, et al., 2003; Sweller et al., 2011). Although some of these approaches have demonstrated some efficacy (e.g., pupil dilation), they will not be considered in this document due their invasive nature. Also, although the secondary task has shown to be a useful measure, it also intrudes in the learning task by interrupting the learner while working with the material. Therefore, the cognitive load measure that will be employed for this project is the subjective measure of mental effort.

Nevertheless, is there any way to differentiate among the intrinsic, the germane, and the extraneous load? Sweller et al. (2011) suggested that the extraneous load can be measured by varying the instructional design for the same learning material (i.e., intrinsic load is constant). Some other authors have proposed different subjective scales trying to map specific questions to certain cognitive loads. For example, Cierniak, Scheiter, and Gerjets (2009) mapped the intrinsic load to the question "How difficult was the learning content for you?"; the extraneous load to "How difficult was it for you to learn with the material?"; and the germane load to "How much did you concentrate during learning?". They found a positive correlation between the germane load and the students' performance suggesting that they were able to measure germane load adequately. However, they found inconsistency in the measurement of intrinsic and extraneous load. As in the Cierniak et al. (2009) study, other attempts have failed to measure different cognitive load. Different authors suggest that learners, especially novice learners, cannot differentiate between different kinds of cognitive load (Cierniak et al., 2009; Sweller et al., 2011). Therefore, it is unlikely that researchers will be able to distinguish among these cognitive loads (Kirschner, Ayres, & Chandler, 2011).

3.5 How to support complex learning?

Using pure complex learning tasks for novice learners would generate a high cognitive load, which would have a negative impact on students' learning, motivation and performance (Van Merriënboer et al., 2003). If the learners are not motivated during the learning activity, independently of how much the extraneous load is reduced, they will not devote the resources to the intrinsic load (Van Merrienboer & Sweller, 2005). Besides, the purpose of an effective instructional design is not just reducing the cognitive load to zero. It should also reduce the extraneous load and activate the germane load so that the working memory can be devoted to the intrinsic load (Moreno & Park, 2010a; Sweller, 2010).

A good instructional design should encourage the creation of schemata and the automation of elements that are consistent across problems (Van Merrienboer & Sweller, 2005). There are different approaches that use CLT to propose instructional design methods, in order to support complex learning. They vary from techniques that reduce the intrinsic or the extraneous load, to general guidelines describing how to organize these techniques.

3.5.1 Reducing Intrinsic Load

There is no yet an agreement on whether the intrinsic load can be reduced (Moreno & Park, 2010b). Sweller et al. (2011) suggest two indirect ways of "reducing" it. Splitting the elements from a learning task, as much as possible, will reduce the interactions and consequently the number of germane resources from the working memory that are required to fully understand a learning material. In the example from Figure 3.2, an alternative would be to introduce loops, by first using a while loop clause, which has a simpler structure. Also, making sure that the student understands that 'i++' increases the value of i by one. However, not everyone agrees on this approach, arguing that the reduction of element interactivity can also reduce the learning outcomes (Moreno & Park, 2010b)

The other way to reduce intrinsic load is when learning occurs. As described before, when learning takes place, multiple elements become a schema in the long-term memory (F. Paas, Tuovinen, et al., 2003). This schema is then treated as a whole and can be retrieved from the working memory using only one germane resource as opposed to the many resources that would be needed to deal with distinct elements.

3.5.2 Reducing Extraneous Load

Problem solving approach alone involves a high extraneous cognitive load (Merril, 2002; F. G. Paas & Van Merriënboer, 1994; Sweller et al., 2011). Novice learners usually approach a problem backwards using means-ends analysis (Bransford et al., 2000; Sweller, 1988). They start from the goal and use the "givens" (i.e., initial problem state) to try to fill the gap with the current problem state (F. Paas & Kirschner, 2012). This search process generates extraneous load to the learners, because they have to consider many elements at once.

Empirical research in CLT has identified several effects of instructional design practices in the extraneous load (see Table 3.1 for a summary). The following sections will describe each of these.

3.5.3 Goal Free Effect

The main idea of this practice is to allow learners to explore the materials without a specific goal in mind (P. L. Ayres, 1993; Gray, St Clair, James, & Mead, 2007; Sweller et al., 2011). For example, instead of asking them to find a specific angle in a triangle, the learners can be requested to use the givens and their previous knowledge to find as many unknowns as possible (P. L. Ayres, 1993).

This approach is founded in the different ways experts and novices solve a problem. As opposed to experts, novices usually start from the goal, using a means-ends analysis to find the solution to a problem (Bransford et al., 2000; M. T. Chi et al., 1981). If the activity does not include a specific goal, the learners have to explore the givens and work forward as experts do (Sweller et al., 2011). They will only use the current state and the givens to find the next possible problem state (F. Paas & Kirschner, 2012). This would become an iterative process until no additional states can be found.

It has been suggested that the goal-free approach generates less cognitive load than the means-ends analysis for problem solving (For a summary, see Ch 7. (Sweller et al., 2011)). However, this can only be productive for problems with a limited number of unknowns (Sweller, 1988). If the problem contains hundreds or thousands of possible states, this technique is unproductive because the learner will

Table 3.1.

Effects and How They Reduce the Extraneous Load - Based on (Moreno & Park, 2010a; Sweller et al., 2011; Van Merrienboer & Sweller, 2005)

Effect	Description	Reduction of Cognitive Load
Goal-free effect	The learner works in the materials without a specific goal in mind. Instead, she/he uses the given information to explore the possible actions and states in a problem.	By removing the goal, the novice learner will no longer be able to use means-ends analysis. The learner now has to focus on the givens and the problem states, and work forward to solve the problem.
Worked example effect	The learners study worked examples instead of a problem solving approach alone.	It focuses learners' attention on the problem states and prevents her/him from using a means-ends-analysis. It provides the learners with an expert solution to the problem
Completion problem effect	Partially solved worked examples are studied and completed by the learner.	It diminishes the cognitive load of the problem solving approach by reducing the problem. After an initial exposure to worked examples, the learners need to engage in practice problem with some support.
Split attention effect	Integrating different representations of mutually referring information into a single one.	The learner does not need to devote cognitive resources to integrate two sources of information
Modality effect	Using two different channels (e.g., visual and auditory) instead of a single one.	The different channels can be processed simultaneously but independently.
Redundancy effect	Integrating different source of "self-contained" information into a single one.	The learner no longer will need to process redundant information.

have to go through all of them. Therefore, this approach will be beneficial for learning whenever the intrinsic load is high and the number of problem states and operator is limited.

3.5.4 Worked Example Effect

A worked example comprises a problem statement, a step-by-step solution, and auxiliary representations of the given problem (Atkinson et al., 2000). It provides novices with a model about how experts solve certain problems (Atkinson et al., 2000; Van Merrienboer & Sweller, 2005). After studying the example(s), this model can become a schema stored in the long-term memory that will support the solution of similar problems (Sweller et al., 2011).

As with the goal-free effect, the worked examples may reduce the cognitive load as compared to novice problem solving (Sweller et al., 2011). Presenting demonstrations is a more effective form of instruction than only presenting information (Merril, 2002). The learners can be focused on specific steps of a well-solved problem, instead of exploring all potential solutions (Kester et al., 2010; F. G. W. C. Paas & Van Merrinboer, 1994). Thus, they are reducing the extraneous load from the working memory, which should be devoted to germane load. The addition of paired practice problems to the worked examples can be more effective than just studying a block of examples (Kester et al., 2010; Trafton & Reiser, 1994).

There are some conditions under which the worked example effect does not occur (Kalyuga, Chandler, Tuovinen, & Sweller, 2001). Examples of these conditions include mutually referring instruments or non-integrated multiple representations in a worked example. The split attention effect and the redundancy effect explain this phenomenon. The learner has to devote their working memory resources to integrate the separate instruments, or to process redundant information. These resources would otherwise be devoted to manage the intrinsic load towards schema creation and automation.

Another condition is when the learners have achieved certain level of knowledge (Kalyuga et al., 2001; Renkl, 2005). Learners with predefined schemata prefer to use them instead of studying examples. Through the use of predefined schemata, they may not need to use means-end analysis for problem solving and consequently, it will not generate a cognitive overload.

However, empirical research has demonstrated the effectiveness of worked examples under specific conditions (For a summary, see Atkinson et al. (2000); Renkl (2005)). These conditions were proposed as instructional principles for worked examples (Table 7.2). The key aspects of introducing worked examples include: (1) the alignment/integration of multiple formats; (2) the clear definition of the problem states and sub goals; (3) the variability of the problems in the examples (Merril, 2002; F. G. W. C. Paas & Van Merrinboer, 1994); and (4) the active exploration of examples through a self-explanation process (M. Chi et al., 1989; Stark, Mandl, Gruber, & Renkl, 2002). The active exploration is highly relevant because the use of worked examples is not equally effective for a learner who just reads them and one who actually depicts understanding through a self-explanation process. This understanding can be identified if the self-explanation comprises these four elements (M. Chi et al., 1989): (1) the conditions of application of the actions; (2) the consequences of actions; (3) the relationship of actions and goals; and (4) the relationship of goals and actions to natural laws and other principles. In fact, it has been demonstrated that providing an initial training session as part of the self-explanation process, may have a positive impact in both the generated explanations and the learning outcomes (Renkl, 2005). See section 3.3.6 for further description of the self-explanation effect.

3.5.5 Completion Effect

The goal-free and the worked example approaches are effective instructional practices for novice learners. However, as the learners' schemata start to consolidate and automate, the expertise reversal effect takes place (Kalyuga, Ayres, Chandler, & Sweller, 2003). This effect is intimately related to the split attention and the redundancy effects. When the learners study the worked example and have acquired

Table 3.2.

Design characteristics for effective worked examples (Atkinson et al., 2000; M. Chi et al., 1989; Renkl, 2005)

Feature	Description
Intra-Example	• The easy mapping guideline (Renkl, 2005): The use of multiple formats and resources is important when designing worked examples. However, different formats should be fully integrated to avoid extra cognitive load generated by the split attention effect. • The meaningful building-blocks guideline (Renkl, 2005): The example should be divided in sub goals or steps to make it easier for the student to understand. Labels and visual separation of steps can be used for this purpose.
Inter-Example	• The structure-emphasizing guideline (Renkl, 2005): The use of multiple worked examples (at least two of them) with structural differences can improve the learning experience. The worked examples should be presented with similar problem statements that encourage the students to build schemata based on analogies and the identification of declarative and procedural rules.
Environmental	• Self-explanation effect (M. Chi et al., 1989): Students should be encouraged to self-explain the worked examples in order to be actively engaged with them. Some strategies that support this process are: (1) Labelling worked examples and using incomplete versions of them; (2) Training self-explanations; and (3) Using cooperative learning.

some schemata, they will try to make the connection between the two sources (i.e., schemata and examples), generating an additional unnecessary cognitive load. Another explanation to the expertise reversal effect is provided by Renkl and Atkinson (2003). They explained that once a schema has been acquired, then the goal is to automate it. Hence, practice in problem solving will be more beneficial than just self-explaining examples.

Anderson, Fincham, and Douglass (1997) presented a framework of skill acquisition called Adaptive Control of Thought-Rational (ACT-R). In this four-stage framework, the learners start to solve problems using analogies from the examples. After studying the worked examples, the learners will have gathered the basic declarative rules of the complex task, which allow them to understand the basic principles of the problems. Then, during a third stage, the procedural rules start to be evident to the learner due to the practice problems. Finally, on the fourth stage, they have already created schemata that allow them to solve different problems without using the examples.

The ACT-R framework is similar to the three phases of skill acquisition identified by VanLehn (1996). During the early phase, the learners are exploring the materials without trying to solve any problems. Instead, they are gathering the concepts and principles around the topic. For the intermediate phase, the learners are starting to solve some problems. Before they try to solve problems on their own, they explore existing solutions (i.e., worked examples). They go back and forth to the examples in order to reference certain steps. During the late phase, learners do not need to reference other materials to solve problems. They have acquired schemata that are starting to be automatized and refined for accuracy.

Both models (Anderson et al., 1997; VanLehn, 1996) suggest that the learner only needs the worked examples during the initial or intermediate stage of skill acquisition. At the beginning, the learners are only able to manage some context, principles and rules. Then, they use analogies from the experts' approaches to acquire some procedural knowledge. After some practice, they are able to solve problems on their own using the schemata and without any additional references to the worked examples. These phases or stages do not have a clear boundary with each other. At a given moment, the learners might be in different phases for different concepts from a complex task (Renkl & Atkinson, 2003).

Renkl and Atkinson (2003) conducted various studies to understand how the transition from worked examples to problem solving should be implemented. They suggested a fading approach in which the learner is first exposed to a fully worked example. One step of the solution is removed for the second task. Two steps are removed for the third one. This process will continue until the learners have to solve a whole problem on their own. They evaluated two different approaches for fading: (1) Backward fading removing the last step of the solution, then the previous one, and so on; and (2) Forward fading removing the first step of the solution, then the second one, and so on. Both approaches were effective in near transfer but only the backward fading supported far transfer. They argued that the backward fading would keep the cognitive load low, by providing the first steps of the solution. However, additional studies have suggested that the better fading approach might depend on the learning material (Sweller et al., 2011).

3.5.6 Explanations

Creating explanations is a vital part of the learning process as well as an important communication skill in science and engineering (Lombrozo, 2006). Scientific theories are basically explanations of phenomena (Sandoval & Millwood, 2005). By constructing explanations we make sense of the world around us. We create connections between the phenomenon experienced and the knowledge previously acquired. Hence, creating effective explanations requires that we have a clear conceptual understanding of the principles and theories, plus the knowledge of how to apply these principles to different context, and the understanding of what entails a high quality explanation (Sandoval & Millwood, 2005).

3.5.7 Nature of Explanations

The process of explaining can be described as a process of pattern subsumption (Williams, Lombrozo, & Rehder, 2010). In this view, an explanation involves the generalization of a phenomenon to a known pattern. The generalization process is given by using features of the phenomenon, either provided by the context or artificially created by the explainer (Lombrozo, 2006). The identification and extension of these patterns allow us to use them in other contexts.

Explaining and self-explaining have demonstrated to be important cognitive processes beneficial to a meaningful learning that can support transfer (M. T. Chi & Roy, 2010; Mayer, 1981). Explaining can help to (a) generalize properties to common abstract patterns (Williams et al., 2010), (b)identify causal inferences constraining the possible causes to the available prior knowledge (Lombrozo, 2006), and (c) identify and repair learners' misconceptions, and fill the gaps of instructional materials with inferences based on previous knowledge (Chiu & Chi, 2014; Lombrozo, 2006). When an explainer tries to identify the causes, she/he will seek for similar phenomena that can be applicable to this context.

The causality perspective to explanations relates to Aristotle's four causes of explanations. Aristotle considered that we cannot really know something until we understand what causes it, answering the question 'why?' Thus, he identified four ways to answer this question: (1) the efficient cause involves the sources of change in the phenomenon, who or what was the responsible for making something what it is; (2) the material cause describes what the object is made of; (3) the formal cause focuses on the properties and characteristics that make the phenomenon what it is; and (4) the final cause describes the function or goal of the phenomenon. The formal and final causes are "psychologically real modes of understanding" (Lombrozo, 2006, p. 465). For instance, while children equally accept writing as the final cause of a pencil, and climbing as the final cause of mountains, adults are more selective about when to accept the final causes. Similarly, the features that make an object what it is (formal cause) can be naturally selected by the explainer, but they do not depend on personal preferences (Prasada & Dillingham, 2006). For example, we accept that a dog has four legs and that is one of the characteristics that makes it a dog, but we do not accept the color of the dog (e.g. brown) as one of these formal causes.

In the learning process, the function of explanations is to facilitate the integration of novel information and previous knowledge (Aleven & Koedinger, 2002; M. T. H. Chi, De Leeuw, Chiu, & Lavancher, 1994; Lombrozo, 2006). The learners' previous knowledge will enable them to identify relevant characteristics of the phenomenon that can be related to the causation. Hence, learners' previous knowledge determines their ability to identify these patterns. Once a concept has been understood, the explainer will apply that knowledge to similar contexts (Shafto & Coley, 2003). Consequently, understanding the characteristics and structure of students' explanations allow us to identify when and how these improve (Sandoval & Millwood, 2005). Moreover, by exploring multiple student explanations within the same course, we can also identify whether the characteristics and the quality of their explanations are changing over time or not.

3.5.8 Quality of Explanations

How can we distinguish high quality explanations to low quality ones? One approach to the quality of explanations correspond to the appropriate use of principles or laws within certain context. For example, Küchemann and Hoyles (2003) developed a coding scheme based on students' explanations to geometric concepts. The lowest quality involved students' perceptions, or simply paraphrasing the given information. Higher quality explanations involved the use of geometrical properties and principles. Similarly, several studies have found that students making connections to laws or principles while self-explaining, perform better in assessment tasks (Pirolli & Recker, 1994; Renkl, 1997).

Another existing approach to the quality of explanations involves their appropriate application of conceptual knowledge, and the structure of the evidence on which the claims are backed up (Sandoval & Millwood, 2005). This structure involves the warrant (i.e., the data used to support a claim), and how the explainer presents this warrant: rhetorical reference. This reference could go from simply including the data, to an actual interpretation of the data in the context of the explained phenomenon. When students use their own experiences to back up their claims, they better integrate the explained ideas to their own beliefs (Bell, 2000). Nevertheless, students often fail to provide warrants during their explanations (Sandoval & Millwood, 2005).

Finally, M. Chi et al. (1989) proposed that for an explanation to depict understanding, it should include these four elements: the consequences of actions, the conditions of application of actions, the relationship action-goals, and the relationship of actions to laws and principles. These three approaches to quality of explanations have in common the use of domain knowledge (laws and principles) to go beyond the explained material.

3.5.9 Self-Explanation Effect

Instructional materials such as textbooks or worked-examples often omit certain components or features of the phenomena they present (M. T. H. Chi et al., 1994). Each phenomenon involves local features and components, hierarchies and relationships among them, as well as relationships with other components. However, describing all these elements in detail would not be feasible nor practical for an instructional material. According to the self-explanation effect, learners must engage in a self-explanation process of examples or instructional materials to take an additional advantage from them (Sweller et al., 2011). A systematic explanation involves understanding all these features and components of the phenomenon, and the interactions among them by contrasting the provided information with background knowledge (M. T. H. Chi et al., 1994).

Self-explaining is a constructive activity that engages the learners in generating explanations of an instructional material (Chiu & Chi, 2014). Constructive activities generate a requirement on the learners to perform an action that will result in an output beyond the provided materials (M. T. H. Chi, 2009).

Constructive activities are a more effective instructional approach than active learning, and passive learning activities. The use of a constructive activity such as promoting self-explanations supports different mechanisms that favor the learning process (M. T. Chi & Roy, 2010): (1) integrating previous knowledge to the learning material; (2) integrating different elements within the learning materials; (3) helping the students to complete missing information; and (4) by integrating to previous knowledge, helps to expose, modify or repair misconceptions (Makatchev, Jordan, & VanLehn, 2004). The self-explanation process involves a mechanism that connects new learning materials with existing knowledge, in order to create schemata in the long-term memory (Chiu & Chi, 2014). The process of making these connections to create new schemata or modify existing ones makes this process a personal meaningful learning (M. T. Chi & Roy, 2010; Mayer, 1981). Therefore, learners with different skills and background knowledge may benefit differently from the self-explanations (Chiu & Chi, 2014).

Self-explanations also enable subjects to identify more abstract categories that can be integrated into a pattern. For instance, Williams et al. (2010) compared students' ability to identify two categories of robots. The experiment asked one group to explain why certain robot belonged to a given category, and the other group just described each robot. While the description group made significantly more mentions to explicit characteristics of the robots (i.e., color, body, and feet), the explaining group was able to identify the pattern under which the categories divided the robots: pointy feet vs. flat feet.

Self-explanations can also be used as a meta-cognitive strategy through the use of monitoring activities that allow the student to identify what they understand and what they do not understand (Williams et al., 2010). For example, M. Chi et al. (1989) suggested that a clear understanding of the example can be seen by a self-explanation containing the following elements: (1) the conditions of application of the actions; (2) the consequences of actions; (3) the relationship of actions to goals; and (4) the relationship of goals and actions to natural laws and other principles. They identified different characteristics of “good” and “poor” explainers of worked examples. “Good” students learn with understanding by: (a) producing more explanations; (b) monitoring their learning process with statements such as “I can see now how they did it”; and (c) visiting the examples less frequently and more accurately when solving additional problems. In a follow-up study, M. T. H. Chi (2009) confirmed that high performers produced more explanations, and additionally identified three sources of these explanations: (1) background knowledge (30%); (2) previous sentences (41%); and (3) experiences, analogies, and logical inferences (29%).

Renkl (1997) built on top of these and other findings to identify individual differences on students' self-explanations with a larger sample size and a more controlled experiment. The author identified two types of effective explainers: the “reasoners” and the “principle-based” explainers. The “reasoners” would try to solve a worked-example while explaining it. Reasoners attempted to describe what would be the next step in the solution before reading it. The “principle-based” ones would come up with instances of laws or principles from their background knowledge to explain the examples.

In spite of the described characteristics for effective self-explainers, only a small number of students fall under this category (Chiu & Chi, 2014). Furthermore, although explanations are beneficial to learning, and can support generalization and transfer, they are profoundly influenced by learners' previous knowledge (Kuhn & Katz, 2009; Lombrozo, 2006). Learners with different skills and background knowledge may benefit differently from the self-explanations (M. T. Chi & Roy, 2010; Chiu & Chi, 2014).

3.5.10 Self-explanations in Programming

Self-explanations have also been explored in the context of programming (Pirolli & Recker, 1994; Vieira, Roy, Magana, Falk, & Reese Jr., 2016; Vieira et al., 2015). Using think-aloud protocols, Pirolli and Recker (1994) analyzed student self-explanations of worked examples on recursion programming. They found similar patterns for good vs. poor explainers as those presented by M. Chi et al. (1989), and Renkl (1997). They analyzed the student think-aloud activities using a coding schema that included domain statements, monitoring statements, strategy statements, activity statements, and reread statements. The domain statements were focused on explanations related to Lisp, programming, and recursion. These domain statements could fall into one of the following categories: (1) an operation of Lisp code; (2) a result of a computation; (3) an input or parameter; (4) a structure of the code, conditionals or loops; (5) an 'is-a' statement mention an instance of a concept; (6) a 'reference' statement mention a concept based on an instance; (7) a purpose of the code; (8) an analogy; (9) entailments or implications of an action; or (10) a programming plan.

The findings from Pirolli and Recker (1994) suggest that good explainers produce significantly more domain explanations. The authors integrated explanations in the categories 'is-a' and 'reference' statements as a single category called 'tie'. Within the domain statements, good explainers produced more 'ties' together with analogies than poor explainers. Good explainers also produced more explanations related to recursion than poor explainers, who focused on surface features of the code.

In this context, written explanations have also been used as an assessment strategy in the context of programming. The BRACElet (Building Research in Australasian Computing Education) project integrated academics from several Australian universities aiming at using action research to explore college student learning process in programming assignments (Lister et al., 2012). Through the analysis of a modified version of the end-of-the-first-semester exams, the BRACElet team determined a hierarchy of skills related to programming (Lopez, Whalley, Robbins, & Lister, 2008): (1) Explaining an existing code; (2) Tracing values of variables after a code has been executed; and (3) Write code. They also identified that student proficiency on explaining the code, and tracing activities can explain a larger percentage of student ability to write an algorithm (Lopez et al., 2008). These findings suggest that explaining and tracing programming code are intermediate skills towards the end goal of writing a computer program, while the basic skills comprise the recognition of programming constructs ("A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer", 2009).

The research team adapted the SOLO taxonomy (Biggs & Collis, 1982) to categorize the student explanations among four levels (Sheard et al., 2008):

- Prestructural comprised incorrect student explanations.
- Unistructural involved brief descriptions of only parts of the code.
- Multistructural described the explanations that showed basic understanding of all independent lines of code, but not the program as a whole. An alternative category that was identified in the data was Multistructural with error, when the explanation involved characteristics of multistructural explanations but was not entirely correct.
- Relational level groups student explanations that describe the overall purpose of the code, the students "see the forest" and not just the individual trees. This level included three subcategories: (a) Relational with extra, when the student not only explained the purpose of the code, but how it was achieved; (b) Relational but error, when the explanation of the overall goal was not entirely correct; and (c) Relational incomplete, when the explanation of the overall goal of the function was limited.

The research teams compared how a novice explanation would differ from an expert explanation. They found that novice programmers will explain each line of code independently from each other, while experts tend to explain the 'big picture': "see the forest and not just the trees." (Tan & Venables, 2010, p. IIP29). For instance, a novice programmer would describe a chunk of code such as a loop in terms of the execution of this instruction instead of summarizing the overall purpose of creating this loop (Thompson, Luxton-Reilly, Whalley, Hu, & Robbins, 2008). Since their focus was on assessment and not on understanding of a worked-example, students were actually instructed on giving this holistic explanations of the code. Nonetheless, novice programmers kept explaining lines of code independently from each other (Lister et al., 2012). Students' levels of explanations were consistent throughout different questions, and students in the multistructural or relational levels would perform better in questions related to writing code, than students in the prrestructural or unistructural levels (Tan & Venables, 2010).

3.5.11 Promoting Self-explanations

Fostering and training learners in how to carry out self-explanations can be beneficial for learning (Aleven & Koedinger, 2002; Chiu & Chi, 2014; Renkl, 2005). Student learning can be enhanced by simply prompting self-explanations (M. T. H. Chi et al., 1994; Schworm & Renkl, 2006). In the context of worked-examples, Schworm and Renkl (2006) suggested that prompting or eliciting self-explanations should be a "must". These prompts can be performed either by a human or a computer with the same effectiveness in both cases (Hausmann & Chi, 2002).

Self-explanations are more frequent and more effective in learning materials that involve multiple representations such as graphics instead of only text. However, the learners need to engage in order to make connections among these multiple representations in order to take advantage of the self-explanation effect (M. T. Chi & Roy, 2010). Hence, the multiple representations in the instructional material should be easy to integrate in order to avoid the split-attention effect. Similarly, these explanations do not necessarily need to be talked, but it can take other forms such as written (Schworm & Renkl, 2006) or using diagrams (M. T. H. Chi et al., 1994), although the use of written explanations can reduce the number of student descriptions as compared to verbal explanations (Hausmann & Chi, 2002).

Identifying effective self-explanations strategies and sharing them with the students can extend the self-explanation effect. For instance, a couple of existing models for introducing self-explanations involve having students evaluating existing self-explanations (Aleven & Koedinger, 2002), or a three step process where students are exposed to: (1) an introduction to self-explanations; (2) a video representation of other students doing self-explanations; and (3) an activity to practice and provide feedback on self-explanations (Chiu & Chi, 2014). Conversely, when instructional explanations are present, the self-explanation effect is diminished (Schworm & Renkl, 2006). A possible cause of this phenomenon is that the teacher is already providing certain schemata that will later limit the reflection process during the self-explanation stage, hindering the learning process.

Other instructional elements that have been studied as related to the self-explanation effect are (Chiu & Chi, 2014): (1) using incorrect or incomplete examples can also promote the monitoring and revision of learners' misconceptions; (2) the coherence in the instructional material makes a difference in the effectiveness of the self-explanation; and (3) encouraging gap-filling when the students have little background knowledge, and transitioning towards example comparison in a more advanced level of expertise.

In general, promoting self-explanations in a learning environment can support the development of student meaningful learning, beneficial for transfer and problem solving skills. The benefit of this process comes from actively engaging students to ask themselves why something is as it is presented, which makes them focus on causes that are not obvious (Williams & Lombrozo, 2010). Moreover, this process makes the learner reflect and validate what they already know, contrasting this with new available information (Chiu & Chi, 2014). Effective self-explainers produce more explanations, connect these explanations to laws or principles existing in their background knowledge, and highlight more often the conditions and goals

(Chiu & Chi, 2014). By doing this, explainers fill the gaps of information existing in the instructional materials, integrate different parts of the instructional material (M. T. Chi & Roy, 2010), and identify abstract patterns from the learning materials that will be applicable in transfer problems (Williams & Lombrozo, 2010).

Lombrozo (2006) presented a review on the structure and function of explanations. A prevalent structure of explanations is the causal explanation. The learner identifies what is causing the phenomenon that is being explained and how that cause can be an instance of a common pattern. Hence, the previous knowledge of the learner determines her/his ability to identify these patterns. Furthermore, the learner needs to identify relevant characteristics of the phenomenon that can be related to the causation.

3.6 Critiques to Cognitive Load Theory

In spite of the 30 years of research and the several applications of CLT in educational research, there is still a lot of work to do. There are some authors that have specific critiques to this approach. Kirschner et al. (2011) presented a review of the good, bad and ugly elements of a special issue in CLT. First, learning is more complex than just reducing extraneous load, and assuming all other cognitive load is germane. It is very difficult to control all the aspects that may affect a learning environment (e.g., previous knowledge, beliefs, motivation) to be able to simplify a phenomenon under research to these forms of cognitive load (De Jong, 2010; Moreno & Park, 2010a). Hence, there are some studies that contradict each other or have unexpected outcomes. Authors usually try to explain their unexpected outcomes with external factors, not included in CLT, as opposed to trying to understand them with a follow-up experiment (Kirschner et al., 2011).

Another challenge in CLT is to find an adequate and trustworthy method to measure the cognitive load. In fact, probably the most popular method is a single question that cannot even be statistically evaluated for reliability and validity in a single study (De Jong, 2010). There are also differences regarding when the question should be asked. To complete the problem, the ideal method should not only measure the overall load, but also should distinguish among the different types of load.

3.7 Summary and Discussion

Complex learning intends to integrate and coordinate concepts, procedures and skills to be applicable to real-life problems. The interactivity of all these elements is what makes it cognitively complex. The cognitive load theory explains how learning occurs by describing a cognitive architecture with a vast long-term memory and a limited working memory.

Two types of cognitive load are directly related to the learning task: intrinsic load and extraneous load. The intrinsic load is given by the element interactivity. It is managed by the germane load, which is directly beneficial for the learning process. When all the interacting elements are processed, a schema is created in the long term memory and learning has happened. The intrinsic load cannot be reduced unless the interactivity or some of the elements are removed from the learning task. This would arguably change the learning outcome and affect the learners' understanding.

The extraneous load is not beneficial for the learning process and is usually generated by a poor instructional design. For example, a problem solving approach generates a high extraneous load in novice learners. Novice learners employ a means-ends analysis technique that engages them in a search process with many variables. They need to close the gap between the current problem state and the goal, using the differences among them, the available operators and the sub goals.

The cognitive effects related to complex learning were introduced: (1) goal-free effect; (2) worked example effect; (3) completion effect; and (4) self-explanation effect. The goal-free effect removes the goal from the problem solving approach so that the learner works forward to solve the problem. She/he will go step-by-step using the current problem state and the available operators. Eventually, the learner will get to complete the solution.

The worked example and the completion effect provide the learner with an expert solution to the problem. Both approaches benefit from the self-explanation effect, which relates the learner prior knowledge to the materials. The difference is that the completion approach deals with the expertise reversal effect. It assumes that after been exposed to worked examples, the learners need to start working on problem solving (at least partially). Otherwise, at that skill acquisition stage, the worked examples would provide extraneous load.

The cognitive load theory provided an explanation to complex learning using the cognitive architecture and the cognitive loads associated with a learning material. Therefore, the instructional strategies presented here are grounded on this theory. However, there are other instructional design strategies that may reduce the extraneous load to effectively support complex learning. Some examples of these strategies are: productive failure (Kapur & Bielaczyc, 2012), feedback (Hattie & Timperley, 2007), and scaffolding for problem solving (Xun & Land, 2004). Futhermore, there are some critiques to CLT related to the lack of availability of measuring methods for the cognitive load and the limited view of learning that this theory has. These can be considered a limitation of the CLT studies.

In the context of the project, programming tasks paired with disciplinary knowledge are a form of complex learning. This project will explore student self-explanations of programming worked examples. It will use different measures of performance and cognitive load to characterize the students' comments to the code. Thus, we expect to reduce the probable lack of reliability of a single-item cognitive load measure. Worked examples have been evaluated in several cognitively complex areas such as mathematics (F. G. Paas & Van Merriënboer, 1994; Schwonke et al., 2011), computer programming (Pirolli & Recker, 1994; Trafton & Reiser, 1994; Vieira et al., 2015), and physics (M. Chi et al., 1989; Renkl et al., 2002). However, it is not clear how worked examples should be designed and delivered for computer programming or computational science contexts. Moreover, the self-explanation effect has been explored in lab settings using think-aloud protocols. This approach has been useful to understand how explanations can foster learning. Nevertheless, this approach do not provide an in-class strategy to encourage self-explanations.

This study will evaluate: what are the characteristics of students' explanations of worked examples for an integrated disciplinary-programming complex task? Students will be asked to write in-code comments as a strategy to self-explain the examples. This strategy would allow teachers and faculty members to integrate the self-explanations into the classroom and homework activities. Chapter 4 describes the methodologies employed for each case study. Chapter 5 presents the first case study that focuses on the glass box approach to computing education: CPMSE. Chapter 6 describes the case study related to the black box approach to computing education: THERMO. Chapter 7 discusses the results of both cases to find similarities and differences, in the light of existing literature of self-explanations.

CHAPTER 4. METHODS

The goal of this dissertation is to *understand students' explanations of worked examples for an integrated disciplinary-programming complex task.* To do so, I explored the following research questions:

- RQ1. What are affordances of in-code commenting self-explanation activities in the contexts of black box and glass box approaches to computational science and engineering?
- RQ2. : What are the characteristics of students' explanations in a glass box and a black box approach to CSE education?
- RQ3. How do the characteristics of students' self-explanations in glass box and black box approaches to CSE education relate to their ability to program?

First, for RQ1, we defined these affordances to be comprised of the benefits derived by students in writing the in-code comments, and the ways individual students take advantage of them (Gibson, 2014). The relevance of exploring affordances lies on the fact that this relationship between an environment and the subjects is different for each individual. Then, for RQ2 we analyzed students' written explanations as in-code comments of the worked-examples, and compared them to students' ability to program for RQ3.

A pilot case study (Vieira et al., 2015) was carried out with two purposes: (1) to identify the characteristics of effective programming worked examples; and (2) to validate the effectiveness of writing in-code comments as a self-explanation strategy. The use of worked examples to scaffold programming and algorithm design learning was explored in the context of an object-oriented programming course. Different instructional design elements were assessed in order to identify effective

Table 4.1.

Overview of the pilot study

Course	Introduction to Object-Oriented Programming Purdue University
Semester	Fall 2013
Participants	35 freshmen Computer and Information Technology students
Computational Component	Introductory C# Programming
Procedures	Three lab sessions solving in class-exercises after self-explaining one worked example.

design characteristics for worked examples. We employed multiple representations of the solution, including textual, graphical and computational representations. The results suggest that providing in-code explanations as simple sentences enhanced code readability and improved students perceptions about the examples. Moreover, introducing the self-explanation process by asking students to write in-code comments helped them to actively engage with the examples. Specific suggestions include encouraging students to write detailed comments as opposed to superficial ones in order to take advantage of the examples. This approach seems to be useful for novice students who did not have previous experience in programming.

The contribution of the study is the detailed description of the implementation of worked examples in a programming context. It included the use of multiple representations as well as the use of comments within the code as a self-explanation process. These design and implementation characteristics of worked examples become the main contribution of this pilot study to this dissertation. The case studies that comprise this dissertation involved writing in-code comments as the self-explanation strategy. Furthermore, the worked examples included multiple representations and no in-code comments. The complete description of the pilot study and its implications for this dissertation are presented in Vieira et al. (2015). Table 4.1 depicts an overview of the context of this pilot study.

4.0.1 Design

This dissertation comprises two case studies to explore students' explanations in two different levels of transparency. The case study is an appropriate approach for this project because the researcher explored specific instances of the self-explanation phenomenon in the real-life context of CSE education. The findings are concrete and context dependent, which is useful to "address research questions concerned with the specific application of initiatives or innovations to improve or enhance learning and teaching" (Case & Light, 2011, p.191). In two different contexts of CSE education, we implemented an innovative instructional strategy that could help to elicit self-explanations: students' written explanations as in-code comments for computer programming.

The aim of a case study is not to generalize among populations, which would require representative samples; instead, the aim of these two case studies is to generalize towards the theory of students' explanations (Yin, 2009). Moreover, the case studies take advantage of using multiple sources and forms of data including quantitative data, qualitative data, or both strands of data (Hartley, 2004; Kohlbacher, 2006). We used students' written explanations, their responses to open-ended questions, and quantitative scores related to their ability to do computer programming. Table 4.2 includes an overview of the two case studies.

The units of analyses for these case studies are the self-explanations reported by the students on each of three activities for each course. Using two case studies allowed the researcher to compare findings doing a cross-case analysis. The cross-case analysis identified common patterns of the self-explanation effect to generalize the findings among the cases (Khan & VanWynsberghe, 2008). This strategy also supports the identification of differences on the phenomenon for these two different contexts (Patton, 2002).

Table 4.2.

Overview of the two case studies

Course	CPMSE	THERMO
Semesters	Spring 2015 / Spring 2016	Spring 2015 / Spring 2016
Participants	25-30 freshmen Materials Science and Engineering students / semester	60 Sophomore Materials Engineering students / semester
Computational Component	Write MATLAB programs to solve engineering problems.	Using Virtual Kinetics of Materials Laboratory (VKML) Python to represent THERMO phenomena

4.1 Analytical Framework

In order to organize the structure and characteristics of students' explanations, I employed the knowledge framework for science achievement (Shavelson, Ruiz-Primo, Li, & Ayala, 2003). This framework describes four independent types of knowledge that comprise science achievement. The declarative knowledge (knowing that) corresponds to definitions and facts of the phenomenon. The procedural knowledge (knowing how) refers to if-then rules, and the steps to achieve a goal. The schematic knowledge (knowing why) comprise the use principles and mental models to describe why something is as it is, and to predict effects of actions. Finally, the strategic knowledge (knowing when, where, and how) corresponds to the identification of the conditions under which certain procedures can be applied, and the use of monitoring activities.

We aligned these four types of knowledge to the three levels of programmers' expertise (Lister et al., 2012). The level one, in which the programmer is able to understand individual instructions and trace the value of a variable falls under the declarative and procedural knowledge. The programmer knows what the variable is, and how this value changes over time once the program is executed. For the level two, the programmer already knows what the overall goal of a program is, so the programmer should know why these instructions are solving the problem, while making connections to principles and background knowledge. Finally, programmers in level three should be able to identify when, where, and how certain chunks of instructions can be applied. Students should not only explain line by line but identify the relationship that the lines have towards a general goal or a sub goal of the program. This alignment was validated with two educational researchers (one senior one junior) who had experience on the use of Shavelson et al. (2003)' framework for qualitative analysis in the context of modeling and simulation practices for engineering education.

The students' explanations were qualitatively analyzed using a coding scheme that was initially derived from previous research, and locally refined with our data. The first iteration of qualitative data analysis started with a coding scheme that included the four elements of understanding that should depict an explanation (M. Chi et al., 1989): conditions of applications of actions, consequences of actions, relationship action-goals, and relationships of the action and goals to laws and principles. This analysis process for students' comments of a worked-example from the CPMSE course extended the coding scheme to ten codes (Vieira et al., 2016). This coding process was carried out in a line by line basis. This is, assigning one or more of these codes to each line, and then counting the total number of instances of each code within each student file.

However, further analysis of different examples showed that students usually write comments by sections. For example, in one of the Python codes that started by creating six different variables, some of the students would simply write a single comment describing what all these lines did, while some other students would write the same comment six times (e.g. #Creates a variable). If we use the former approach (i.e., line by line), students replicating the same text multiple times would inflate the results. As a consequence, the qualitative analysis is now presented by code sections instead of individual lines. These types of sections were established for each context because the programs, the goals, and the way students used them were different from each other.

Table 4.3 presents how we implemented this conceptual framework for the qualitative data analysis process. The declarative knowledge involves comments where students limited their explanations to what a line of code was doing (COA), what a variable represented (VAR), what data type it was (DAT), what the parameters of an instruction were (PAR), and what an instruction of code was (COD). The procedural knowledge involves describing how something was done (HOW), or how it would happen in execution time (EXE). The schematic knowledge involves understanding the problem (PRO) and its goal (GOA), the use of background knowledge (BGK) or examples (INS), usually to explain why something happened (WHY), or its relationship with the overall goal (RAG). The strategic knowledge is related to the conditional knowledge (CON), monitoring statements (MON), and the identification of chunks of code with certain purpose (CHK). If a student wrote and explained their own solution (OWN), it is also considered an instance of strategic knowledge because they were able to identify when, where, and how to apply their knowledge.

In addition to the four types of knowledge and the three levels of expertise, we added a Level Zero of expertise, which comprises the explanations that were limited, incorrect, paraphrased, or copied literally from the example text.

These categories within each type of knowledge were also grouped based on the type of explanation the students wrote. For instance, the declarative knowledge was divided between those categories focused on what the program did, and the ones focused on the code itself (e.g. DAT data type). Likewise, the schematic knowledge was divided into two types of explanations: use of schemata and rationale. The former type focuses on student use of background knowledge (e.g. PRO, BGK), while the latter refers to explanations of why the code was built this way. Finally, the strategic knowledge was divided between explanations identifying chunks of code that worked under certain conditions (i.e. CON and CHK), and the

Table 4.3.

Coding scheme for the characteristics of students explanations

Level	Type of Knowledge	Type of Explanation	Code	Description
Zero	Limited Knowledge (LK)	Limited	SIM	The comment is similar to the example text
			INC	The statement is incorrect
			LIM	The statement is incomplete
			PHR	The comment is paraphrasing the instruction
One	Declarative Knowledge (CK)	Program	COA	Describes what an instruction does, the consequences of actions
			VAR	Describes what a variable represents
			PAR	Describes the parameters or returning values of an section
		Code	DAT	Describe the data type of the variable
			COD	Describe the code in terms of the program
	Procedural knowledge (PK)	Process	HOW	Describes how this instruction does what it does
			EXE	The comment describes what will happen in execution time
Two	Schematic Knowledge (SK)	Use of Schemata	PRO	The student makes connection with the problem phenomenon
			GOA	Describes the goal what is the goal of a function for example
			BGK	The student uses background knowledge or principles to describe the instruction
		Rationale	WHY	Describes the rationale for an instruction Why it is set certain way
			RAG	Describes the relationship action-goal what is some instruction for
			INS	The comment includes an instance of the instruction with certain values
Three	Strategic Knowledge (TK)	Chunks	CON	Describes the conditions under which the instruction works
			CHK	The student identifies chunks of code that with certain goal (one set of comments for more than one line describing its purpose).
		Meta - cognition	MON	The comment involves monitoring statements
			OWN	The student developed his/her own solution to the example

use of meta-cognitive strategies such as the monitoring statements or building their own solution.

4.2 Methods Case One - Glass Box Approach

The glass box approach corresponds to a freshmen level course called computation and programming for materials scientists and engineers (CPMSE). This course was considered to be the glass box approach to CSE education because students had access to the underlying mechanisms of the simulations they used and created. The learning outcomes of this course were focused on students' ability to create MATLAB® programs to solve engineering problems using models of physical and biological systems (Magana, Falk, & Reese Jr, 2013).

4.2.1 Participants

The students enrolled in the course Computation and Programming for Materials Scientists and Engineers (CPMSE) at Johns Hopkins University during the spring 2015 and spring 2016 semesters participated in this study. Approximately 25 students enrolled in this freshmen level course every spring semester. The main learning outcome of this course was for the students to *"to apply algorithmic thinking and computer programming toward the solution of engineering and scientific problems"* (Magana, Falk, & Reese Jr, 2013). The students came with different background from high school and therefore, it is not possible to describe a unique programming background on the students.

4.2.2 Procedures, Data Collection and Data Analysis

The CPMSE course used an inverted classroom as the pedagogical approach; students watched a video lecture and took an online quiz before coming to class. During each session, students worked on solving a set of in-class programming exercises. Each set of exercises included a worked example using multiple forms of representation for the solution (e.g., text, figures, MATLAB® code, video). A sample worked example for this class is included in Appendix A.

Two strands of data were collected for this case study. The qualitative strand focused on student self-explanations from the in-code comments (RQ2), and students' response to open-ended questions where they described their affordances of the written explanations (RQ1). The quantitative strand comprised students' perceived ability to program, student performance (RQ3), and their frequency of use of the worked-examples (RQ1).

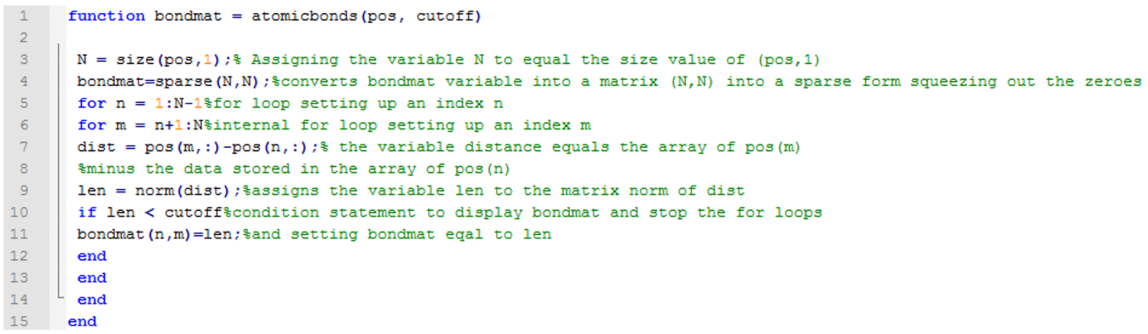
Qualitative Strand

Students' explanations

Each of the worked-examples was posted as a PDF document on the learning management system (Blackboard) as an assignment within the in-class exercises section. The document was organized in four sections (1) problem statement; (2) understanding the problem; (3) addressing the problem; and (4) MATLAB® solution. In some cases, the worked-example would include one or more links to online videos with additional explanations of the example. The instructions for the assignment were:

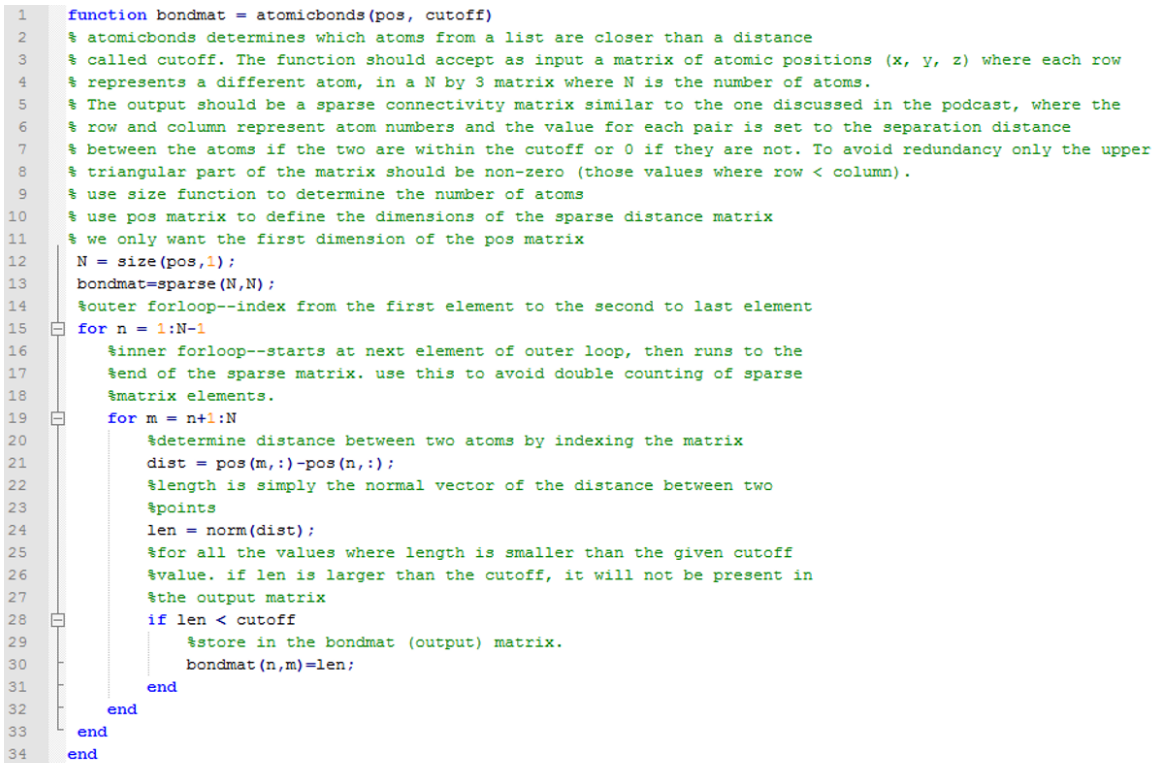
1. Please download the attached worked-out example.
2. Insert comments at the code segment of the example to explain what the code is doing.
3. Upload a MATLAB® (.m) file with the comments you made.
4. In order to receive CREDIT, please complete the Exercise 02 Worked example Quiz once you have submitted the commented file

These comments were considered the student self-explanation to the worked-examples. Sample comments for the activity #11 (Appendix A) from two students are presented in Figure 4.1 (SA1) and Figure 4.2 (SA2). These names correspond to neutral pseudonyms assigned to two students from spring 2015.



```
function bondmat = atomicbonds(pos, cutoff)

 N = size(pos,1);% Assigning the variable N to equal the size value of (pos,1)
 bondmat=sparse(N,N);%converts bondmat variable into a matrix (N,N) into a sparse form squeezing out the zeroes
 for n = 1:N-1%for loop setting up an index n
 for m = n+1:N%internal for loop setting up an index m
 dist = pos(m,:)-pos(n,:);% the variable distance equals the array of pos(m)
 %minus the data stored in the array of pos(n)
 len = norm(dist);%assigns the variable len to the matrix norm of dist
 if len < cutoff%condition statement to display bondmat and stop the for loops
 bondmat(n,m)=len;%and setting bondmat eqal to len
 end
 end
 end
end
```

Figure 4.1. SA1 in-code comments for the Example #11.



```
function bondmat = atomicbonds(pos, cutoff)
% atomicbonds determines which atoms from a list are closer than a distance
% called cutoff. The function should accept as input a matrix of atomic positions (x, y, z) where each row
% represents a different atom, in a N by 3 matrix where N is the number of atoms.
% The output should be a sparse connectivity matrix similar to the one discussed in the podcast, where the
% row and column represent atom numbers and the value for each pair is set to the separation distance
% between the atoms if the two are within the cutoff or 0 if they are not. To avoid redundancy only the upper
% triangular part of the matrix should be non-zero (those values where row < column).
% use size function to determine the number of atoms
% use pos matrix to define the dimensions of the sparse distance matrix
% we only want the first dimension of the pos matrix
 N = size(pos,1);
 bondmat=sparse(N,N);
 %outer forloop--index from the first element to the second to last element
 for n = 1:N-1
    %inner forloop--starts at next element of outer loop, then runs to the
    %end of the sparse matrix. use this to avoid double counting of sparse
    %matrix elements.
    for m = n+1:N
        %determine distance between two atoms by indexing the matrix
        dist = pos(m,:)-pos(n,:);
        %length is simply the normal vector of the distance between two
        %points
        len = norm(dist);
        %for all the values where length is smaller than the given cutoff
        %value. if len is larger than the cutoff, it will not be present in
        %the output matrix
        if len < cutoff
            %store in the bondmat (output) matrix.
            bondmat(n,m)=len;
        end
    end
 end
end
```

Figure 4.2. SA2 in-code comments for the Example #11.

Several differences can be spotted from these two examples. The most obvious distinction is the number of self-explanations. SA2 described the objective of the function as well as their parameters. Justice limited her comments to the inner lines of the function. Furthermore, SA2 described the function and the steps of the solution in terms of atoms (i.e., the disciplinary problem), while Justice was only describing programming steps. Finally, contrary to SA1's self-explanation, SA2 explicitly described the initial and end conditions of the for-loops.

The student self-explanations were analyzed using a coding scheme described in section 4.2. For instance, if a student wrote a comment within a section that says *"Define function"*, this was coded as COA. However, if the student also described the goal and parameters of this function like: *"Create a function that keeps asking numbers until it's less than all the numbers so far, then output the biggest number"*, the explanation for this section would be coded as COA, GOA, and PAR. Another example is when they explained the rationale for the way a section of the code was built (WHY): *"Since the previous line only determined the difference between each coordinate vector (x,y,z), this line finds the magnitude of the distance"*, or when they make a connection between an action and a goal (RAG): *"Essentially we are finding the number of atoms involved so we can set up our sparse matrix"*. Appendix C presents sample quotes from all the worked-examples for all the categories in this coding scheme.

Since different students wrote different explanations, we grouped them based on the type of explanations they wrote within each section. For this analysis, each type of explanation was treated as a binary variable within each section. The binary data corresponded to whether the student used (1) or not (0) each type of knowledge within the sections of the code. For instance, Figure 4.3 shows the types of explanations students wrote for the second section of activity #5. The rows in the box represent students while the columns represent the codes described in Table 4.3. The colors represent the different types of explainers we identified for this activity. In this example, student S5 (in red) was the only one who identified the conditions of application of actions (CON), so he/she had a one in Chunks for this section. The rest of the students had a zero. When students showed more than one category within the same type (e.g. S14 COA and VAR within Program), this was still counted as one. The distance between students was computed as the proportion of columns that do not match with a zero or one value, and the number of clusters depended on the variability of students' explanations within each activity.

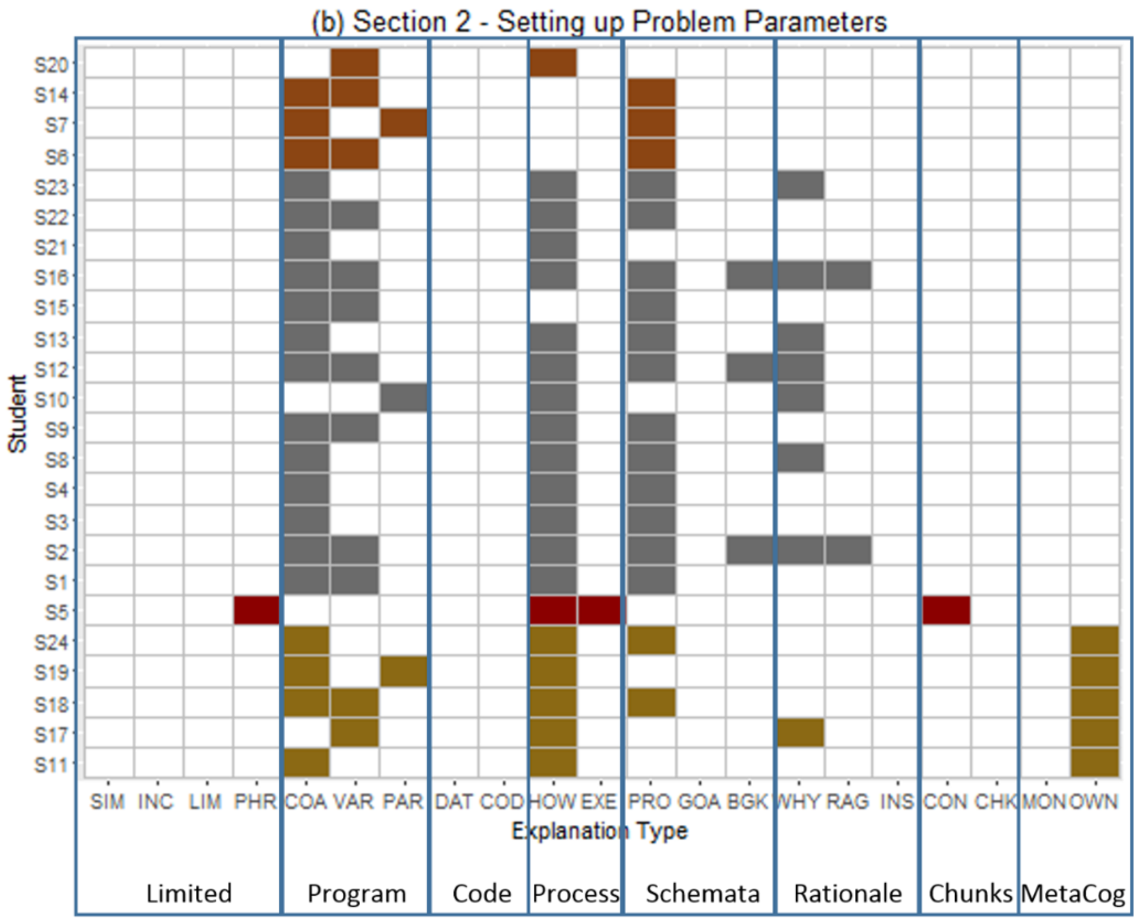


Figure 4.3. Sample analysis of students' explanations for section two of activity #5

When students showed incorrect or limited knowledge, we classified these instances using the common misunderstandings in programming (Table 2.1). These were be used to try to identify students' misunderstandings of the programming code from their written explanations (RQ3). Note that some important concepts were excluded because these were unrelated to this study. For instance, our examples do not include inheritance, or the design of tests, and therefore we do not expect to find them within students' explanations. The concepts listed on this table can potentially being identified within the examples explored in this study.

Open-ended Responses

An additional qualitative data source for this study corresponds to open-ended questions students answered either in an interview or at the end-of-the-semester survey. Six students from each semester participated in a retrospective think-aloud protocol aimed at identifying student experiences with: (1) the curricular innovation; (2) modeling and simulation practices to approach a disciplinary-computational project; and (3) instructional support they are provided with. The participants of these procedures kept an online journal while working on the project: Modeling Heart Tissue and Diffusion of the Electrical Potential. Once finished, they individually attended a 90 minute session to discuss their approach to the project. At the end of the interview protocol, these six students were specifically asked about their experiences and perceptions regarding the self-explanations. Sample questions for the last section of the interview protocol are:

Use of the examples

- Have you explored the examples along the semester?
- What do you think should be added to the examples to be more useful to you?

Self-explanations

- Did you write in-code comments in the examples for extra-credit? Why or why not?
- What do you think was the value of commenting the examples?

Quantitative Strand

The quantitative data comprises three main sources: students' usage of the worked-examples (RQ1), students' perceived ability to do computer programming, and students' performance in both midterms (RQ3).The course started with a pre-survey where students described the courses they had taken previously, and their perceptions about their ability to write computer programs. Two questions that were asked in this survey were: (1) [Likert Scale] I have the ability to design an algorithm; and (2) [Likert Scale] I have the ability to write a computer program.

Student performance on the midterm exams was considered as the student course performance. Both scores were normalized (0-10), and students were grouped as low-mid-high performers. The groups for these scores were created as: low below 6; mid between 6 and 8.5; and high above 8.5. The distribution for midterm one was as: six students were low performers, seven students were mid performers, and 11 students were high performers. Meanwhile, six students were mid performers and 18 students were high performers for midterm two. There were no low performers in midterm two. The rationale for using these scores instead of the overall course grade is that these were individual scores that were not affected by other components of the course. For instance, the overall course grade could have been higher for those students who decided to submit all the extra-credit assignment as compared as those who did not. Likewise, the collaborative nature of the projects, where students discussed ideas and troubleshoot together, may have had an effect that does not relate to their own ability to do computer programs.

Summary Methods for Glass Box Approach

The summary of the research procedures for the glass box context during spring 2015 and spring 2016 is depicted in Table 4.4. The main difference between the two semesters is that in 2015, the self-explaining activities were only implemented as extra-credit, starting in activity #9. On the other hand, activities #2 to #5 were graded and the rest of them were extra-credit for the spring semester 2016.

Table 4.4.

Summary of research procedures for the glass box context

Research Question	Semester	Data Source	Analysis
RQ1 Affordances	Spring 2014/ 2015/ 2016	Students' responses to open-ended questions	Thematic Analysis
		Students usage statistics of the worked-examples	Identification of trends and comparison among years
RQ2 Characteristics of students' explanations	Spring 2016	Students written explanations of activities #2, #5, and #11	Coding scheme from Table 4.3, and cluster analysis
RQ3 Relationship between students' explanations and student ability to program	Spring 2016	Survey scale questions regarding student self-perceived ability	Comparison of the characteristics of students explanations based on the performance measures
		Student performance in midterms #1 and #2	

4.3 Methods Case Two Black Box

The glass box approach to CSE education corresponds to a sophomore level course in thermodynamics of materials. These students were exposed to three computational modules in the spring semesters 2015 and 2016. The modules used the Virtual Kinetics of Materials Laboratory (VKML) to represent disciplinary phenomena. Although students had access to the Python code in VKML, this was considered a black box approach because students mostly interacted with the graphical user interface. Furthermore, the changes that they needed to make in the code only involved changing parameters, and most of the Python code was implemented under the functions of the GIBBS library. Thus, students did not have access to the underlying implementation.

The participants of this part of the study were students enrolled in a Thermodynamics of Materials (THERMO) course (45 students/semester), which is part of the Materials Engineering (MSE) program at Purdue University. Students in this program were first exposed to a general engineering first-year experience and then, they moved to the School of Materials Engineering (Purdue, 2014a). During the First Year Engineering (FYE) program, students were required to take the course CS 15900: Programming Applications for Engineers.

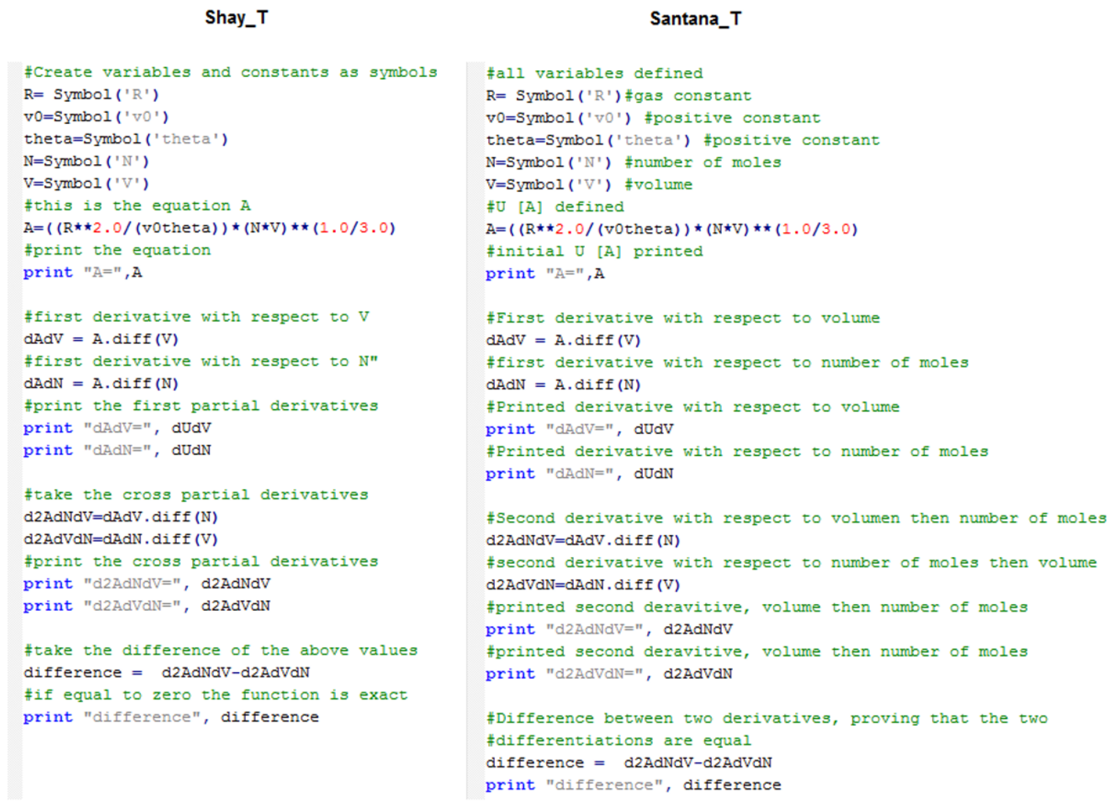
During the CS 15900 course, students worked with C and MATLAB as programming environments to solve engineering problems taking advantage of fundamental computational methods and concepts. The FYE program also required students to enroll ENGR 13100: Transforming Ideas to Innovation I, and ENGR 13200: Transforming Ideas to Innovation II (Purdue. 2014c, 2014d). Although not specifically focused on computational concepts, these courses deal with some MATLAB programming skills such as data structures, decision and loop clauses, and transforming flowcharts to algorithms. Therefore, when students got to the thermodynamics course, they were expected to understand basic programming structures but not necessarily to be familiar with the Python syntax.

4.3.1 Procedures, Data Collection and Data Analysis

The THERMO course started with a baseline test to identify what is the students' previous knowledge in Chemistry, Calculus, and programming background. Three 50-minute computational modules were implemented as part of the course in 2015 and 2016. A sample module is described in Appendix B. Each module started with a ten-minute pretest regarding thermodynamics concepts. The students then explored a computational worked example guided by the course instructor. As part of a homework assignment, students were asked to write in-code comments to explain the example. The students submitted their homework assignment one week later. At this point, students completed the posttest intended to assess changes in their understanding of the thermodynamics concepts.

Qualitative Strand

As in case study one (i.e., Glass Box CPMSE), students in THERMO had the possibility of commenting the code of the given example for extra credit. Figure 4.4 depicts sample self-explanations that comprise the first THERMO module described in Appendix B. The worked example was intended to evaluate whether a given equation was a state function or not. Here we compare the self-explanations provided by Shay_T and Santana_T as in-code comments for this example.



Shay_T

```
#Create variables and constants as symbols
R= Symbol('R')
v0=Symbol('v0')
theta=Symbol('theta')
N=Symbol('N')
V=Symbol('V')
#this is the equation A
A=((R**2.0/(v0theta))*(N*V)**(1.0/3.0)
#print the equation
print "A=",A

#first derivative with respect to V
dAdV = A.diff(V)
#first derivative with respect to N"
dAdN = A.diff(N)
#print the first partial derivatives
print "dAdV=", dUdV
print "dAdN=", dUdN

#take the cross partial derivatives
d2AdNdV=dAdV.diff(N)
d2AdVdN=dAdN.diff(V)
#print the cross partial derivatives
print "d2AdNdV=", d2AdNdV
print "d2AdVdN=", d2AdVdN

#take the difference of the above values
difference =  d2AdNdV-d2AdVdN
#if equal to zero the function is exact
print "difference", difference
```

Santana_T

```
#all variables defined
R= Symbol('R')#gas constant
v0=Symbol('v0') #positive constant
theta=Symbol('theta') #positive constant
N=Symbol('N') #number of moles
V=Symbol('V') #volume
#U [A] defined
A=((R**2.0/(v0theta))*(N*V)**(1.0/3.0)
#initial U [A] printed
print "A=",A

#First derivative with respect to volume
dAdV = A.diff(V)
#first derivative with respect to number of moles
dAdN = A.diff(N)
#Printed derivative with respect to volume
print "dAdV=", dUdV
#Printed derivative with respect to number of moles
print "dAdN=", dUdN

#Second derivative with respect to volumen then number of moles
d2AdNdV=dAdV.diff(N)
#second derivative with respect to number of moles then volume
d2AdVdN=dAdN.diff(V)
#printed second deravitive, volume then number of moles
print "d2AdNdV=", d2AdNdV
#printed second deravitive, volume then number of moles
print "d2AdVdN=", d2AdVdN

#Difference between two derivatives, proving that the two
#differentiations are equal
difference =  d2AdNdV-d2AdVdN
print "difference", difference
```

Figure 4.4. In-code comments of Shay_T and Santana_T for the first THERMO module.

Santana_T described what each variable stands for in terms of thermodynamics variables (e.g., volume, number of moles, gas constant). Shay_T did not describe all the steps or variables and, when she/he did, the comments were not too explicit. Interestingly, neither of them mention the result as being a state function. Shay_T mentioned that the “function is exact” while Santana_T said that the “differentiations are equal”.

These differences in student in-code comments were analyzed using the coding-book described in Table 4.3. Then, a hierarchical cluster analysis was carried out as described in section 5.3.2.1. Students were grouped as different explainers based on the types of explanations they submitted for each of the modules.

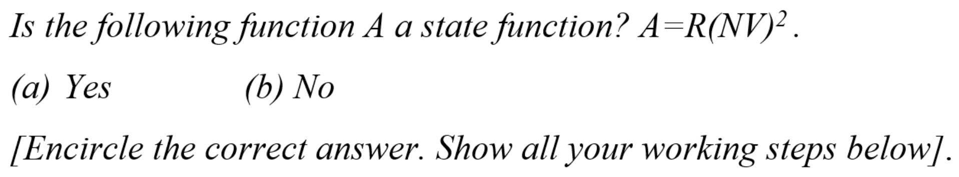
The second qualitative component corresponded to students’ responses to a set of open-ended questions. At the end of the semester, students completed a survey focused on identifying the change on students’ perceptions about computation, and students’ comments about the self-explanation activities. Two of the questions that were used as qualitative data source for RQ1 are: (a) Please include any additional ideas in which you think that commenting the examples or the lab sessions supported your learning process in this course (b) What would you improve about the Python lab sessions?

Six students were invited to participate in a retrospective think aloud protocol aimed at identifying student computational problem solving process after being exposed to worked examples and self-explanations. However, only three students participated of the interview in 2015, and none in 2016. At the end of the interview, students asked specific questions about the worked examples and the self-explanation process such as (1) How did the worked examples help you to solve your problem? (2) Did you write in-code comments for the extra credit? How did that process help or hinder your understanding of the examples? (3) What do you think should be added to the examples to be more useful to you?

Quantitative Strand

The quantitative strand for the THERMO course involved several data points. At the beginning of the semester, students completed a survey that asked two Likert scale questions: (1) I have the ability to design an algorithm; and (2) I have the ability to write a computer program. Students' responses for these two questions were averaged to compute a composite score related to the self-perceived ability to do computer programming. This score was used to identify whether student initial ability to do computer programming was related to the types of explanations they write (RQ3).

Pretest/posttest instruments were used before and after the computational module to assess student disciplinary conceptual change. The concepts that were tested had been explored during the lectures previous to the module. Students were awarded 5 extra points for every correct instrument (i.e., pretest/posttest) they completed. For instance, the pretest instrument for the first module consisted of a single question, as depicted in Figure 4.5.



Is the following function A a state function? $A=R(NV)^2$.

(a) Yes (b) No

[Encircle the correct answer. Show all your working steps below].

Figure 4.5. Sample pretest for Module # 1

The posttests took place one week after the computational module, when the homework was due. The posttest instruments were similar to the pretest but involving different values/equations.

The pretest/posttest instruments were scored by the teaching assistant of the course in a scale 0-5. The gain for each student was computed as posttest -pretest. Furthermore, the gain score was employed to group students as low, mid, and high performers, and compare these performances to their written explanations. A low gain score was considered to be zero or less, an intermediate gain comprised values between one and two, and a high gain score corresponded to values of three or more.

Then, at the end of the semester, students were asked to answer three five-level Likert scale questions related to the in-code comments activities in the form : “I feel writing comments within the sample code helped me to”: (1) understand the Python examples; (2) solve the homework exercises; (3) understand better thermodynamics concepts. The distribution of students’ responses to these three questions were compared using a non-parametric Wilcoxon rank test.

Summary Methods for Black Box Approach

The summary of the research procedures for the black box context during spring 2015 and spring 2016 is presented in Table 4.5. Note that the self-explanation activities in 2015 were not graded but students were granted extra credit for them. Moreover, students in 2015 could submit them in groups. This changed in 2016, were the homework assignments became individual, and students’ written in-code comments were graded.

4.4 Validity and Reliability

The procedures that ensured the reliability of the qualitative data analysis of students’ explanations were as follow. First, one researcher coded the whole data set to identify the possible categories within the students’ explanations. This researcher then compared the coded sections across students looking for consistency among different data points. A second researcher was introduced with the coding scheme as well as sample explanations within the different activities for each code (see Appendix C). This researcher completed a qualitative coding for a random sample of 20% of students’ explanations for each worked-example. The two researchers compared their coded data and negotiated discrepancies to refine the coding

Table 4.5.

Summary of research procedures for the black box context

Research Question	Semester	Data Source	Analysis
RQ1 Affordances	Spring 2014/ 2015/ 2016	Students' responses to open-ended and Likert-scale questions	Thematic Analysis / Distribution and Wilcoxon Rank-test
		Number of submitted self-explanations	Identification of trends and comparison among semesters
RQ2 Characteristics of students' explanations	Spring 2016	Students written explanations of modules #1, #2, and #3	Coding scheme from Table 4.3, and cluster analysis
RQ3 Relationship between students' explanations and student ability to program	Spring 2016	Survey scale questions regarding student self-perceived ability	Comparison of the characteristics of students explanations based on the performance measures
		Student performance pretest/posttest instruments	

scheme. The first researcher then refined the rest of the qualitative coding process based on the new coding scheme.

For the open-ended responses, one of the researchers first reviewed all student responses and assigned an emerging category to each response. Then, a second researcher reviewed 50% of the responses. The reviewer was provided with the set of categories and definitions to use while coding. If the second researcher considered that additional categories were required, they were allowed to create them. The two researchers then negotiated their categories until agreement. Finally, the first researcher went through the whole data set again to re-code the quotes using the refined set of categories.

4.5 Research with Human Subjects

The research procedures presented for this dissertation were approved by the Instructional Review Board under the protocols #1308013870 and #1412015574. Consent forms were be signed by the interview participants and the participants were allowed to withdrawal at any time. These consent forms as well as all data collection instruments were kept by the principal investigator.

Student identity and confidentiality was kept throughout the study. Student identity was not, and will not be revealed to any of the course instructors or teaching assistants. Gender neutral pseudonyms replaced student names.

4.6 Summary

This dissertation explored the characteristics of students' written explanations in two courses using different levels of transparency to teach CSE concepts. The first course corresponded to a programming course for materials scientists and engineers (CPMSE) where students mostly focused on algorithm design and mathematical models to solve disciplinary problems. Because students had access, and often implement the underlying mechanisms of the simulations, this was considered a glass box approach to CSE education. The second course was a thermodynamics of materials course (THERMO) where students used VKML to manipulate computational representations of disciplinary phenomena. Although these students had access to the Python code that implements these representations, this was considered the black box approach because the simulations mostly used encapsulated functions from VKML, and so the underlying mechanisms were not actually visible. Each course was treated as an independent case study, and the two cases were compared to each other in the discussion (Chapter 7).

Three elements were explored within each case to understand the characteristics of students' explanations in CSE (Figure 4.6): (1) the affordances of in-code comments for students; (2) the characteristics of students' explanations; and

(3) the relationship between students' explanations and student ability to program. First, the affordances in-code commenting activities for students were explored from open-ended responses to interview and survey questions. We also used the usage statistics of the worked examples to identify changes on students' engagement with the worked-examples after the self-explanation activities were implemented. Then, students' in-code comments were qualitatively analyzed using the coding book described in section 5.2. The presence or absence of each of these codes within students' explanations was used to group students based on a hierarchical clustering technique. Finally, the characteristics of these explanations and the groups of explainers were compared to students' ability to do computer programming. We employed survey questions and performance measures to assess this construct. These three elements were connected to each other in the figure because as part of the discussion, we aimed to identify whether there was relationship between them or not.

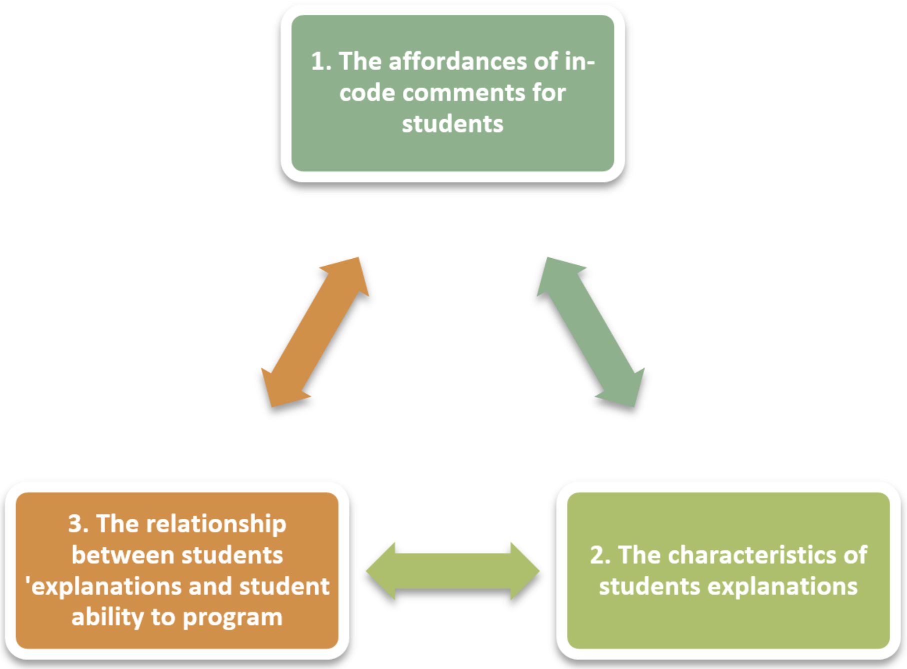

1. The affordances of in-code comments for students
2. The characteristics of students explanations
3. The relationship between students 'explanations and student ability to program

Figure 4.6. Three main elements of students' explanations explored in this study

The next chapter presents the detailed procedures and results for case one: glass box CPMSE. Then Chapter 6 describes the procedures and results for case two: black box THERMO. Chapter 7 presents a discussion of the findings in both cases to identify commonalities and differences under the lens of explanation literature. Finally, Chapter 8 presents the conclusions of this dissertation, and proposes future work to expand on the findings of this study.

CHAPTER 5. GLASS BOX APPROACH

The first case study that we explored corresponded to a programming course for materials scientists and engineers (CPMSE). The learning goals of this course focused on students' ability to create computer programs to solve disciplinary problems. Students often need to start from understanding the disciplinary problem, identify or design a mathematical model, and implement an algorithmic representation of this model. Students in this context had access and often created themselves the underlying mechanisms of these simulations. Hence, this level of transparency was denominated as the glass box approach to CSE education.

This chapter is divided in three main sections plus a final section summarizing the findings for this context. Section 5.1 focuses of the affordances of in-code commenting activities students in this course: RQ1. Section 5.2 identifies the characteristics of students' explanations and the types of explainers in this context: RQ2. Section 5.3 compares the types of explainers and the characteristics of students' explanations to students' ability to do computer programming: RQ3. Each section defines its own questions that contribute to answer the overall research questions of the study. These sections also describe in more detail than Chapter 4, the specific procedures for each research question.

5.1 Affordances of in-code commenting activities for students

The worked-examples were first implemented into the CPMSE course during the spring semester 2014. They were accessible through a website including a problem statement, a description of the solution, and a MATLAB® commented code. The different components were presented to the students using multiple representations: video explanations, graphics, text, mathematical equations, and programming code. There was no course credit associated with the exploration of the examples, nor was there a self-explanation strategy associated with this implementation. At the end of the semester, students claimed to be unaware of their existence, and the usage statistics of the examples were nearly zero.

As a consequence, the implementation strategy for the worked-examples was refined for future offerings of CPMSE course (spring semesters in 2015 and 2016). Each of the worked-examples was posted as a PDF document on the learning management system (Blackboard). The document was organized in four sections: (1) problem statement; (2) understanding the problem (3) addressing the problem; and (4) an uncommented MATLAB® solution. During spring 2015, for in-class activities starting with number nine and up through number fifteen 10-points of extra-credit (out of 1000 total) could be earned by those students who submitted in-code comments as a self-explanation strategy. During spring 2016, the in-code comments for in-class activities number two to number five constituted part of the students' class grade; students could additionally earn five-extra credit points for submitting the explanations and completing a quiz evaluating their understanding of the example. The instructions for the assignment were:

1. Please download the attached worked-out example.
2. Insert comments at the code segment of the example to explain what the code is doing.
3. Upload a MATLAB® (.m) file with the comments you made.
4. In order to receive CREDIT, please complete the Exercise 02 Worked example Quiz once you have submitted the commented file

After the changes the way we implemented the worked-examples in this course, the first part of this study was to identify what was the effect of this changes for the students in this course. Specifically, we looked into the effect of these self-explanation activities on student' engagement with the example, and the different ways students afforded these activities. The guiding research questions for this component of the study are:

- What is the effect of using in-code commenting activities on students' engagement with the worked-examples in the context of a glass box approach to computational science and engineering?
- What are affordances of in-code commenting self-explanation activities in the context of black box and glass box approaches to computational science and engineering?

5.1.1 Data Collection

The data sources for this part of the study involved three main components. First, the usage statistics of the worked-examples during the spring semesters in 2014, 2015, and 2016. These helped us to identify the effectiveness of in-code comments as the self-explanation strategy to engage students in the study of worked-examples. These statistics comprise both, the number of views of the video explanations within the examples, and the number of extra-credit assignments submitted. Note that the students were not required to watch the videos accompanying the examples. This was only one of the representations, but some students might have chosen to study the other representations (e.g. text and graphics, MATLAB® code).

At the end of the semester, students completed a survey that asked them about the self-explanation activities, whether they chose to complete them or not, and why. In addition to this open-ended question, students answered two five-level Likert scale questions: (1) Writing comments within the sample code helped me understand the examples; (2) Writing comments within the sample code helped me to solve the projects. Finally, six students per semester were invited to participate in a retrospective think-aloud protocol in which they described how they solved one of the course projects, and what resources they used. Both the survey and some of the interview questions were employed to identify in-code comments' affordances for students in a glass box approach to CSE.

5.1.2 Data Analysis

The usage statistics data from the video explanations, and the number of extra-credit submissions were compared among the three semesters: spring 2014, spring 2015, and spring 2016. In order to make it comparable across years, the number of views for the worked-example videos, and the number of extra-credit submissions are presented as a percentage of the number of students enrolled in the course that semester. Additional trends, such as usage peaks for certain activities at a given year are described and explained in the context of that course offering.

The open-ended questions and the interview responses were analyzed using categorical analysis. This approach allows us to identify common student perceptions about the worked-examples and the self-explanation activities, as well as their affordances for students in this context. The two Likert-scale questions were first plotted as a histogram to identify trends in students' level of agreement. A non-parametric Wilcoxon Rank Test was employed to compare the distribution of the two questions.

5.1.3 Results

The percentage of students submitting in-code commenting assignments during spring 2015 and 2016 are presented in Figure 5.1. Activities #2 to #5 were graded during spring 2016, and as a result the number of student submissions approaches 100%. Once the extra-credit assignments started, both in 2015 (activity #9) and in 2016 (activity #6), the number of students who submitted their explanations decreased over time. Interestingly, the number of submissions stabilized in 2016 between activities #8 and #9.

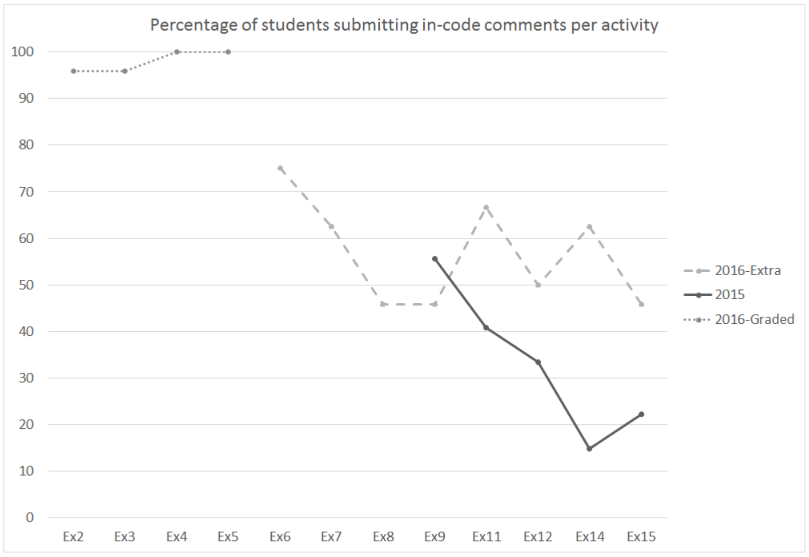


Figure 5.1. Percentage of student submissions of in-code comments assignments

The percentage of students watching the video-explanations from the worked-examples for 2014, 2015, and 2016 are presented in Figure 5.2. Although the videos were available for most of the examples (11 out of 14), students were not required to watch them. The videos were an additional representation besides the text, graphics, and MATLAB® code. Hence, most of the percentages do not reach 100%, even in 2016 for the graded activities (number #3 and #4).

The number of views by students of the videos within the examples was larger for both spring 2015 and 2016, as compared to 2014. During the first part of the semester (i.e., activities #3 to #9), the number of views was higher for 2016 compared to 2015. Note that the extra-credit assignments in 2015 started from in-class activity number nine. Consequently, the number of views showed a boost from this activity onwards during 2015. In activity #9, students should have watched the videos more than once, since the percentage was 185.2%, above the maximum possible value of 100%. For visualization purposes in Figure 5.2, we adjusted the percentage to the maximum value of 100%. This result suggests that

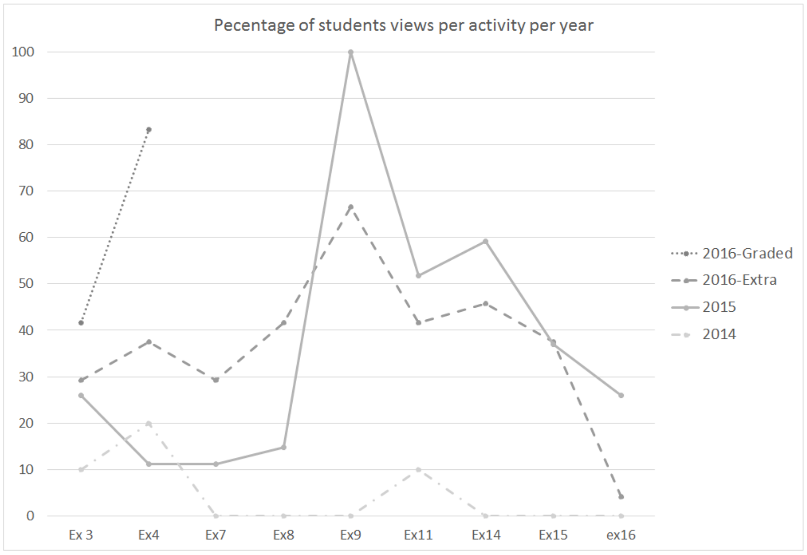


Figure 5.2. Percentage of students who watched the video explanations of the worked-examples

promoting in-code comments as extra-credit assignments is an effective strategy to increase student engagement in studying the examples.

At the end of the semester, students voluntarily completed a survey that asked them whether they used the worked-examples or not, and why. In 2015, 13 students completed the survey; 11 students answered that they had used them, and only two students said that they did not use them. The students who did not use them argued that they did not feel they needed the extra-credit nor did they see value in exploring the examples; therefore, they preferred to spend more time on the course projects. The rest of the students provided the following reasons for using the worked-examples: self-explaining helped to understand the examples (five students); the worked-examples helped to solve other problems (three students); and self-explanation activity gave them extra-credit (three students).

On spring semester 2016, 21 students completed the survey question regarding the use of worked-examples. Three students said they did not use them, while 18 students said they did. Two of the students who did not use them said they did not need them, while the other student did not provide any reason for not using them. These three students must have referred to the extra-credit ones, since the number of submissions showed that up to 100% of students completed the graded activities. Eleven students said that the worked-examples helped them to better understand difficult concepts, four students said they did the activity mostly because of the extra credit, and three students said they used them, but did not give any reasons.

A representative quote from one student who found the worked examples to be useful was:

> "Yes, all the time. Especially in the earlier stages of the course. Dissecting the worked examples line-by-line really helped with gaining an intuition for algorithm design and knowing how to think like a naïve computer."

Another student said:

> "Yes. They helped me transition from Java to vectorized MATLAB® code. They also helped guide my coding to more efficient and abbreviated writing. Also, the worked examples solved some questions I had."

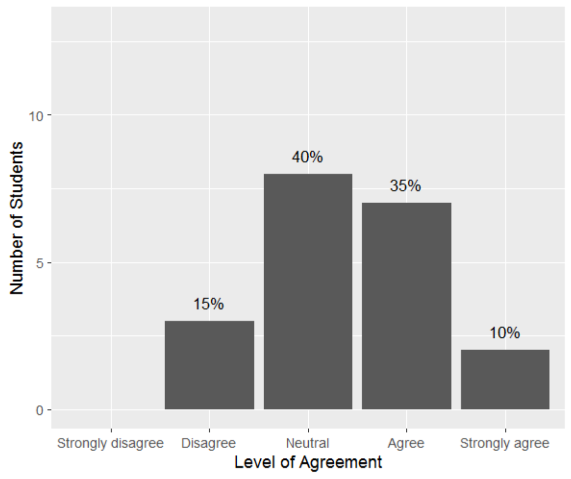
During the interview, six students were invited to explain how they completed one of the projects and what resources they used. Within the resources, they were explicitly asked about the worked-examples and the extra-credit activity. A table in appendix D includes the complete student responses to this question. In 2014, only five students participated in the interview. There were not self-explanation activities and, as described above, most of the students said they were unaware of the existence of the worked examples. Those students who accessed at least one of the worked-examples during the semester, thought they were confusing and stopped exploring them.

Extra-credit self-explanation activities were introduced starting on in-class activity #9 during spring 2015. Five out of the six interviewed students said they did these activities. Three of the students said the activity engaged them to go line by line through the MATLAB® code, which helped them to better understand the examples and learn new functions. Two students said it was useful because it forced them to explore the exercises that were not discussed in class, and one of the students also mentioned that it was a good practice for them to improve their commenting skills.

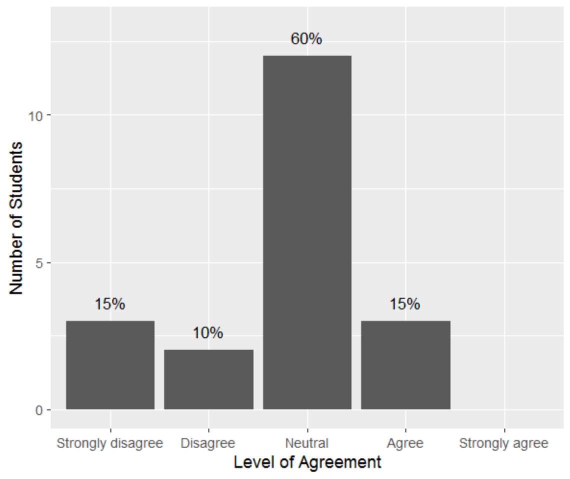
Spring 2016 involved graded extra-credit assignments for in-class exercises two to five, and extra-credit ones for the rest of the exercises. As a consequence, all students had explored some of the worked-examples, and had done at least a few of the commenting activities. Five out of six students considered these activities as helpful to better understand the example. However, three of them confessed that they did them mostly for the extra-credit. One student said it was useful at the beginning of the class to get familiar with the MATLAB® syntax, since he/she already had some programming experience. One of the students considered that these activities enabled him or her to connect the individual lines of code with the overall goal of the examples:

> "... it made you actually engage with the program on a very detailed level you have to actually say, All right, what is this line actually doing and how does that relate to sort of the overall functionality that you're trying to implement?' ".

The two Likert-scale questions asking to what extent writing comments within the examples had helped them to either understand the examples or solve the projects were only asked in 2016. The distribution of student responses are presented in Figure 5.3. The measures of central tendency associated with each question are: (1) Understanding the worked-examples (mean=3.48, median=3, standard deviation= 0.93, N=21); and (2) Solving the Projects (mean=2.75, median=3, standard deviation= 0.91, N=20). While a large portion of the students agreed that writing comments within the code helped them to better understand the worked-examples (a), a rather neutral response was found for the benefits that the worked-examples provided in competing the projects (b). The non-parametric Wilcoxon rank test suggests that the distributions for both questions are significantly different (Z-value=3.28, p-value <0.01, effect-size r=1).



(a) Writing Comments within the sample code helped me understand the examples



(b) Writing Comments within the sample code helped me to solve the projects

Figure 5.3. Student distribution on the five-level Likert scale questions for 2016

Overall, providing extra-credit activities for writing in-code comments helped students to be aware of the existence of the worked-examples, and engaged students to study them. The number of extra-credit submissions decreased after a few exercises, but stabilized at approximately fifty percent. We hypothesize this can be attributed to the course-load students may have with the CPMSE projects and other assignments.

Writing in-code comments within the MATLAB® code helped students to better understand the worked-examples, connecting the individual lines of code to the overall goal of the example. These activities also helped students to better understand algorithm design, how the computer works along with other difficult concepts, as well as to getting familiar with the MATLAB® syntax, especially for those students who already had some programming experience. Although some of the students did not feel that commenting their code helped them to solve the course projects, several students said that the activity engaged them enough to learn new algorithms, syntax, and programming skills that they could use in other problems.

5.2 Characteristics of Students' Explanations

After identifying that students benefited in different ways from writing explanations, the next step was to identify what were the characteristics of these explanations. Twenty-six students enrolled in the CPMSE course during the spring semester 2016 participated in this part of the study. There were 16 in-class activities, of which 15 included one worked-out example describing an expert solution to one of the programming challenges. The guiding research questions to characterize students' written explanations are:

- What are the characteristics of students' explanations in a glass box approach to CSE education?
- How does the characteristics of students' explanations in a glass box approach to CSE education change over time?
- What common misunderstandings in programming can be identified from students' explanations in a glass box approach to CSE education?

5.2.1 Data Collection

For the scope of this dissertation, students' explanations of three examples were explored in the glass box approach: Activity #2, Activity #5, and Activity #11. These examples were purposefully selected to comprise a variety of complexity and a distribution from the beginning to the end of the semester. The first example that was analyzed corresponded to the first activity in which students submitted explanations during this semester. The second worked-example corresponded to the activity #5, which involved the concept of loops for the first time within an example, a challenging concept in programming. The third worked-example that was explored corresponded to activity #11 and also included complex programming concepts: nested loops and a sparse matrix.

Each section within the sample codes was assigned a section type that described their purpose. Thus, we were able to compare the characteristics of students' explanations among different section types within a worked example, as well as students' comments for the same type of sections from different worked examples. The section types that were defined for these worked-examples are: (1) "Creating the Function" when a function is been declared; (2) "Setting up problem parameters" when a value related to the problem is computed; (3) "Setting up Supporting Variables" when other variables that are used to solve the problem are computed; (4) "Validating the result" when the result is being computed and sometimes printed for validation; (5) Iterating when a loop structure starts; (6) Validation when an if-clause starts; and (7) "End of the function" when the function is being closed by an end statement.

Activity #2

The purpose of the first worked example was to create a sequence of steps within a function that could compute the length of the side c2 for the geometric figure presented in Figure 5.4, given the lengths b1, b2 and c1, and the angles A1 and A2. The example introduced the Law of Cosines and described how the solution could be approached.

The code consisted of eight lines of code divided in five sections (see Figure 5.5). The first section was the declaration of the function, its parameters and returned values. The second section applied the Law of Cosines to find the length of the side a, a problem parameter. The third section identified the coefficients of a quadratic equation (i.e., supporting variables) to then find out the two possible

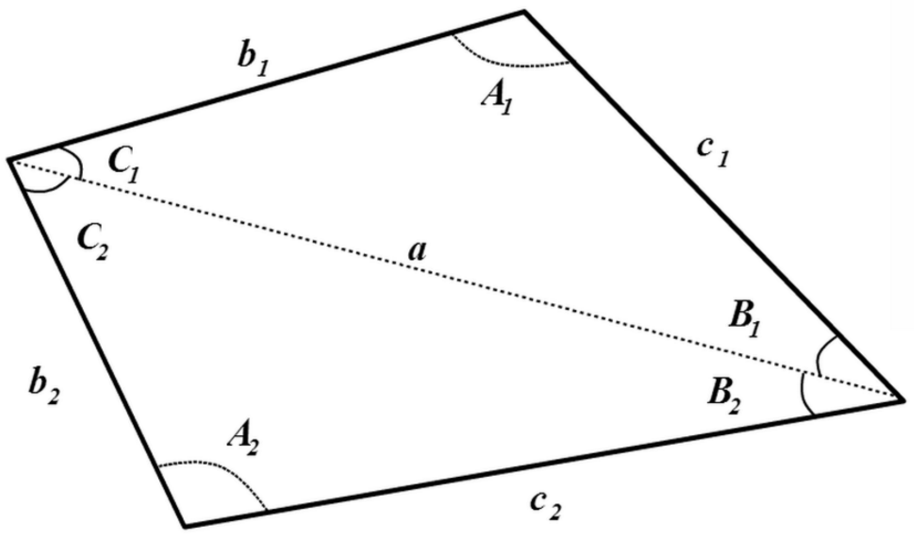
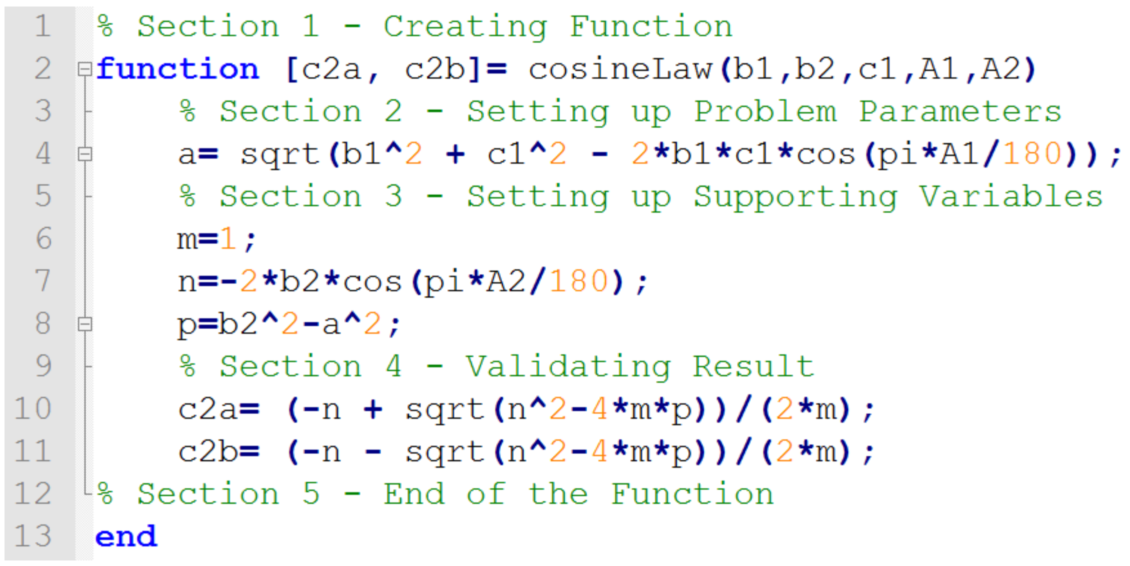


Figure 5.4. Geometric figure for the example in activity #2

solutions on section four: validating the results. The final section closed the function with an end' command. Students were not given these sections distinguished from each other, nor their descriptions, but some of their explanations demonstrated that students were able figured out the sections themselves.

Activity #5

Activity #5 was the last graded assignment in which students needed to write their comments to explain the worked examples. The rest of the in-class activities involved extra-credit assignment for this purpose. The goal of activity #5 was to have students exposed to the concept of loops and user inputs. The problem statement required students to ask for a number to the user and continue asking until the most recent input corresponded to the smallest number so far. The returning value should be the largest inputted value. Figure 5.6 presents the sample code that was provided to the students with the sections marked as identified during the data analysis process. The first section defined the function with no input parameters, and the biggest number as an output. Section two requested the inputs



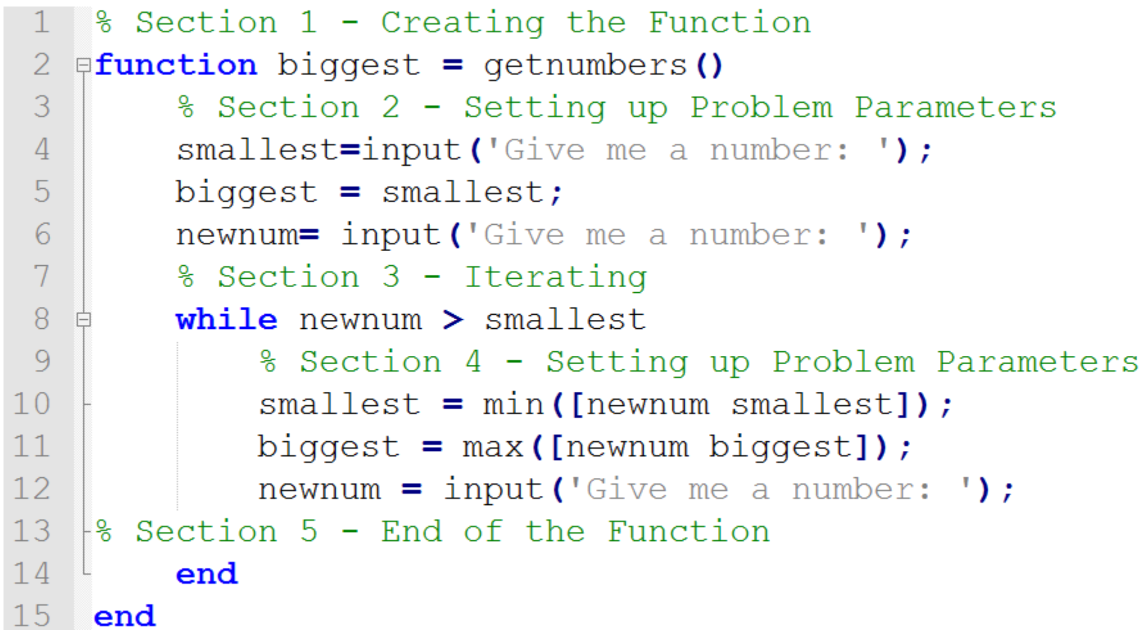
```
% Section 1 - Creating Function
function [c2a, c2b]= cosineLaw(b1,b2,c1,A1,A2)
    % Section 2 - Setting up Problem Parameters
    a= sqrt(b1^2 + c1^2 - 2*b1*c1*cos(pi*A1/180));
    % Section 3 - Setting up Supporting Variables
    m=1;
    n=-2*b2*cos(pi*A2/180);
    p=b2^2-a^2;
    % Section 4 - Validating Result
    c2a= (-n + sqrt(n^2-4*m*p))/(2*m);
    c2b= (-n - sqrt(n^2-4*m*p))/(2*m);
% Section 5 - End of the Function
end
```

Figure 5.5. MATLAB code of worked-example activity #2: CosineLaw

from the user an initialize the variables related to the problem statement (i.e., smallest and biggest). The third section comprised a loop structure that only stops when a smaller number has been entered. Section four compared the new input with the current values, and requested an additional new input. The final section closed the while-loop structure and the function.

Activity #11

Activity #11 involved a worked-example called Atomic Bonds. The purpose of the activity was to identify which ones of a set of atoms where within certain cutoff distance of the rest of them in a three-dimensional space. The input for this function is the cutoff distance an Nx3 matrix where each row corresponds to an atom, and the three columns represent the position vector X, Y, Z of this atom. The output is a matrix with the distances between the atoms that satisfied the cutoff condition, or zero otherwise. The output should be a sparse matrix for

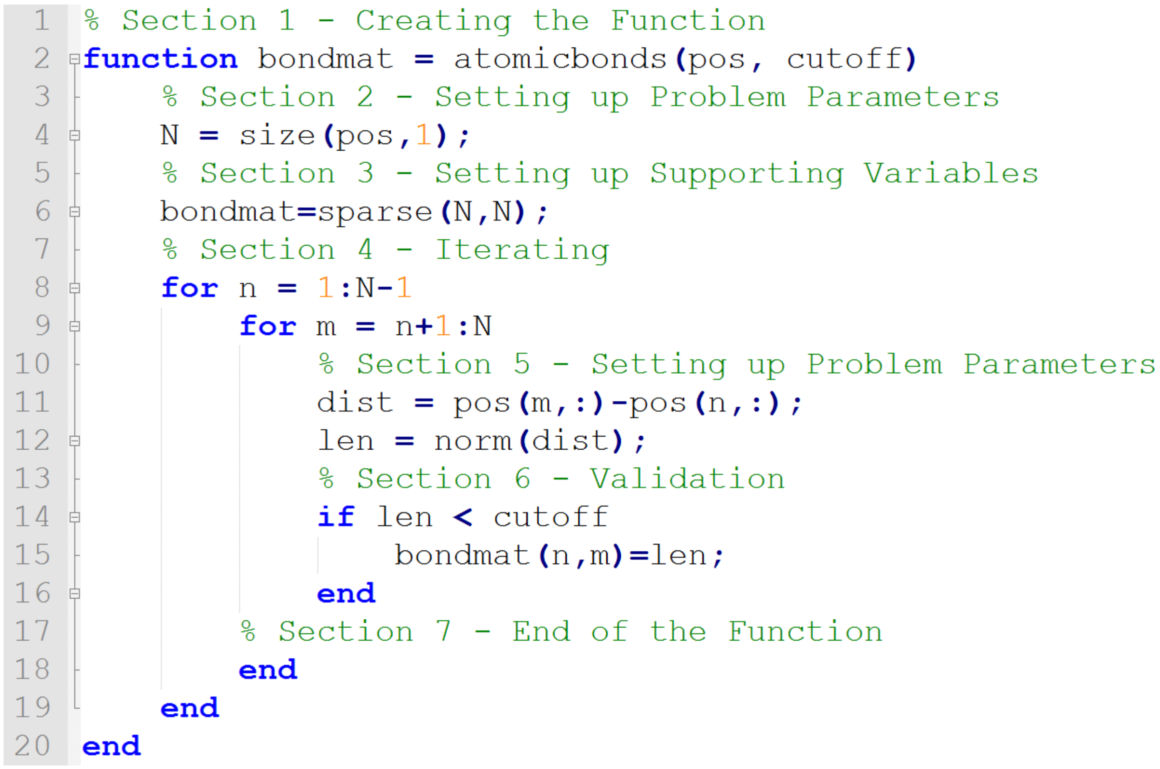


```
% Section 1 - Creating the Function
function biggest = getnumbers()
    % Section 2 - Setting up Problem Parameters
    smallest=input('Give me a number: ');
    biggest = smallest;
    newnum= input('Give me a number: ');
    % Section 3 - Iterating
    while newnum > smallest
        % Section 4 - Setting up Problem Parameters
        smallest = min([newnum smallest]);
        biggest = max([newnum biggest]);
        newnum = input('Give me a number: ');
% Section 5 - End of the Function
    end
end
```

Figure 5.6. MATLAB code of worked-example activity #5 - Get Numbers

efficiency reasons, since the output would include multiple zeros and repeated values (i.e., the upper and lower triangular parts of the matrix are the same).

Figure 5.7 presents the MATLAB ® code that the students submitted explanations for. The first section created the function with the atom-position matrix pos, and the cutoff distance. Section two identified the number of atoms (a problem-related parameter) and section three initializes the output sparse matrix. The two loop structures in section four were designed so that each atom is only compared once to the rest of them. Section five found the Euclidian distance between the two vectors by first subtracting them, and then normalizing this vector. The sixth section validated the cutoff condition and saves the distance in the output sparse matrix.



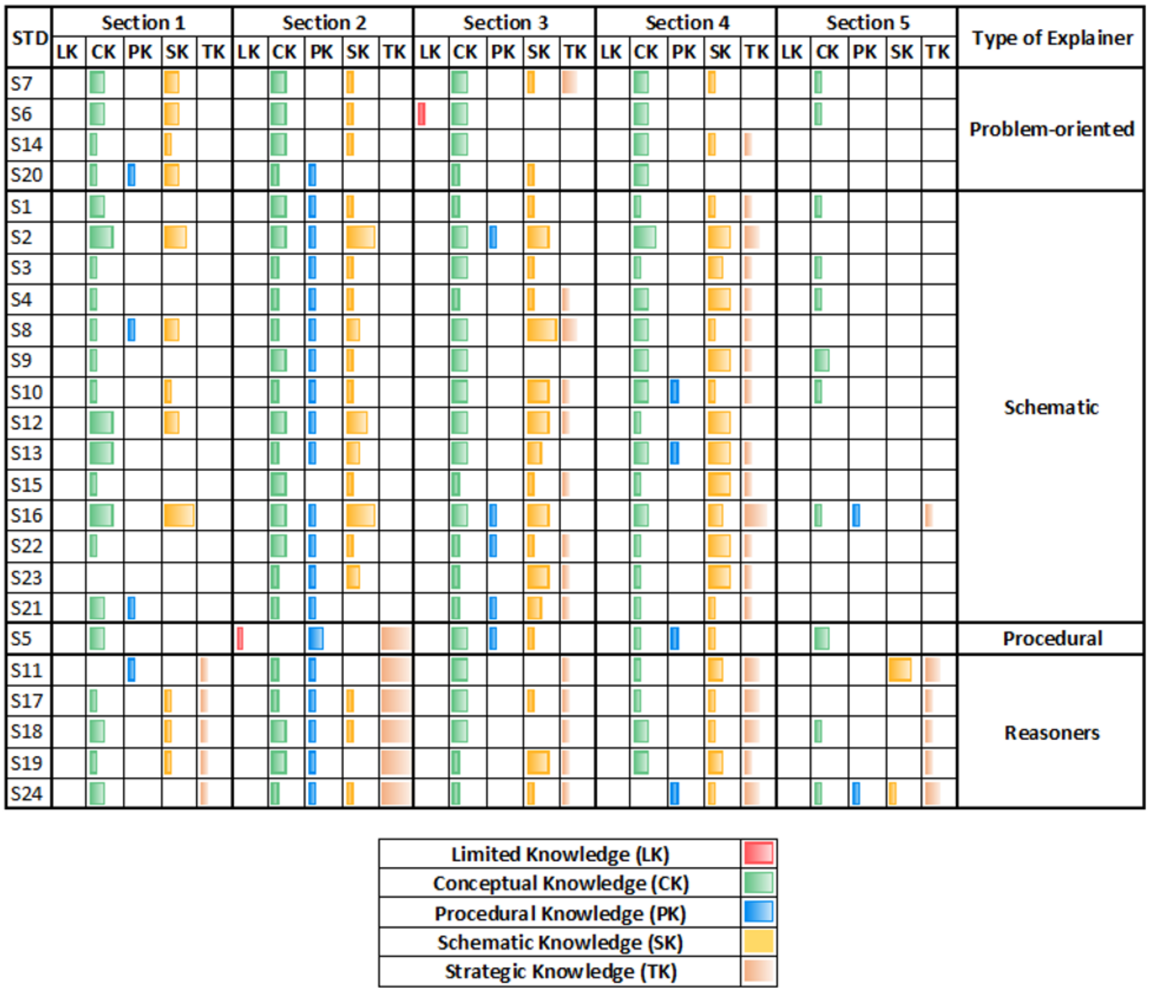
```
% Section 1 - Creating the Function
function bondmat = atomicbonds(pos, cutoff)
    % Section 2 - Setting up Problem Parameters
    N = size(pos,1);
    % Section 3 - Setting up Supporting Variables
    bondmat=sparse(N,N);
    % Section 4 - Iterating
    for n = 1:N-1
        for m = n+1:N
            % Section 5 - Setting up Problem Parameters
            dist = pos(m,:)-pos(n,:);
            len = norm(dist);
            % Section 6 - Validation
            if len < cutoff
                bondmat(n,m)=len;
            end
        % Section 7 - End of the Function
        end
    end
end
```

Figure 5.7. MATLAB code of worked-example activity #11 - Atomic Bonds

5.2.2 Patterns within each activity

Activity #2

Twenty-four students submitted their explanations for this activity. The distribution of students' use of the different types of knowledge within each section is presented in the Figure 5.8. The most commonly used type of knowledge throughout all sections was the declarative knowledge. Students used procedural knowledge mostly in section two, and schematic and strategic knowledge more often in sections three and four, although one group of students used them also to explain section two. Only two students showed limited knowledge within their explanations and in different sessions. Eight students used the four types of knowledge (CK, PK, SK, and TK) within the same section: three in section two, three in section three, and two students did in section four. One student (S24) used these four types of knowledge again in section five, but this section was not considered for the cluster analysis. The rationale for this decision was that section five only represented the end of the function, and the only rich explanations within this section corresponded to the five students who actually created their own solution to the problem.



STD	Section 1					Section 2					Section 3					Section 4					Section 5					Type of Explainer
	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	
S7		🟩		🟨			🟩		🟨			🟩		🟨	🟧		🟩		🟨			🟩				Problem-oriented
S6		🟩		🟨			🟩		🟨		🟥	🟩					🟩					🟩				
S14		🟩		🟨			🟩		🟨			🟩					🟩		🟨	🟧						
S20		🟩	🟦	🟨			🟩	🟦				🟩		🟨			🟩									
S1		🟩					🟩	🟦	🟨			🟩		🟨			🟩		🟨	🟧		🟩				Schematic
S2		🟩		🟨			🟩	🟦	🟨			🟩	🟦	🟨			🟩		🟨	🟧						
S3		🟩					🟩	🟦	🟨			🟩		🟨			🟩		🟨	🟧		🟩				
S4		🟩					🟩	🟦	🟨			🟩		🟨	🟧		🟩		🟨	🟧		🟩				
S8		🟩	🟦	🟨			🟩	🟦	🟨			🟩		🟨	🟧		🟩		🟨	🟧						
S9		🟩					🟩	🟦	🟨			🟩					🟩		🟨	🟧		🟩				
S10		🟩		🟨			🟩	🟦	🟨			🟩		🟨	🟧		🟩	🟦	🟨	🟧		🟩				
S12		🟩		🟨			🟩	🟦	🟨			🟩		🟨	🟧		🟩		🟨							
S13		🟩					🟩	🟦	🟨			🟩		🟨			🟩	🟦	🟨	🟧						
S15		🟩					🟩		🟨			🟩		🟨	🟧		🟩		🟨	🟧						
S16		🟩		🟨			🟩	🟦	🟨			🟩	🟦	🟨			🟩		🟨	🟧		🟩	🟦		🟧	
S22		🟩					🟩	🟦	🟨			🟩	🟦	🟨	🟧		🟩		🟨	🟧						
S23							🟩	🟦	🟨			🟩		🟨	🟧		🟩		🟨	🟧						
S21		🟩	🟦				🟩	🟦				🟩	🟦	🟨	🟧		🟩		🟨	🟧						
S5		🟩				🟥		🟦		🟧		🟩	🟦	🟨			🟩	🟦	🟨			🟩				Procedural
S11			🟦		🟧		🟩	🟦		🟧		🟩			🟧		🟩		🟨	🟧				🟨	🟧	Reasoners
S17		🟩		🟨	🟧		🟩	🟦	🟨	🟧		🟩		🟨	🟧		🟩		🟨	🟧					🟧	
S18		🟩		🟨	🟧		🟩	🟦	🟨	🟧		🟩			🟧		🟩		🟨	🟧		🟩			🟧	
S19		🟩		🟨	🟧		🟩	🟦		🟧		🟩		🟨	🟧		🟩		🟨	🟧					🟧	
S24		🟩			🟧		🟩	🟦	🟨	🟧		🟩		🟨	🟧			🟦	🟨	🟧		🟩	🟦	🟨	🟧	

Limited Knowledge (LK)	🟥
Conceptual Knowledge (CK)	🟩
Procedural Knowledge (PK)	🟦
Schematic Knowledge (SK)	🟨
Strategic Knowledge (TK)	🟧

Figure 5.8. Students' use of the types of knowledge for each section of the code: Activity #2

Four groups of students were identified using this clustering approach: problem-oriented (S6, S7, S14, S20), schematic explainers (S1, S1, S2, S3, S4, S8, S9, S10, S12, S13, S15, S16, S21, S22, S23); procedural explainer (S5); and the reasoners (S11, S17, S18, S19, and S24). In order to characterize these groups, it was important to go into a more detailed level. Figure 5.9 differentiates these clusters using different colors for their explanations within each sections. The boxes in Figure 5.9 are highlighted if the student demonstrated one of these categories in their explanation for the given section. The different colors correspond to different clusters.

The problem-oriented explainers (brown) mostly focused on using declarative knowledge throughout the four sections, with connections to the problem in sections one and two (Figure 5.9(a) and 5.9(b)). They would explain what an instruction does and what a variable represent, but would not explain how, why, or when this instruction would work. For instance, student S6 talked about the coefficients for the quadratic formula as: "set the value of variable 'm','m' is the a variable in the quadratic formula"

The schematic explainers comprises a total of 15 students who used schematic and strategic knowledge to explain sections two, three, and four. This group would consistently explain how to compute the value of side a in section two, making connections to the problem ("by using the law of cos states for the top tr[i]angle, we could calculate the length of a, also since A1 is given in degree we need to convert it to rad"), and would use the background knowledge to connect the use of quadratic formula with the goal to find the solutions ("now we directly use the equation $x = (-n + / - sqrt(n^2 - 4 * m * p))/(2 * m)$ to solve the problem since it has two option +/2 it will also have to possible solution c2a and c2b").

Student S5 was the only student in the group of the procedural explainers. This student was not close to any other student in the group because his/her explanations included procedural knowledge in sections two, three, and four, and where explicit about what would happen in execution time. For most of the sections, this student would explain that because the instruction had a semicolon at the end, "This line of code will be hidden from the command window". These

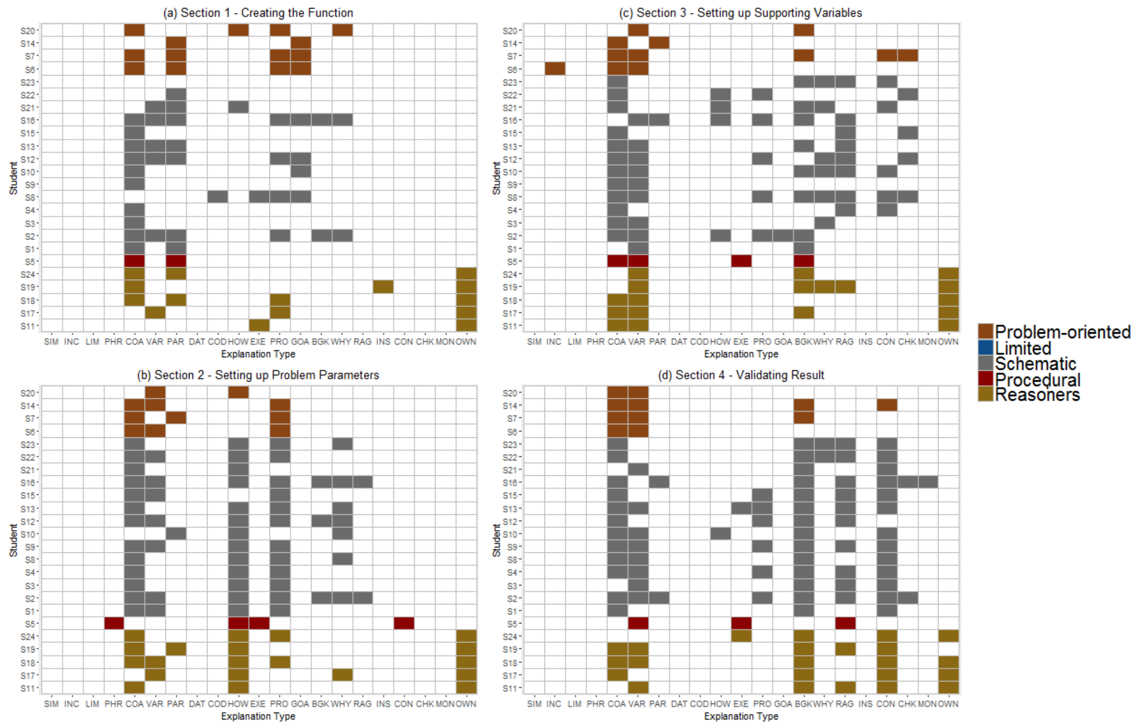


Figure 5.9. Patterns of students' explanations by section type: Activity #2

explainers that focused on what will happen on execution time, are denominated procedural explainers.

The fourth and final group can be described as the reasoners, which corresponds to the group of students who explained their own solution to the program instead of the provided one (OWN). This group also made rich conclusions in the last sections, making connections to their schematic and strategic knowledge (BGK, INS, and CON). This group of explainers had been previously identified by Renkl (1997), when students tried to come up with their own solution to the example while explaining it.

Figure 5.9 also describes the patterns of students' explanations per section. For instance, on the function definition, more than half of the students described that a function was being created (COA) and listed the parameters (PAR), while only a third of the students explicitly mentioned the goal of the function (GOA) and made a connection to the original problem (PRO). Five students commented their own solution instead of the worked example (OWN).

The second section, in which the Law of Cosines was applied to estimate the value of a corresponding to the diagonal, showed a larger number of connections to the problem (PRO), and students described how this parameter was being computed (HOW). In section three, students used their schematic knowledge to describe why the quadratic formula could be used to solve this problem, and five students identified the definition of coefficients as a chunk of code with certain purpose (CHK). The fourth section identified the two solutions of the quadratic formula, and so students needed to identify the two conditions for the two possible solutions. Hence, 19 students identified these conditions (CON), 13 students made connections between the action and the goal (RAG), and 17 of the students used their background knowledge to explain this section (BGK). In general, connections between the background knowledge and the goal of the function were present more often in sections that involved the use of the quadratic formula to find the solution (sections three and four), while the connections to the problem took place when specific problem parameters were being computed. These patterns were further compared to similar sections in the other activities.

Activity #5

The explanations for activity #5 were submitted by 26 students (see Figure 5.10). Similar to what had been identified in the previous activity, the most used type of knowledge was the declarative knowledge. However, this activity showed an increment on the use of procedural knowledge in sections two, three and four, and schematic knowledge in section two, where several students explained why biggest and smallest needed to be the same at the beginning. More instances of strategic knowledge for section three (i.e., the loop section) and section four means that students clearly highlighted the conditions of the loop (CON), and related these conditions to the user interaction during execution time (EXE). The additional complexity of this activity that involved the use of loops did not represent more instances of limited knowledge (LK), of which only two were identified (COD).

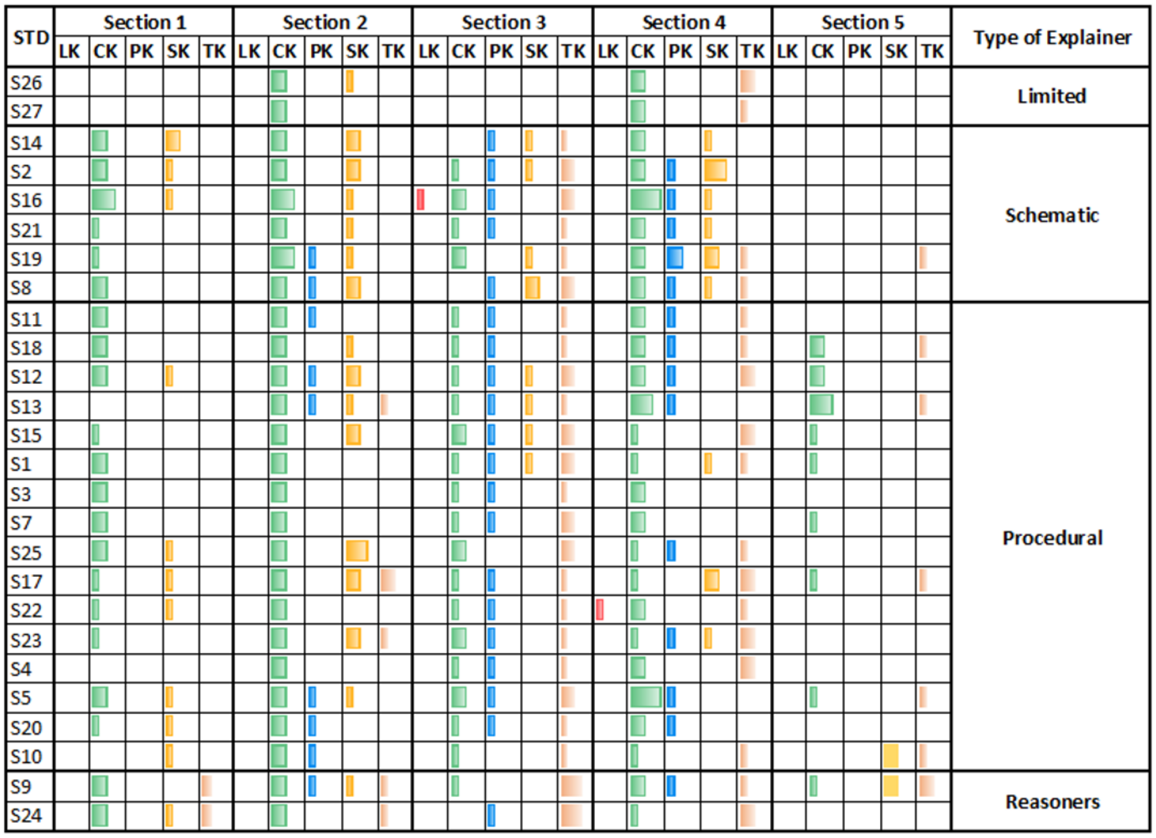


Figure 5.10. Students' use of the types of knowledge for each section of the code: Activity #5

The hierarchical cluster analysis suggested four different groups of students for this activity: limited (S26, S27), schematic (S14, S2, S16, S21, S19, S8), procedural (S11, S18, S12, S13, S15, S1, S3, S7, S25, S17, S22, S23, S4, S5, S10, S20), and reasoners (S9, S24). The detailed categories these groups used are presented in Figure 5.11. The limited explainers only explained sections two and four, using mostly declarative knowledge with simple explanations like: "the smallest number is determined" and "this is the initial input, which will be both the smallest and largest number starting out". However, these two students did not described the goal of the function or explained how this program was solving the problem.

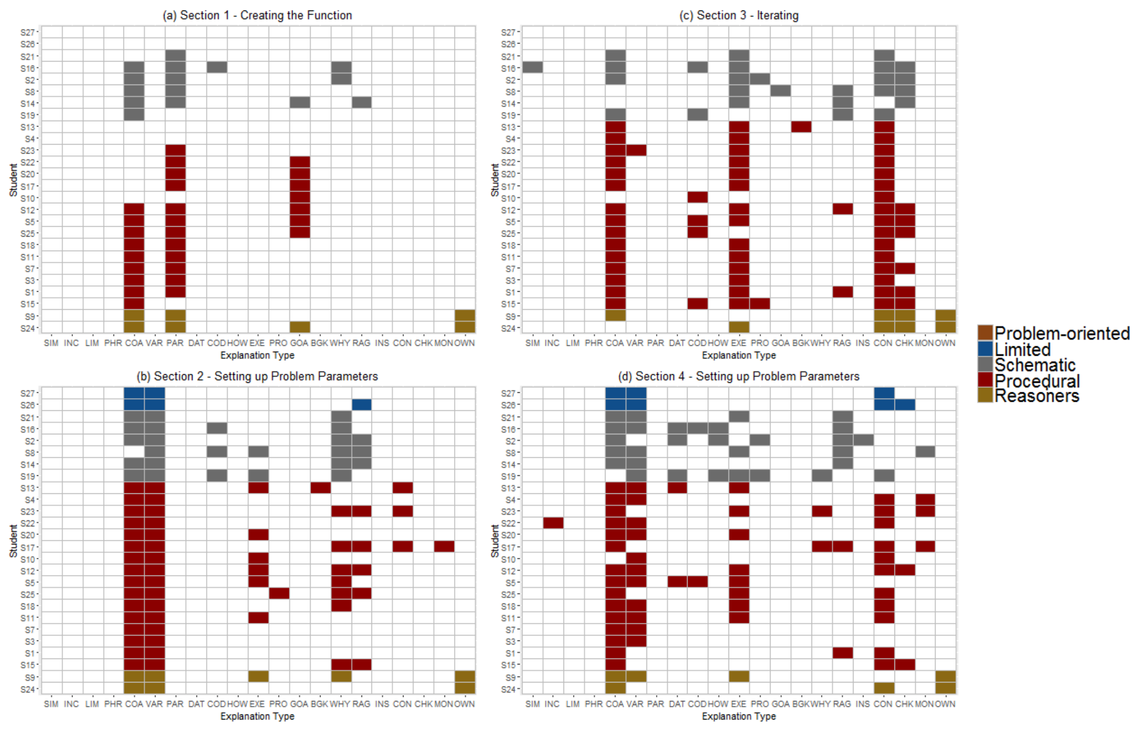


Figure 5.11. Patterns of students' explanations by section type: Activity #5

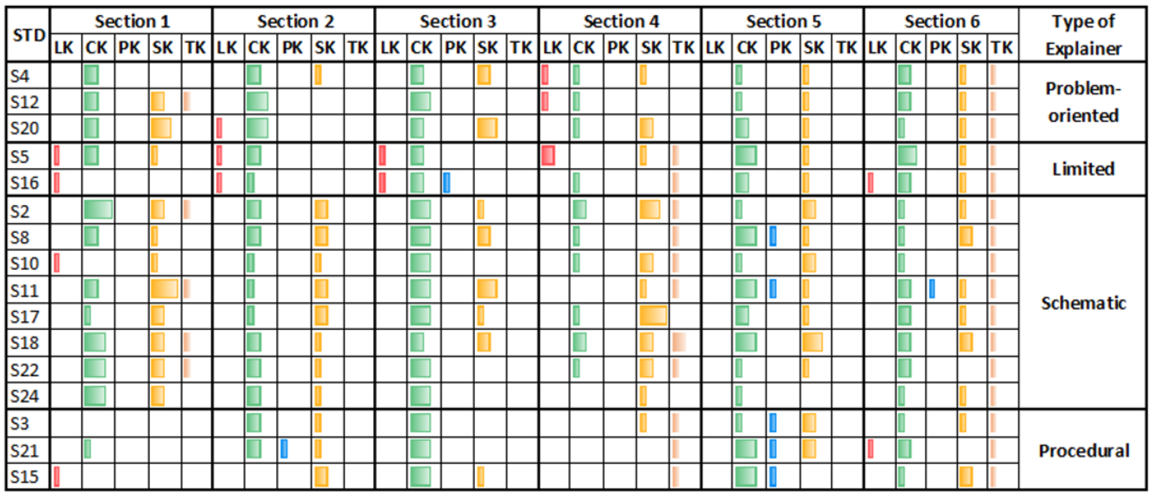
The schematic explainers described rationales for the instructions in section two ("Here we are defining biggest as smallest because as it stands, the first entry is the biggest number."), and the relationship of actions with goals in section four ("assigns biggest number as the largest number input so far, so if the new input was larger than any other input so far, it would be stored as the new largest"). This group is slightly different to the schematic explainers from the activity #2, because they did not use background knowledge in their explanations. However, none of the students did. A possible reason for this phenomenon is that the problem itself did not involve any disciplinary knowledge that could be applied here during the explanation process.

As in the previous activity, the procedural explainers focused mostly in what would happen in execution time. In this case, students described the user interacted with the program: “variable smallest is inputted by the user after the prompt ’Give me a number:”’. Finally, the reasoners commented all four main sections, but describing their own solution to the problem. These students may have had the background knowledge to be able to solve the problem on their own as opposed to being interested on learning from examples.

In general, Figure 5.11 shows that a larger number of students mentioned the goal of the function (GOA nine students) as compared to the previous activity (seven students). There were almost no connection to the problem (PRO) or to the background knowledge (BGK) in students’ explanations. When setting the initial conditions, all students used the declarative knowledge to describe what the variables represented (VAR), and more than half of the group were able to explain why the code was setting the same value for the variables smallest and biggest (WHY). In the loop section (third section), the conditions of application of the loop were highlighted by 23 out of 26 students(CON), and 20 students explained that these conditions depended on the user input, which would occur in execution time (EXE). The fourth section (i.e., setting up problem parameters) was a mix between the sections two and three, with instances of procedural (EXE 12 students), schematic (RAG seven students), and strategic knowledge (CON 15 students). The parameters that were being estimated would become the conditions for the loop, and one of these values would come from the user in execution time. Another interesting thing about the fourth section is that four students included monitoring statements. The first line of code among the three that comprised this section was unnecessary because if the variable newnum was smaller than the variable smallest, the loop condition would not have allowed its execution on the first place. These four students included comments like: “I think we don’t need this command, because if there is a number smaller than smallest the program stop”.

Activity #11

Sixteen students submitted their explanation for this activity that was not graded, but provided extra-credit for those who decided to write comments within the code. The most common type of knowledge continued to be the declarative knowledge in all sections. However, since this was a more complex and disciplinary-related example, the schematic and strategic knowledge were also used in all sections. The use of procedural knowledge was limited to a handful of students, especially in section five, and only one of the students wrote an explanation involving the four types of knowledge within the same section. Figure **??** presents the types of knowledge students used within each section.



STD	Section 1					Section 2					Section 3					Section 4					Section 5					Section 6					Type of Explainer
	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	LK	CK	PK	SK	TK	
S4		🟩					🟩		🟨			🟩		🟨		🟥	🟩		🟨			🟩		🟨			🟩		🟨	🟧	Problem-oriented
S12		🟩		🟨	🟧		🟩					🟩				🟥	🟩					🟩		🟨			🟩		🟨	🟧	
S20		🟩		🟨		🟥	🟩					🟩		🟨			🟩		🟨			🟩		🟨			🟩		🟨	🟧	
S5	🟥	🟩		🟨		🟥	🟩				🟥	🟩				🟥			🟨	🟧		🟩		🟨			🟩		🟨	🟧	Limited
S16	🟥					🟥	🟩				🟥	🟩	🟦				🟩			🟧		🟩		🟨		🟥	🟩		🟨	🟧	
S2		🟩		🟨	🟧		🟩		🟨			🟩		🟨			🟩		🟨	🟧		🟩		🟨			🟩		🟨	🟧	Schematic
S8		🟩		🟨			🟩		🟨			🟩		🟨			🟩			🟧		🟩	🟦	🟨			🟩		🟨	🟧	
S10	🟥			🟨			🟩		🟨			🟩					🟩		🟨	🟧		🟩		🟨			🟩			🟧	
S11		🟩		🟨	🟧		🟩		🟨			🟩		🟨					🟨	🟧		🟩	🟦	🟨			🟩	🟦	🟨	🟧	
S17		🟩		🟨			🟩		🟨			🟩		🟨			🟩		🟨			🟩		🟨			🟩		🟨	🟧	
S18		🟩		🟨	🟧		🟩		🟨			🟩		🟨			🟩		🟨	🟧		🟩		🟨			🟩		🟨	🟧	
S22		🟩		🟨	🟧		🟩		🟨			🟩					🟩		🟨	🟧		🟩		🟨			🟩			🟧	
S24		🟩		🟨			🟩		🟨			🟩							🟨			🟩					🟩		🟨	🟧	
S3							🟩		🟨			🟩							🟨	🟧		🟩	🟦	🟨			🟩		🟨	🟧	Procedural
S21		🟩					🟩	🟦	🟨			🟩								🟧		🟩	🟦	🟨		🟥	🟩			🟧	
S15	🟥								🟨			🟩		🟨						🟧		🟩	🟦				🟩		🟨	🟧	

Figure 5.12. Students' use of the types of knowledge for each section of the code: Activity #11

Four groups of explainers were identified among the 16 students (see Figure 5.13): problem-oriented (S4, S12, S20), limited (S5, S16), schematic (S2, S8, S10, S11, S17, S18, S22, S24), and procedural (S3, S21, S15). The problem-oriented explainers mostly used declarative knowledge but made multiple connections to the problem. The limited explainers mostly used declarative knowledge, and showed limited knowledge of the problem and the program by paraphrasing (PHR) and making incorrect statements (INC) in at least four of the sections. The schematic explainers made connections to the problem and explained the rationale behind the use of a sparse matrix in section two (RAG and WHY). Finally, the procedural explainers identified the parameters in section two, and explained how the distance between the atoms was being computed in section five (HOW).

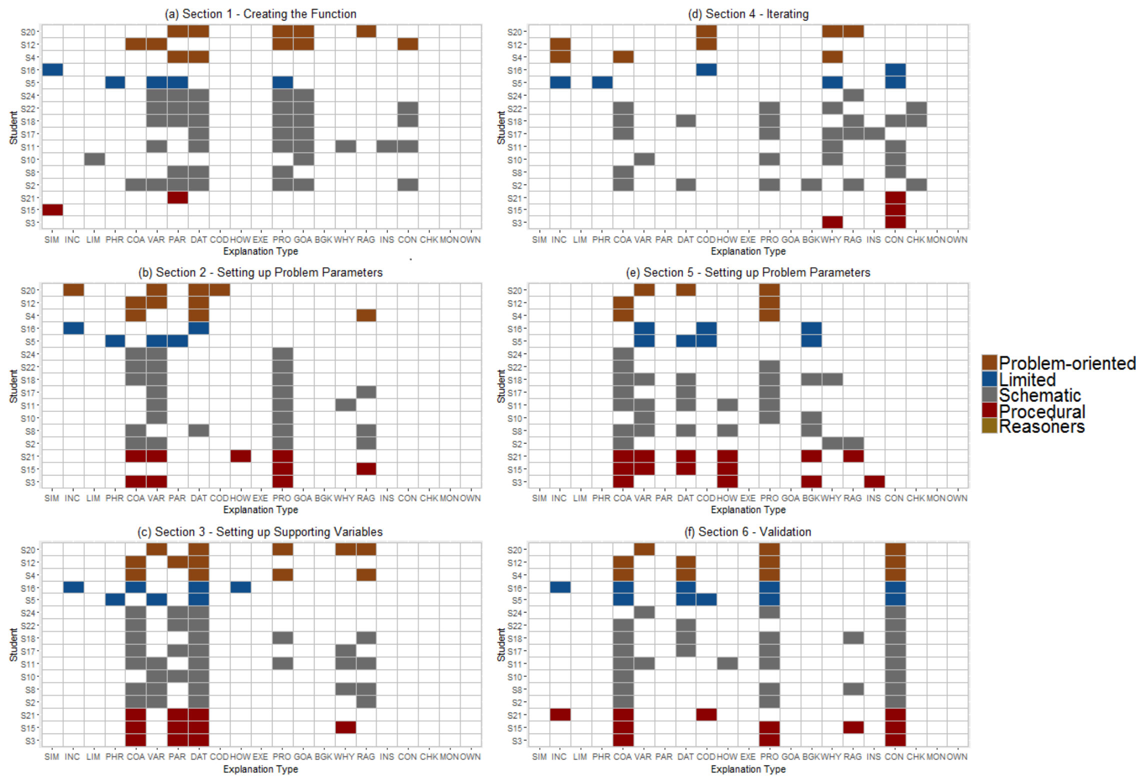


Figure 5.13. Patterns of students' explanations by section type: Activity #11

There are coincidences and differences in students' explanations of these sections as compared to the two previous activities. For instance, the sections two and five (setting up problem parameters) shown connections to the problem, and section three (setting up supporting variables) depicted rationales from schematic knowledge, a pattern that had been identified in activity #2. On the other hand, the portion of students describing the goal of the function (GOA) and the data type of the parameters (DAT) increased significantly. This could be the case because of the nature of the problem, which was more applied to students' disciplinary knowledge, and requires to deal with complex data structures. Also, the conditions in the loop section were not highlighted as often as in Activity #5. More than 65% of the students made connection to the problem in section two, when the number of atoms was being calculated, and more than half of the students (68.75%) used the schematic knowledge in section three to describe why a sparse matrix was important for the goal of the function (WHY, RAG). Nevertheless, these differences in portion of students should be considered with caution because the number of students that submitted these explanations decreased due to the extra-credit nature of the assignment.

5.2.3 Results

What are the characteristics of students' explanations in a glass box approach to CSE education?

Student use of different types of knowledge slightly varied among the three activities, although a consistent distribution can be identified. Figure 5.15 represents the percentage of occurrences for each type of knowledge within each activity. The most used type of knowledge was declarative knowledge throughout the three activities, which was employed to explain what an instruction was doing, often limiting students' explanations. The procedural knowledge was rarely used, so students did not often explain how each section was actually completing the action they described. The schematic knowledge was also commonly used, especially for activities #2 and #11, which involved disciplinary problems that required some kind background knowledge (e.g. law of cosines and distance function). The strategic knowledge was the third most used one among the students, especially for activity #5, where students highlighted the conditions for the loop to iterate through. Finally, the percentage of limited knowledge increased as the complexity of the activity increased, particularly with activity #11 that involved nested loops and if-clauses, as well as the use of a complex structure as the sparse matrix.

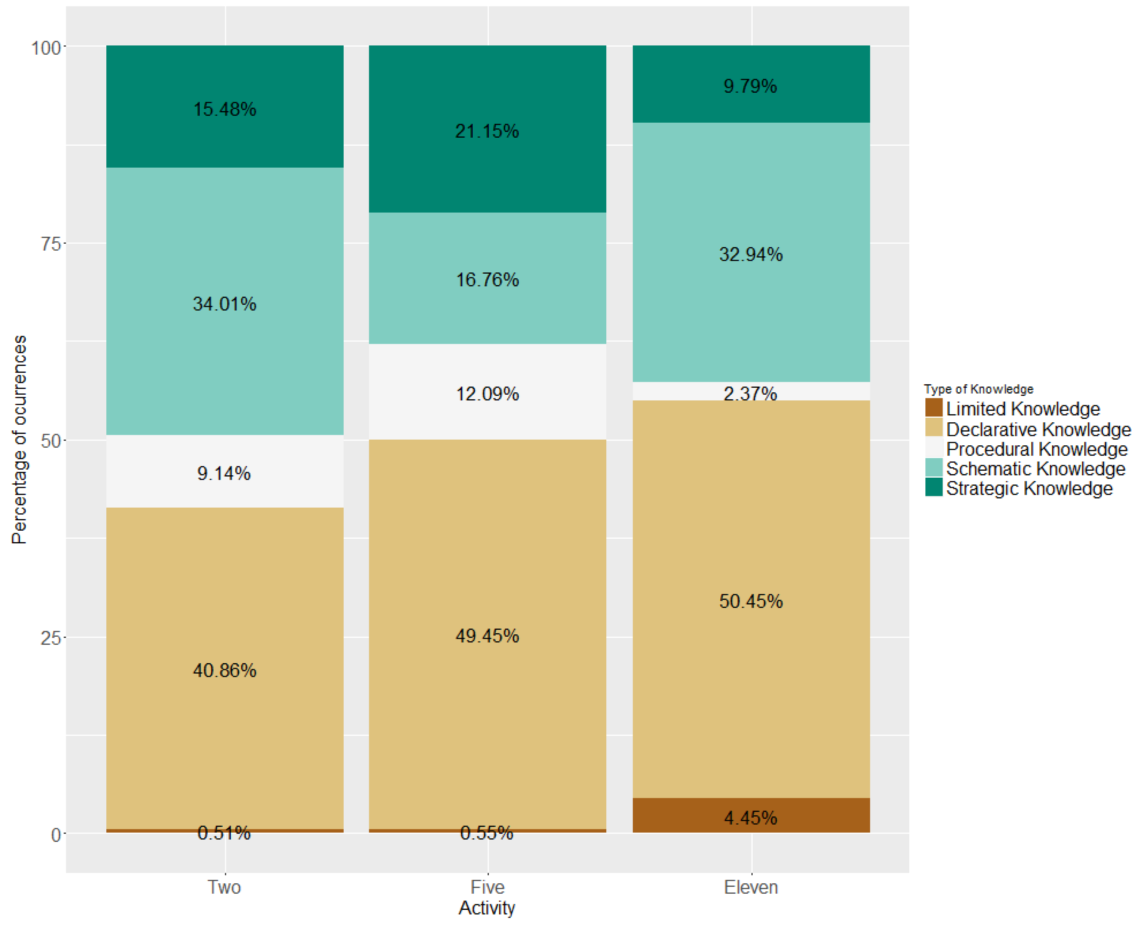


Figure 5.14. Percentage of occurrences of the types of knowledge within each activity

In order to identify what type of categories were the most used within each type of knowledge, Figure 5.15 depicts the number of instances for all the categories grouped by type of knowledge. The most common limited knowledge category among the three activities was the use if incorrect explanations (INC 10 instances), followed by paraphrasing (PHR 5 instances). Students' use of declarative knowledge often focused on the consequences of actions ("what an instruction does"), but a third part of these instances were also dedicated to what the variables represented (VAR). The lower number of parameter descriptions (PAR) is understandable given that not all sections involved parameters.

Student who used procedural knowledge focused mostly on how it would happen in execution time (EXE), rather than how the instructions where doing what they did. Likewise, 97 of the instances in schematic knowledge involved connections to the problem (PRO), almost twice the connections to background knowledge (BGK), but a significant number of rationale explanations (WHY 56) and relationship action-goals (RAG 68) were also present. Finally, the most used strategic knowledge involved the identification of conditions of application of actions (CON), especially loops and if-clauses.

These values should be revised in relation to the possible number of instances that we could have found among the three activities. One or more categories could be assigned to each commented section of the code, and we had 24, 26, and 16 students' explanations for activities #2, #5, and #11 correspondingly. Since we had five sections in activity #2, five in activity #5, and seven sections in activity #11, assuming that students did not comment the last section (i.e. end of the function), we have a total of (24*4)+(26*4)+(16*6) = 296 "commentable" sections. Hence, even though we see important categories for all the types of knowledge in terms of the number of instances, these might not correspond to the most cognitively engaging. For instance, the categories who appeared at least 30% of the times were COA, VAR, PRO, and CON. These categories comprised either simple descriptions or the use of information at hand (e.g., the problem statement and the conditions of a loop). On the other hand, the number of monitoring statements in the three activities was only seven, and the number of identified chunks was only 25. Finally, in the middle of the spectrum we have the connections to background knowledge (BGK) and rationale for decisions within the code (WHY and RAG), which were all around 20% of the possible instances.

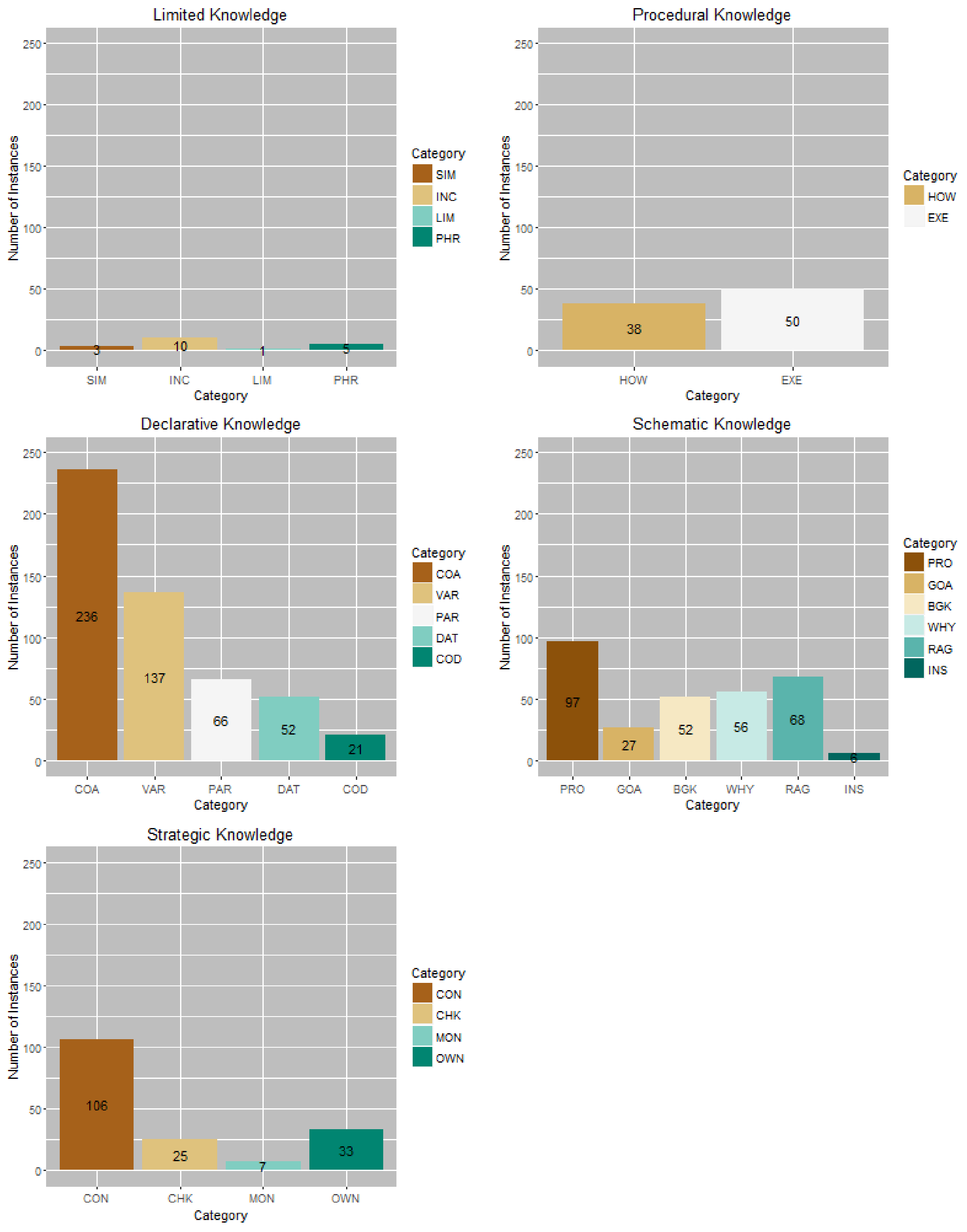


Figure 5.15. Number of instances of each category within the types of knowledge

How does the characteristics of students' explanations in a glass box approach to CSE education change over time?

Six types of explainers where identified among these three activities: problem-oriented, limited, declarative, procedural, schematic, and reasoners. Students' distribution among these types of explainers changed from one activity to another, and the time of the semester for the activity as well as the type of example and the characteristics of the programming code seem to have influenced in these shifting. For instance, only activities #2 and #5 involved reasoners, students who explained their own solutions instead of the provided one. It is possible that these students were able to come up with their own solution given the low complexity of these explaining activities, but needed to use the worked-out solution for more complex exercises.

Another example is that when the worked-example involved disciplinary knowledge such as activities #2 and #11, we were able to identify a large percentage of schematic explainers. These explainers used their background knowledge to describe the rationale for the instructions. However, in activity #5 that was basically a programming problem with user interactions, students did not have to use background knowledge to explain it, and the size for this group of explainers decreased. Instead, the percentage of students in the procedural group increased for activity #5 to describe how the interaction with the user in execution time would affect the program. Finally, the percentage of limited explainers increased with the complexity of the activities. For instance, activity #11, the most complex activity, which solved a disciplinary problem involving atoms and distance functions, showed the largest percentage of schematic explainers, and the largest percentage of limited explainers. Figure 5.16 presents the distribution of explainers within each activity.

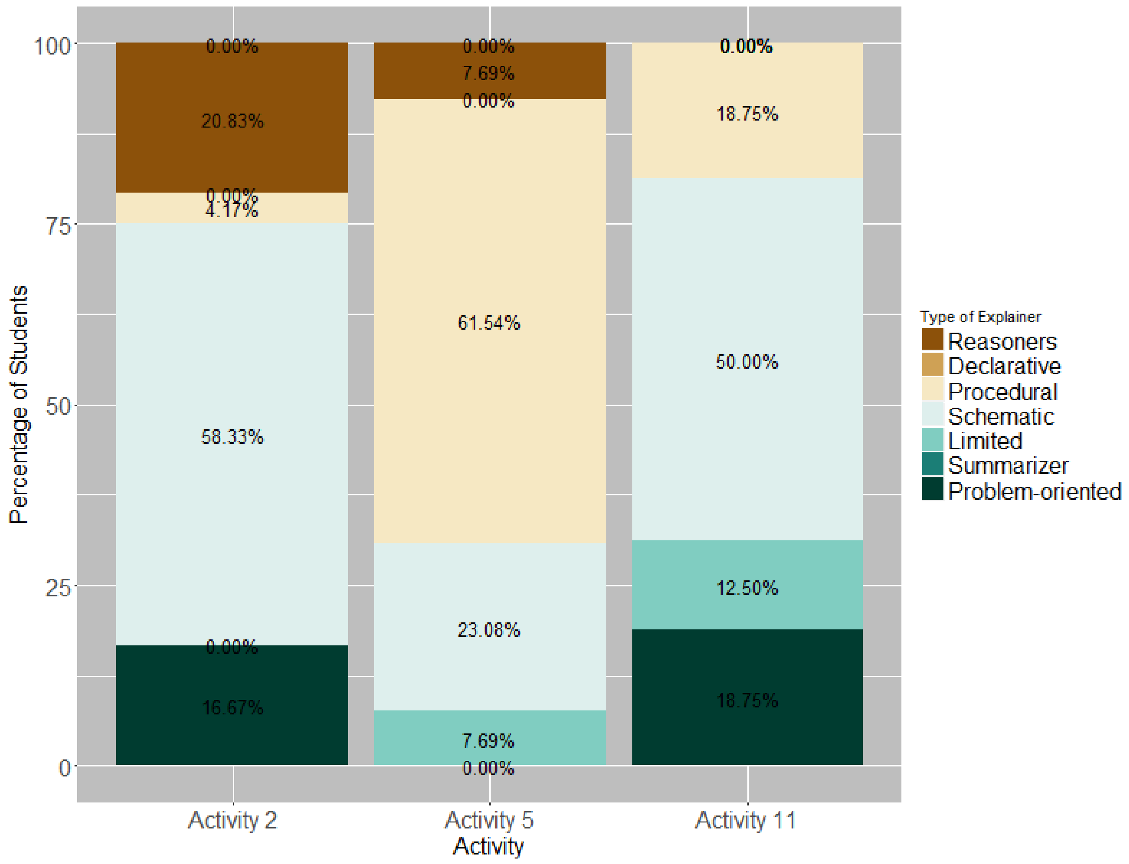


Figure 5.16. Distribution of types of explainers between activities: CPMSE

What common misunderstandings in programming can be identified from students' explanations in a glass box approach to CSE education?

The percentage of explanations involving limited knowledge for these three activities was low compared to the other types of knowledge. Only 10 instances of incorrect explanations and one instance of limited explanation were identified among students' in-code comments for these three activities. One of the misunderstandings that students showed was related to the sequence of execution of multiple lines of code. For example, in a while loop that had three instructions inside, one student said that the third line would not be executed if the condition of the loop was later satisfied: "this part of loop will not run if newnum ¡ smallest from the first line of the body of the loop". This is assuming that the loop condition is constantly checked, as opposed to at the end of every iteration.

The loops also showed students' misunderstanding of their use during activity #11. While one student considered that a loop and an if-else clause were the same thing: "We can use an if loop to correct for the cutoff minimum distance."; three students assumed that the conditions for the loop was to iterate through rows and columns: "Create two for loops' that will first loop through different columns in a row and then rows". This was not the case because both indexes were being used to iterate through rows, since the columns of the matrix represented the coordinates X, Y, Z. It is possible that students were referring to the sparse matrix, where the data is saved by row-column pairs, but the iteration process was definitely not through that matrix. There was also a misunderstanding related to the complex data structures such as matrices, and sparse matrices. For instance, one of the students assumed that if the if-clause condition was not satisfied then the bondmat would keep the original assigned value: "the output 'bondmat(n,m)' is equal to 'len' or the vector length of 'dist'. If the conditon len¡cutoff is not true, then 'bondmat=sparse(N,N)'." This is incorrect because the original instruction simply initializes the structure, but the initial values correspond to zero. In the same instruction, other student assumed that this was storing n and m instead of the length between atoms n and m: "stores n and m as the "bondmat" that is output".

Another element to highlight for this group was that four students demonstrated monitoring statements within their explanations for activity #5. These students highlighted that one of the lines was unnecessary because once the condition of the loop was met, this comparison inside the loop was simply replicating it. These four students included comments like "I think we don't need this command, because if there is a number smaller than smallest the program stop".

5.3 Relationship between Students' Explanations and Student Ability to Program

The previous section showed how students wrote different explanations for all the activities, and how we could group them based on the types of explanations they used. This chapter focuses on identifying the relationship between the characteristics of students' explanations and their ability to do computer programming.

Twenty-six freshmen engineering students enrolled in the CPMSE course participated in this part of the study. Eighteen students reported not having any prior programming experiences, seven students had taken one programming course, and two students had taken two previous courses, and two students had been exposed to more than two courses. The guiding research questions are:

- How do the characteristics of students' self-explanations in a black box context relate to relate to their perceived ability to program?
- How do the characteristics of students' self-explanations in a black box context relate to their performance in the course?

5.3.1 Data Collection and Data Analysis

Two of the self-explanation activities were explored in this part study: exercise #2 and exercise #5. The activity #2 corresponded to the first exercise in which students would be required to submit their written explanations, while the in-class assignment #5 was the last one that was graded. These two activities allowed us to have the entire classroom as the sample, and not only to have those students who decided to do the extra-credit assignment. Moreover, activity #5 was the first worked-example that included a while-loop structure, a complex concept in computer programming.

The qualitative analysis of exercises #2 and #5 resulted in four clusters for each activity. Twenty four students submitted their explanations for in-class activity #2, while 26 students completed the assignment for activity #5. The four groups of explainers for activity #2 are: problem-oriented (S6, S7, S14, S20), schematic explainers (S1, S1, S2, S3, S4, S8, S9, S10, S12, S13, S15, S16, S21, S22, S23); procedural explainer (S5); and the reasoners (S11, S17, S18, S19, and S24). While the explainers in activity #5 were distributed as: limited (S26, S27), schematic (S14, S2, S16, S21, S19, S8), procedural (S11, S18, S12, S13, S15, S1, S3, S7, S25, S17, S22, S23, S4, S5, S10, S20), and reasoners (S9, S24). These groups of explainers in both activities were used to identify the relationship among perceived ability, explanation style, and performance.

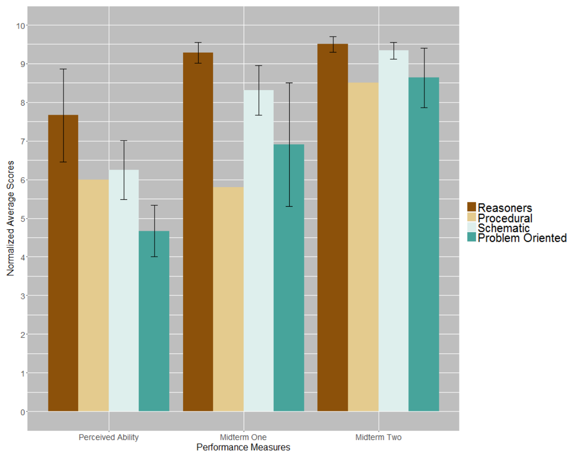
Previous studies had identified that good explainers produce more explanations (e.g. (M. Chi et al., 1989)). Hence, it is important that we identify this quantity in our context. In order to account for the number of explanations, three different measures were considered. The most obvious one corresponded to the number of words students employed within their explanations. However, a lot of words does not necessarily mean multiple explanations. Therefore, the second measure corresponded to the number of categories from the coding scheme found within students' explanations. Finally, the third measure corresponded to the number of categories without considering those showed as limited knowledge. Besides the characteristics of students' explanations, the first data source we used in this section corresponded to students' perceived ability to do computer programming. The five-level Likert-scale questions from the beginning of the semester survey were averaged out to compute a composite score that could assess the student perceived ability to do computer programming. Then students were grouped as low perceived ability, mid perceived ability, and high perceived ability. Values between 1 and 2.5 were considered low, values between 2.51 and 3.99 were considered mid ability, and values equal or above 4 were considered high perceived ability. These groups were used to identify whether student perceived ability had an effect in the way they generated explanations in this context.

Finally, students' performance on the midterm exams were considered as the student course performance. Both scores were normalized (0-10), and students were grouped as low-mid-high performers. The groups for these scores were created as: low below six; mid between six and 8.5; and high above 8.5. The distribution for midterm one was as: six students were low performers, seven students were mid performers, and 11 students were high performers. Meanwhile, six students were mid performers and 18 students were high performers for midterm two. There were no low performers in midterm two. The rationale for using these scores instead of the overall course grade is because they were individual scores that were not affected by other components of the course. For instance, the overall course grade could be higher for those students who decided to submit all the extra-credit assignment as compared as those who did not. Likewise, the collaborative nature of the projects, where students discussed ideas and troubleshoot together, may have an effect that does not relate to their own ability to do computer programs.

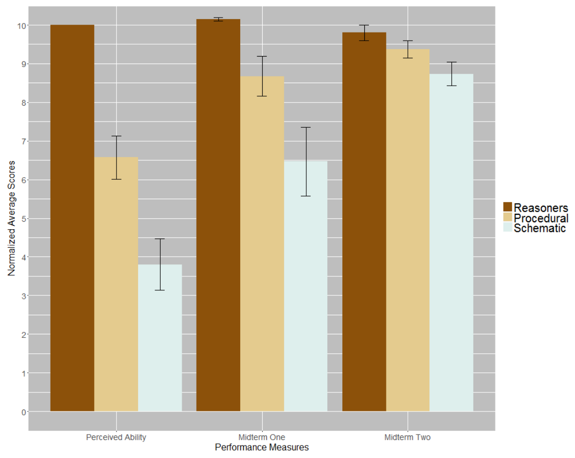
5.3.2 Results

The first analysis we conducted aimed at identifying whether there were differences in students' perceived ability and performance measures among the groups of explainers that had previously been identified. Figure 5.17 presents a comparative bar plot of these performance measures for the groups of explainers within activity #2 (a) and activity #5 (b).

Schematic explainers showed a lower perceived ability compared to reasoners explainers in both activities, and to procedural explainers in activity #5. Although the trends were similar in both activities, the results were only significant for activity #5 (F(2,17)=5.74, p-value=0.0125). These schematic explainers from activity #5 also showed a lower performance in midterm one as compared to reasoners and procedural explainers. This result suggests that the less experienced students created explanations that involved more schematic knowledge than students with more experience. The two students that were part of the limited explainers did not complete this survey. Thus, the group of limited explainers are inconclusive, because it only involved two students, one of which dropped the course and did not complete the performance measures.



(a) Activity #2



(b) Activity #5

Figure 5.17. Average differences in student performance by explainer type

The second step was to identify whether the initial perceived ability also had an effect on the number of explanations the students created. Figure 5.18 presents the comparison among the number of identified categories, explanations, and words, from students' in-code comments in activity #2 (a,b, and c) and #5 (d, e, f). The number of categories correspond to the instances of all the codes from the coding scheme (see Section 4-2) found in students' in-code comments, while the number of explanations did not consider the categories related to limited knowledge. Figure 5.18 (a and d) confirms that students with a higher perceived ability wrote shorter explanations, as compared to mid and low performers. This results were only statistically significant for activity #2 (F(2,16)=3.796, p-value=0.048). Likewise, the number of words for low performers in midterm one was significantly higher compared to high performers both in activity #2 (F(2,20)=6.543, p-value=0.0065) and #5 (F(2,21)=4.519, p-value=0.0233), trend that was confirmed in midterm two.

These results suggest that students with higher ability to program write shorter explanations. Although these explanations showed non-significant differences in the number of categories (see Figure 5.18), the types of knowledge students used were indeed different. While, high performers wrote simple declarative and procedural explanations, the low performers used schematic knowledge. It is possible that while high performers were simply describing what they already understand from the code, the low performers were actually engaging in a thorough self-explanation process to make sense out of the example.

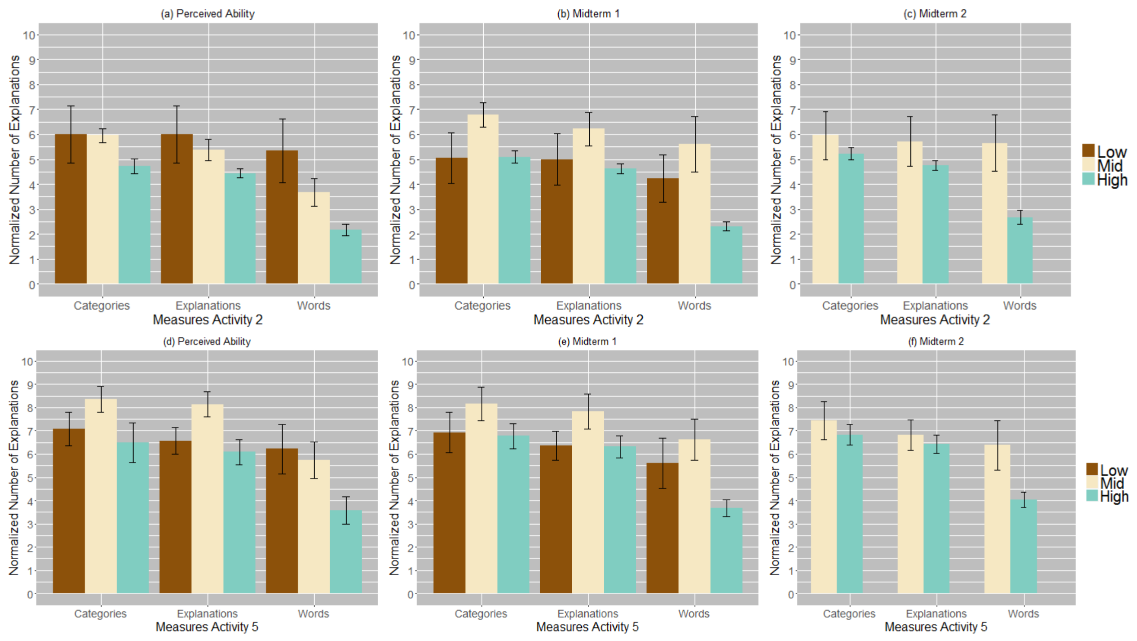


Figure 5.18. Number of categories, explanations, and words in students' explanations for activity #2: (a) perceived ability; (b) performance in midterm one; (c) performance in midterm two; and activity #5: (d) perceived ability; (e) performance in midterm one; (f) performance in midterm two

5.4 Summary of the findings

This case study explored the characteristics students' explanations as in-code commenting activities in a glass box approach to CSE education. Students in a programming course submitted their written explanations to programming worked-examples as part of the weekly in-class activities. Specifically, we explored the affordances of these activities for different students, the characteristics of students' explanations, and the relationship between students' explanations and students' ability to do computer programming.

Four types of explainers were identified within each of the activities that were analyzed. The reasoners corresponded to students who had a strong background knowledge that enabled them to solve the problem on their own instead of self-explaining the provided solution. The limited explainers corresponded to a small group of students who depicted some common misunderstandings in programming, while the procedural explainers focused on the execution of the program, and the interactions with the user. The schematic explainers were students who described the rationale for several sections of the code and made connections to background knowledge. These students actually started with a low perceived ability to do computer programming, but performed on average, as good as the rest of the students. We hypothesize that these students were actually engaged in a reflective process, while the high-ability students were simply describing what they already knew.

The findings from this study also suggest several affordances of in-code commenting activities for different students. For instance, out of twenty-one students who answered the final survey, three students said they did not need them, while three more gave no reason for their use. Four students said they only did them for extra-credit, while 11 students said these activities actually helped them to better understand concepts of the class. From students' explanations in activity #2, four out of these 11 students were described as reasoners, while six of them were described as schematic explainers, and one student was described as problem-oriented. Similar patterns were identified in activity #5 (two reasoners, three schematic, and six procedural), and in activity #11 (six schematic, on procedural, one limited, and one problem-oriented). This connection between the affordances and the characteristics of students' explanations reinforces our hypothesis: most of the students doing comprehensive schematic explanations were actually reflecting on their own learning.

CHAPTER 6. GLASS BOX APPROACH

The second case corresponds to a sophomore level course in materials engineering called thermodynamics of materials. As opposed to the first case, this was not a programming course but a core course in materials engineering that involved the use of computational tools to represent disciplinary phenomena. Students had access to Python code and could use it for their projects, but most of the functions were encapsulated into libraries. Students did not have access to the underlying mechanisms of the simulations, and therefore, they were considered a black box approach to CSE education.

This chapter is divided into four sections aimed at responding the three guiding research questions. Section 6.1 explores the affordances of in-code commenting activities students in the black box context: RQ1. Section 6.2 describes the characteristics of students' explanations for three computational modules in this course: RQ2. Section 6.3 compares the types of explainers and the characteristics of students' explanations to students' ability to do computer programming: RQ3. Each section defines its own questions that contribute to answer the overall research questions of the study. These sections also describe in more detail than Chapter 4, the specific procedures for each research question.

6.1 Affordances of in-code commenting activities for students

Students participating in this part of the study were enrolled in a sophomore-level course called thermodynamics of materials (THERMO) at Purdue University, either during spring 2015 or spring 2016. The THERMO course included three computational modules in which students used a simulation tool to represent disciplinary phenomena. The simulation tool is a set of GIBBS Python libraries as implemented in the Virtual Kinetics of Materials Laboratory (VKML) hosted a nanohub.org (Alabi et al., 2015; Cool et al., 2010). In each module, the course instructor walked students through a worked-example, interpreting the program output, and briefly describing the Python code. The students had attended two lecture sessions the same week before the module, where they were explained the THERMO concepts. Students were assigned two questions related to the module within the weekly homework assignment. One of the questions asked students to write in-code comments as part of the worked-example presented during the module. The second question asked students to modify part of the code to make it work for some other situation (e.g. changing the equations). These questions were only extra-credit and students could submit the homework working in groups in 2015. The questions and the homework assignments became individual and graded for the spring semester, 2016. This section explores the effect of this activities on students' engagement with the worked-examples, and the different ways in which students afforded them. The guiding research questions are:

- What is the effect of using in-code commenting activities on students' engagement with the worked-examples in the context of a black box approach to computational science and engineering?
- What are affordances of in-code commenting self-explanation activities in the context of black box approach to computational science and engineering?

6.1.1 Data Collection and Data Analysis

The data collected for this part of the study comprised three elements: (1) the number of submitted self-explanations in 2015 and 2016; (2) students' responses to interview open-ended questions related to their use of the worked-examples and the self-explanation activities; (3) students' responses to Likert scale and open-ended questions in an end-of-the-semester survey. For the final project of the course, students were invited to describe in an interview how they approached their solution, what was the role of computation in their learning process, and their perceptions about the computational modules and the self-explanation activities.

At the end of the semester, students were asked three five-level Likert scale questions related to the in-code comments activities in the form : "I feel writing comments within the sample code helped me to": (1) understand the Python examples; (2) solve the homework exercises; (3) understand better thermodynamics concepts. In 2016, two open-ended questions were added to this final survey asking: (a) Please include any additional ideas in which you think that commenting the examples or the lab sessions supported your learning process in this course (b) What would you improve about the Python lab sessions?

Student responses to the open-ended, as well as the interview comments were analyzed using categorical analysis. The procedures of analysis for reliability were the same as the ones employed in the glass box context, having two researchers looking at fifty percent of the data and negotiating until agreement was reached. The Likert-scale questions were plotted as a histogram to identify trends in their distributions, and the non-parametric Wilcoxon rank test was employed to compare distribution of student responses.

6.1.2 Results

The number of students who commented the worked-examples code for extra-credit in 2015 decreased over time. The worked-example for module 1 was commented by 14 groups of students corresponding to 45 students. Three groups of students wrote in-code comments for the worked-example in the module 2, and none of the groups wrote in-code comments for the module 3. When asked about this phenomenon during the interview, the three students who agreed to participate explained their own reasons, and what they thought could be the rationale from other groups:

Student One: “I never did that [the extra credit]- I decided I wasn’t very worried about the code. I was like, ”Okay. Do I understand what’s happening, kind of? Yes. All right. Then I don’t need to do the code.”

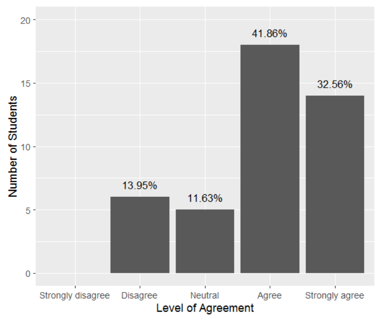
Student Two: “I think I did for the first one [commenting the code] was sort of just to be able to bridge the gap between, you know, what something is and how it looks when you’re trying to do it in a computer.”

Student Three: “I explained it [the worked examples] for other people, so that they could comment it. So I understand it a little bit. .. the way [the course instructor] does extra credit, is not really extra credit, because he teaches a curved class. I think I understand pretty well. Uh.. but other groups had a lot more trouble on the homework, so by the time they’d get to the extra credit, they’re like ”I’m just ready to be done with the assignment.”

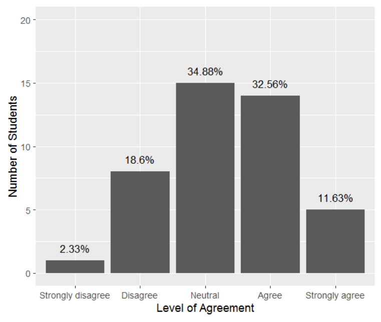
The distributions of student responses to the Likert scale questions are presented in Figure 6.1. Similar patterns were found for both semesters regarding the comparison of distributions to student responses: a large percentage of students (70%) agrees or strongly agrees that the self-explanation activities help them to understand the examples.

The Wilcoxon ranked test showed that writing in-code comments helped students to understand the Python examples more than to: (1) solve the homework exercises ([2015] Z-value=3.99, p-value <0.01, effect size r=0.75; [2016] Z-value=3.17, p-value <0.01, effect size r=0.89); and (2) better understand THERMO concepts ([2015] Z-value=3.86, p-value <0.01, effect size r=0.75; [2016] Z-value=3.96, p-value <0.01, , effect size r=0.86). The student responses regarding the usefulness of writing comments to solve homework exercises or to better understand THERMO concepts were neutral and non-significantly different from each other ([2015] Z-value=-0.14, p-value=0.9; [2016] Z-value=1.71, p-value=0.09).

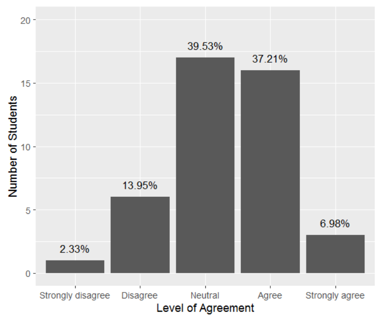
This result suggests that writing in-code comments within the Python code helped students to better understand the Python code within the worked-example even in a black box approach.



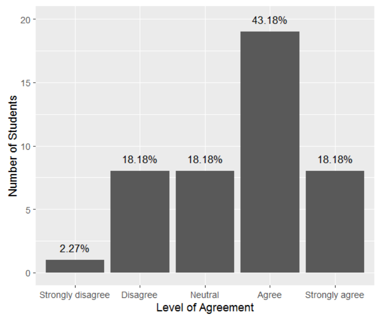
(a) Understand the examples



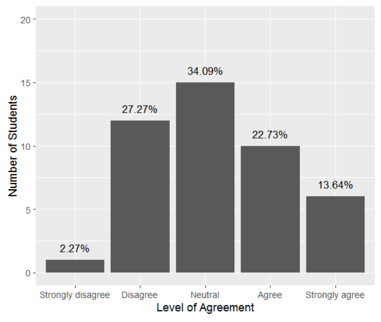
(b) Solve the homework



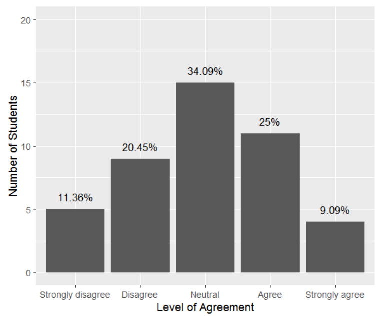
(c) Understand Thermo concepts



(d) Understand the examples



(e) Solve the projects



(f) Understand Thermo concepts

Figure 6.1. Student distribution on the five-level Likert scale questions for 2015 and 2016 - I feel writing comments within the sample code helped me to: (a) Understand the examples (2015); (b) Solve the Homework Assignments (2015); (c) Understand Thermo Concepts (2015); (d) Understand the examples (2016); (e) Solve the Homework Assignments (2016); (f) Understand Thermo Concepts (2016);

Additional insights regarding the affordances of writing in-code comments for the worked-examples were provided at the end of the semester. Nineteen students answered to the question: "Please include any additional ideas in which you think that commenting the examples or the lab sessions supported your learning process in this course". Their responses were distributed as follows: four students considered this was helpful to better understand how the code works, three students used it to learn how to use these programs to solve other THERMO problems, three students identified the underlying equations of the model while writing comments, two students said it helped to understand how computation can support thermodynamics models, two students thought the examples were useful to visualize THERMO concepts, one student became familiar with Python syntax, and one student was able to connect individual lines of code with the overall purpose of the example. Two quotes from these responses are:

> **Student One:** "The commentary helped me better understand how the Python session was connected to the thermodynamics problem it modeled. [Understanding how computation can support THERMO]"
>
> **Student Two:** "I think commenting helps me look the code more carefully, and in a more general view, rather than just focus one line, but a section of code. [Connect individual lines with overall purpose]"

Finally, 30 students made suggestions to the computational modules. Ten students would like to have additional support in terms of Python programming, while nine students would like to have more transparency to the libraries, being able to access and even edit the underlying functions of the GIBBS framework: "More explanation of what the built-in functions do, like comon tangentsolver or PhaseDiagramSolver". Five students would like to have more sessions dedicated to computation in the context of THERMO, and three students suggested having more coding activities during the sessions. Two students suggested changes in the graphical user interface, and one student would like to work in a locally installed version of the GIBBS framework instead of using the online version.

6.2 Characteristics of Students' Explanations

The previous section showed how the implementation of in-code commenting activities helped students: to better understand the examples, see the connection between computation and thermos, and to even get familiar with the Python syntax. In this section, we explore the characteristics of students' explanations based on the analytical framework described in section 4.2.

Forty-three students enrolled in thermodynamics of materials participated in this part of the study. As part of the weekly homework, students were asked to comment the worked-example code explaining what each line was doing, and to make changes to the code. The submission of the homework assignment was due one week after the computational module had been implemented. We analyzed the written comments to answer the following guiding research questions:

- What are the characteristics of students' explanations in a black box approach to CSE education?
- How does the characteristics of students' explanations in a black box approach to CSE education change over time?
- What common misunderstandings in programming can be identified from students' explanations in a black box approach to CSE education?

6.2.1 Data Collection and Data Analysis

Students' comments were analyzed using the framework described in section 4.2. The types of sections that were identified for these examples slightly varied with respect to the ones in the other context: (1) importing libraries when packages or libraries are being imported to be used into the code; (2) setting up Problem Parameters when a value related to the problem is computed ; (3) setting up Supporting Variables when other variables that are used to solve the problem are computed; (4) creating GUI Object when an object to control the graphical user interface is created; (5) creating Function when a function is been declared; (6) setting up GUI when sliders and user interface objects are being configured; (7) validation when an if-clause starts; (8) validating Result when the result is being computed and sometimes printed for validation ; (9) launching Program when the graphical user interface is related to the created functions and launched.

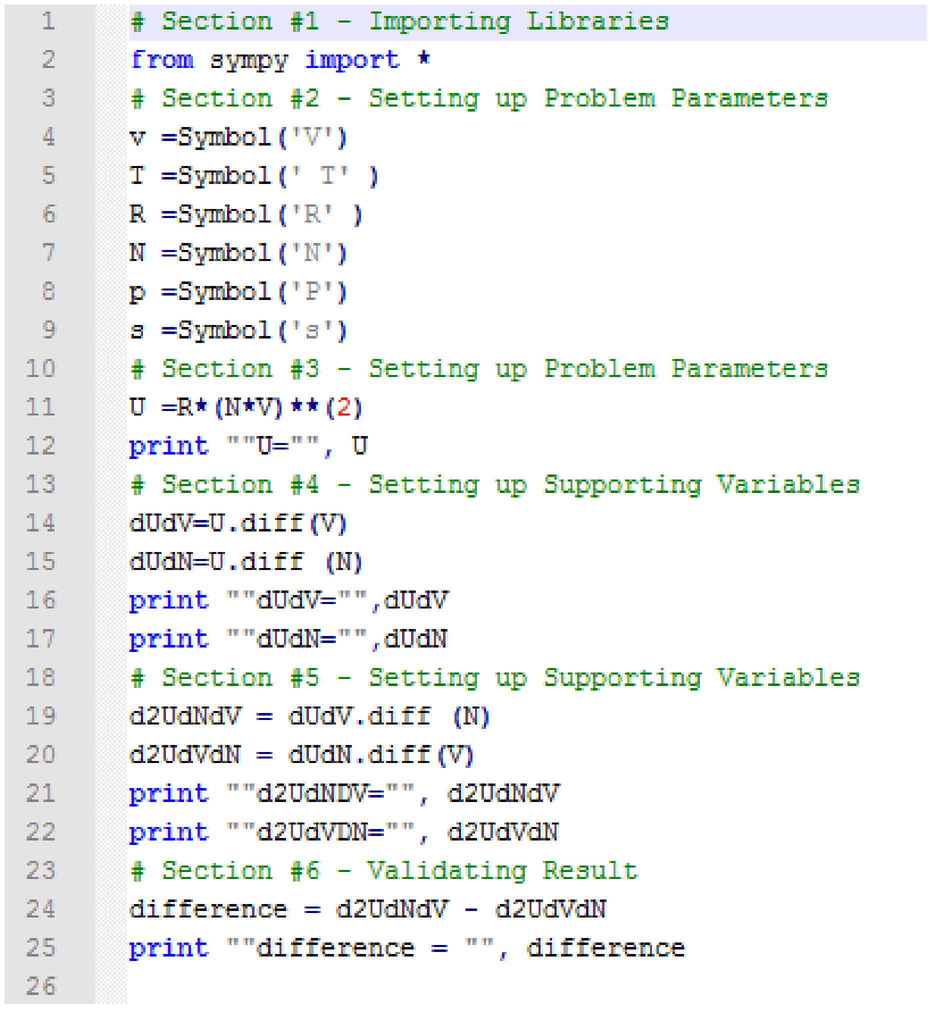
Module #1

The purpose of the first computational module was for the students to get familiar with the VKML user interface and Python programming. The example was a simple sequence of steps to identify whether a given function was a state function or not. A state function is “a mathematical property of a material system that mathematically describes the equilibrium state of the system independently on what process the system followed to arrive at that condition” (DeHoff, 2006). Thus, the way to validate a state function is by differentiating the function with respect to one variable, and then differentiate to a second variable. If this process is repeated in reverse order, and the result is the same, the function is a state function.

The sample Python code is presented in Figure 6.2 divided into six sections. The first two sections imported the required libraries to deal with symbolic variables, and defines the symbolic variables. The third section defined the function U to be tested. Sections four and five differentiated the function first with respect to one variable and then with respect to the other one. The final section validated whether the function was a state function or not by subtracting one of the derivatives from the other one. The result must be zero if U was indeed a state function.

Module #2

The second computational module focused on the creation of an interactive free energy plot in which the user could modify the variables omega, enthalpy, and melting points from the graphical user interface (GUI). The Python code (Figure 6.3 started by importing the required libraries in section one: the regular free energy variable, the binary solver, the graphical user interface, and the plot viewer. The second section created the plot object, where the free energy plot is presented. The



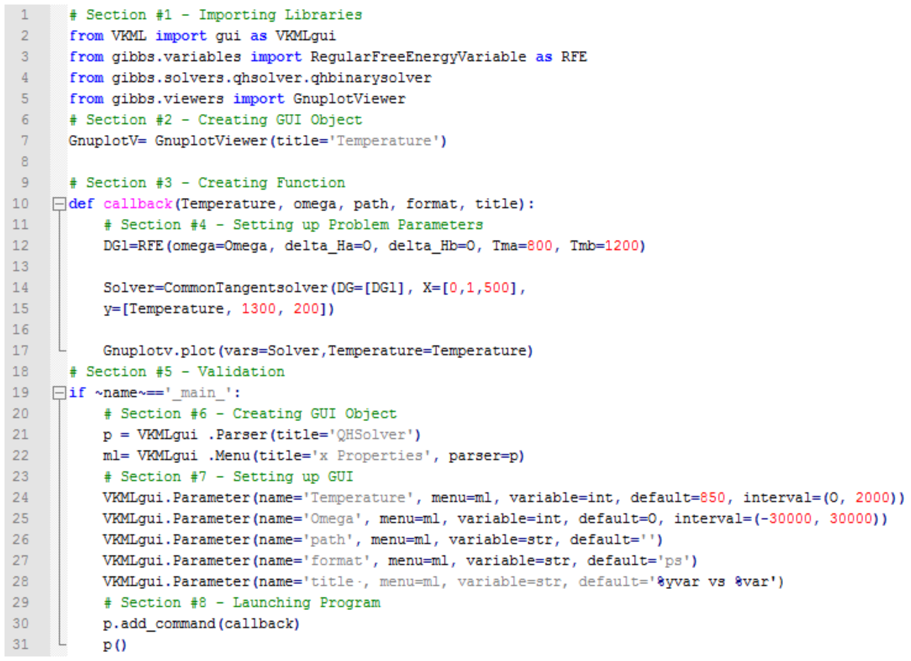
```
# Section #1 - Importing Libraries
from sympy import *
# Section #2 - Setting up Problem Parameters
v =Symbol('V')
T =Symbol(' T' )
R =Symbol('R' )
N =Symbol('N')
p =Symbol('P')
s =Symbol('s')
# Section #3 - Setting up Problem Parameters
U =R*(N*V)**(2)
print ""U="", U
# Section #4 - Setting up Supporting Variables
dUdV=U.diff(V)
dUdN=U.diff (N)
print ""dUdV="",dUdV
print ""dUdN="",dUdN
# Section #5 - Setting up Supporting Variables
d2UdNdV = dUdV.diff (N)
d2UdVdN = dUdN.diff(V)
print ""d2UdNDV="", d2UdNdV
print ""d2UdVDN="", d2UdVdN
# Section #6 - Validating Result
difference = d2UdNdV - d2UdVdN
print ""difference = "", difference
```

Figure 6.2. Python code for Module 1 - State Function

call back function created in section three encompasses the THERMO computational model created in section four. Section five, six, and seven initialized the graphical user interface and the parameters. The last section made the connection between the GUI and the callback function, and runs the GUI.

Module #3

The third module included an example that aimed at plotting a phase diagram and the common tangent lines associated with the two different phases of a



```python
# Section #1 - Importing Libraries
from VKML import gui as VKMLgui
from gibbs.variables import RegularFreeEnergyVariable as RFE
from gibbs.solvers.qhsolver.qhbinarysolver
from gibbs.viewers import GnuplotViewer
# Section #2 - Creating GUI Object
GnuplotV= GnuplotViewer(title='Temperature')

# Section #3 - Creating Function
def callback(Temperature, omega, path, format, title):
    # Section #4 - Setting up Problem Parameters
    DG1=RFE(omega=Omega, delta_Ha=0, delta_Hb=0, Tma=800, Tmb=1200)

    Solver=CommonTangentsolver(DG=[DG1], X=[0,1,500],
    y=[Temperature, 1300, 200])

    Gnuplotv.plot(vars=Solver,Temperature=Temperature)
# Section #5 - Validation
if ~name~=='_main_':
    # Section #6 - Creating GUI Object
    p = VKMLgui .Parser(title='QHSolver')
    ml= VKMLgui .Menu(title='x Properties', parser=p)
    # Section #7 - Setting up GUI
    VKMLgui.Parameter(name='Temperature', menu=ml, variable=int, default=850, interval=(0, 2000))
    VKMLgui.Parameter(name='Omega', menu=ml, variable=int, default=0, interval=(-30000, 30000))
    VKMLgui.Parameter(name='path', menu=ml, variable=str, default='')
    VKMLgui.Parameter(name='format', menu=ml, variable=str, default='ps')
    VKMLgui.Parameter(name='title·, menu=ml, variable=str, default='%yvar vs %var')
    # Section #8 - Launching Program
    p.add_command(callback)
    p()
```

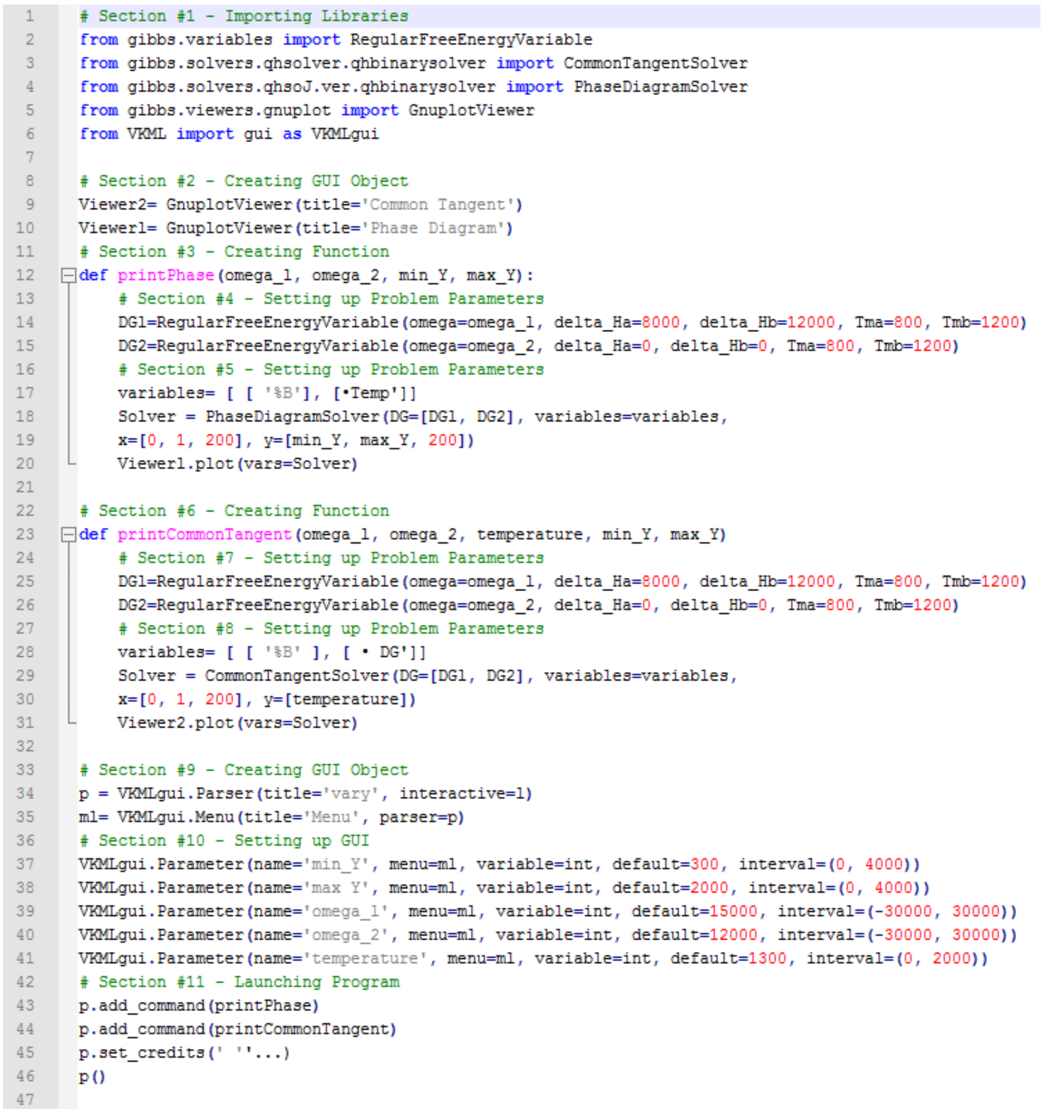
Figure 6.3. Python code for Module 2 - Free Energy Plot

material. The Python code presented in Figure 6.4 is somewhat similar to the one for the previous module but has two important changes. First, two functions were created to be controlled from the graphical user interface. Section three created the function that configures and plots the phase diagram. Section six created the function for configuring and plotting the common tangent lines. Hence, the parameters to change these plots were different to the previous ones, and the parser p needed to connect the two functions with the GUI. The second change was that two phases (DG1 and DG2) were defined within each function.

6.2.2 Patterns within each activity

Module #1

Forty-three students submitted their in-code explanations as part of their weekly homework assignment. Figure 6.5 presents the distribution of students' use



```python
# Section #1 - Importing Libraries
from gibbs.variables import RegularFreeEnergyVariable
from gibbs.solvers.qhsolver.qhbinarysolver import CommonTangentSolver
from gibbs.solvers.qhsoJ.ver.qhbinarysolver import PhaseDiagramSolver
from gibbs.viewers.gnuplot import GnuplotViewer
from VKML import gui as VKMLgui

# Section #2 - Creating GUI Object
Viewer2= GnuplotViewer(title='Common Tangent')
Viewer1= GnuplotViewer(title='Phase Diagram')
# Section #3 - Creating Function
def printPhase(omega_1, omega_2, min_Y, max_Y):
    # Section #4 - Setting up Problem Parameters
    DG1=RegularFreeEnergyVariable(omega=omega_1, delta_Ha=8000, delta_Hb=12000, Tma=800, Tmb=1200)
    DG2=RegularFreeEnergyVariable(omega=omega_2, delta_Ha=0, delta_Hb=0, Tma=800, Tmb=1200)
    # Section #5 - Setting up Problem Parameters
    variables= [ [ '%B'], [•Temp']]
    Solver = PhaseDiagramSolver(DG=[DG1, DG2], variables=variables,
    x=[0, 1, 200], y=[min_Y, max_Y, 200])
    Viewer1.plot(vars=Solver)

# Section #6 - Creating Function
def printCommonTangent(omega_1, omega_2, temperature, min_Y, max_Y)
    # Section #7 - Setting up Problem Parameters
    DG1=RegularFreeEnergyVariable(omega=omega_1, delta_Ha=8000, delta_Hb=12000, Tma=800, Tmb=1200)
    DG2=RegularFreeEnergyVariable(omega=omega_2, delta_Ha=0, delta_Hb=0, Tma=800, Tmb=1200)
    # Section #8 - Setting up Problem Parameters
    variables= [ [ '%B' ], [ • DG']]
    Solver = CommonTangentSolver(DG=[DG1, DG2], variables=variables,
    x=[0, 1, 200], y=[temperature])
    Viewer2.plot(vars=Solver)

# Section #9 - Creating GUI Object
p = VKMLgui.Parser(title='vary', interactive=1)
m1= VKMLgui.Menu(title='Menu', parser=p)
# Section #10 - Setting up GUI
VKMLgui.Parameter(name='min_Y', menu=m1, variable=int, default=300, interval=(0, 4000))
VKMLgui.Parameter(name='max Y', menu=m1, variable=int, default=2000, interval=(0, 4000))
VKMLgui.Parameter(name='omega_1', menu=m1, variable=int, default=15000, interval=(-30000, 30000))
VKMLgui.Parameter(name='omega_2', menu=m1, variable=int, default=12000, interval=(-30000, 30000))
VKMLgui.Parameter(name='temperature', menu=m1, variable=int, default=1300, interval=(0, 2000))
# Section #11 - Launching Program
p.add_command(printPhase)
p.add_command(printCommonTangent)
p.set_credits(' ''...)
p()
```

Figure 6.4. Python code for Module 2 - Free Energy Plot

of the five types of knowledge within the code sections. The declarative knowledge was commonly used among the six sections, while the procedural knowledge was only used in sections three to six, only by a group of students who described what would happen during execution time (EXE). The schematic knowledge was mostly used in sections two and three, where students made connections between the variables and the state function to the THERMO principles. Four of these students actually brought principles that were not evident in the example by mentioning that the differentiation that was carried out corresponded to the Maxwell equations: “using same sympy functions to take cross derivatives of dudn and .dudv. to check is maxwells equations hold.” The strategic knowledge was almost limited to section six, when students described the conditions of the difference to conclude that U was a state function: “The difference line test to see if (du/dUdN)=(dU/dNdV), in which case the function is a state function”. Twelve students showed limited knowledge in at least one of the sections. These instances of limited knowledge ranged from incorrect statements (INC) to paraphrasing (PHR) explanations

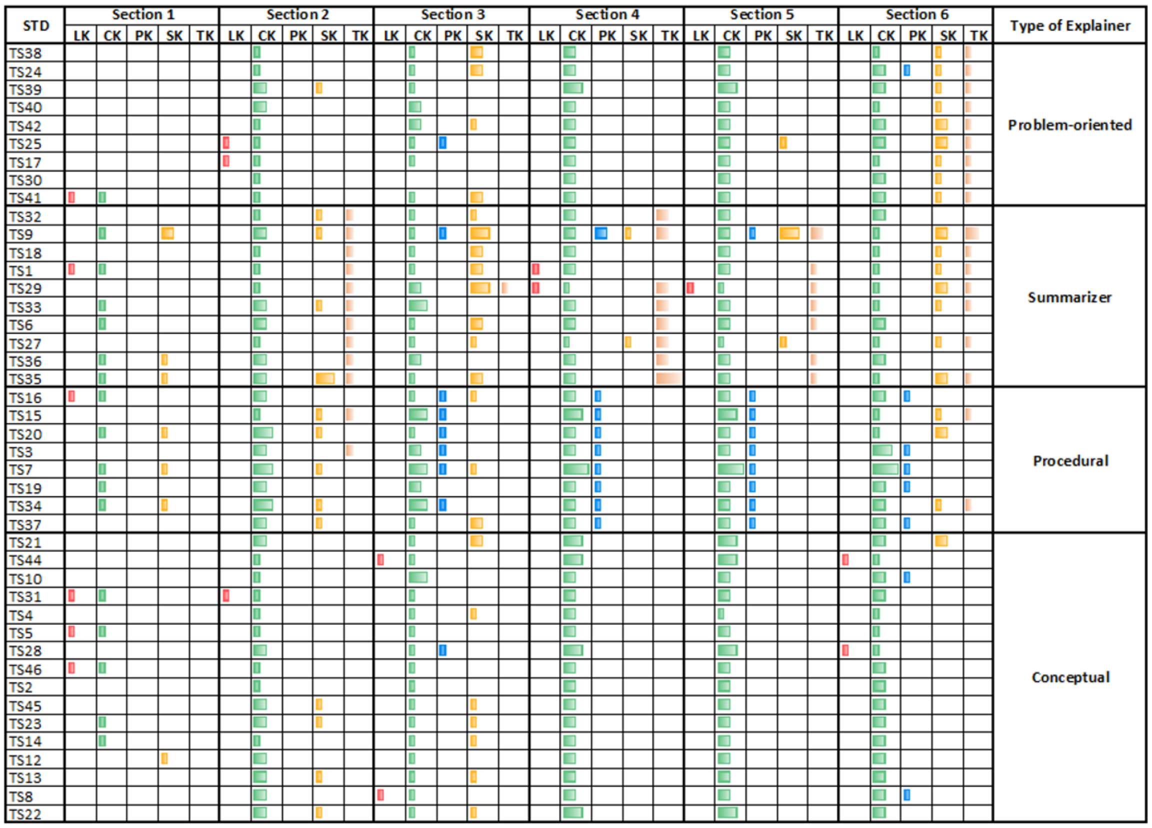


Figure 6.5. Students’ use of the types of knowledge for each section of the code: Module #1

Four clusters were identified from the types of knowledge students used along the six sections: problem-oriented (TS38, TS24, TS39, TS40, TS42, TS25, TS17, TS30, and TS41); summarizers (TS32, TS9, TS18, TS1, TS29, TS33, TS6, TS27, TS36, and TS35); procedural (TS16, TS15, TS20, TS3, TS7, TS19, TS34, and TS37); declarative (TS21, TS44, TS10, TS31, TS4, TS5, TS28, TS46, TS2, TS45, TS23, TS14, TS12, TS13, TS8, and TS22). Since many students did not explain the first section, where the libraries were imported, and those who did only mention that something was being imported, this section was not considered for the clustering process.

The first group (problem-oriented) did not explain the first section (i.e., importing libraries) and did not identify any group of instructions with common purpose (CHK). These students rarely showed any instances of procedural or schematic knowledge, and only did it to make reference to the problem. Moreover, all of them identified the conditions of application of actions (CON) in the validation of results section, making appropriate connections to the problem (PRO).

The second group corresponded to the summarizer explainers, who also identified these conditions and made connections to the problem at the end of the code, but also identified chunks of code with certain goal, and highlighted these goals. These students rarely used background knowledge (BGK) and never explained the rationale for certain instructions (WHY). The third group of students comprised the procedural explainers, who consistently talked about the problem in execution time throughout sections three, four, five, and six, while the fourth group, comprising declarative explainers mostly focused on saying what the instruction does (COA), and which parameters uses (PAR). Figure 6.6 (b f) shows the characteristics of students' explanations for sections two to six.

Overall, we see that the definition of the variables in the second section (Figure 6.6 (b)) was the one with the largest number of connections to background principles (BGK 12), since the students described what these variables represented using THERMO principles: "the symbols for volume, Temperature, R, Pressure, Entropy and N are being defined as variables". The third section (Figure 6.6(c)), in which the function to be tested was defined, was the first one to present connections to the problem (PRO 17 students) to define of the overall goal (GOA 13 students). In the sections in which the derivatives were calculated (four and five Figure 6.6(d, e)), almost all students described the parameters for these derivatives. Interestingly, students did not include (or very rarely) any explanations as how something was done (HOW only one student) nor a rationale for any given instruction (WHY one student).

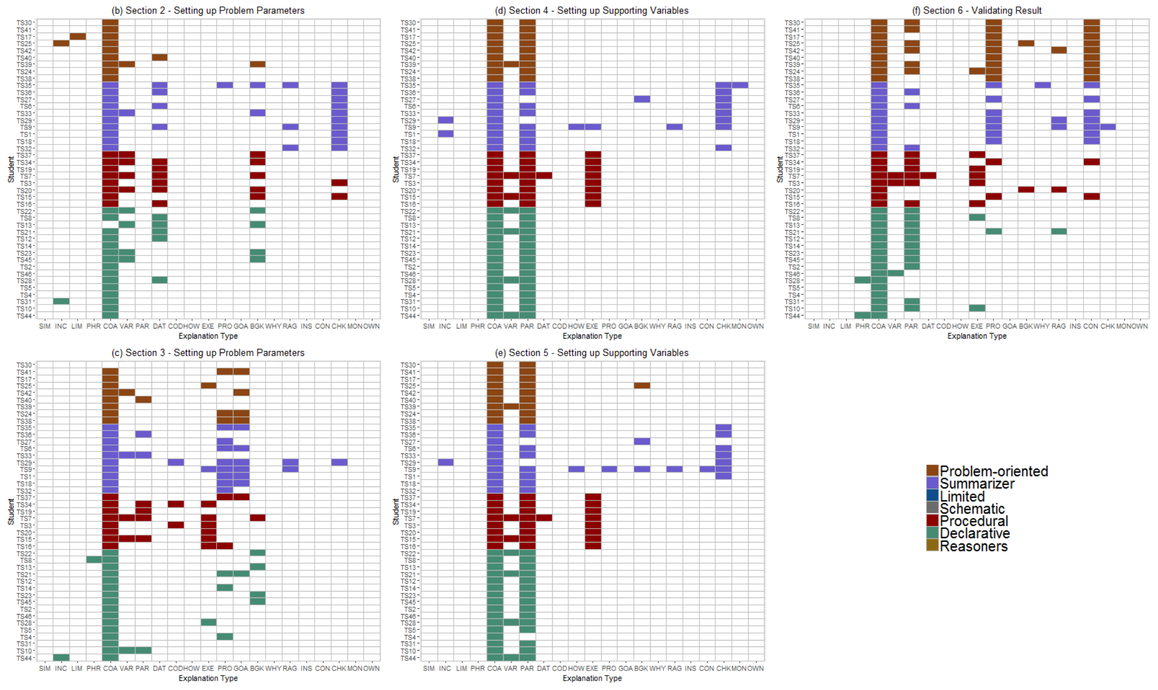


Figure 6.6. Patterns of students' explanations by section type: Module #1

Module #2

Thirty six students submitted their explanations for the second module. Sections two and six often showed several instances of limited knowledge involving a common misunderstanding that students had: students seemed to guide their interpretation of the instructions by the parameters these received. For instance, when the plot viewer variable was created in section two, students often explained that this instruction "#Set[s] a title to GUI". While the instruction did set a title for the GUI, this was a very limited explanation. The main objective of this instruction was to create a plot object that could be used to show the free energy diagram. As part of this process, the code gave a title to this object. The same misunderstanding was commonly identified in section six. Figure 6.7 depicts the distribution of the types of knowledge students used within each section of the worked-example.

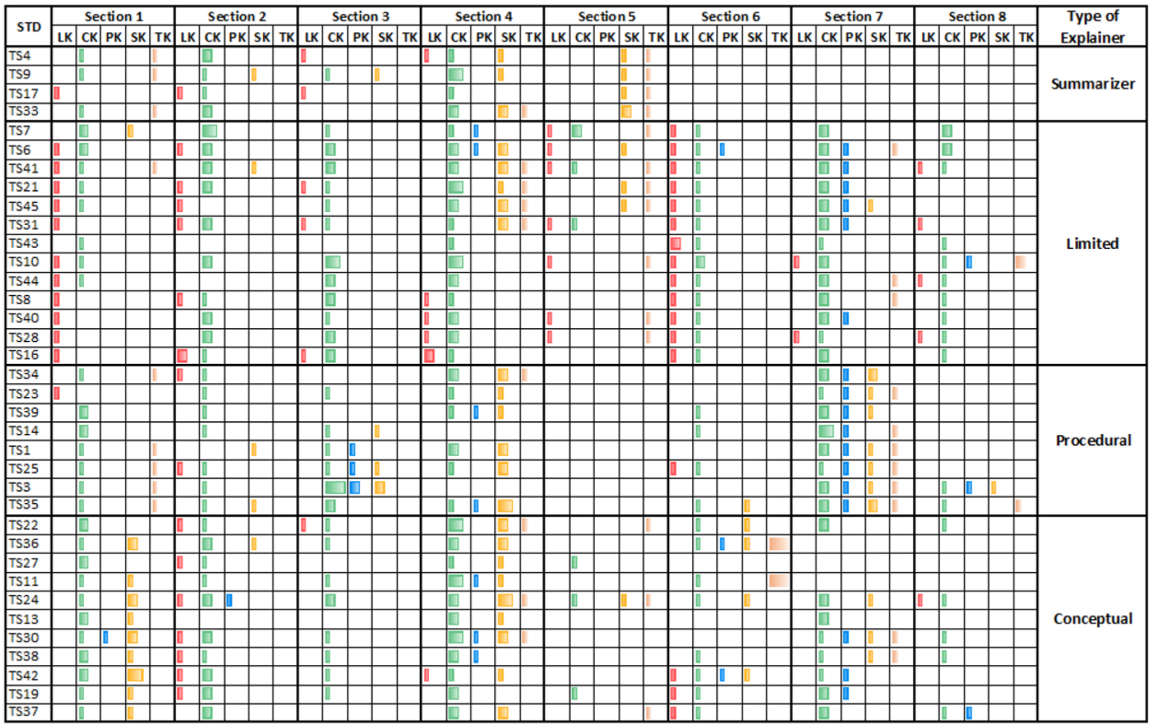


Figure 6.7. Students' use of the types of knowledge for each section of the code: Module #2

Regarding the other types of knowledge, the largest number of instances of procedural knowledge was found in section seven, when students talked about the interaction of the user with the menu. The schematic and strategic knowledge often appeared in section four, when the actual THERMO variables were being configured, and several students either did not explained the section three or limited their explanations to declarative knowledge. This section involved the definition of a function to configure the parameters of the problem, but these students often just said things like: “## defines a function, and calls certain variables to be in the function”.

The cluster analysis for student explanations related to the second computational module revealed four groups of explainers: summarizer (TS4, TS9, TS17, TS33), limited (TS7, TS6, TS41, TS21, TS45, TS31, TS43, TS10, TS44, TS8, TS40, TS28, TS16), procedural (TS34, TS23, TS39, TS14, TS1, TS25, TS3, TS35), and declarative (TS22, TS36, TS27, TS11, TS24, TS13, TS30, TS38, TS42, TS19, TS37) explainers. The patterns of their explanations within each section of code are presented in Figure 6.8. The first group (i.e. summarizer) corresponded to four students who wrote summarizing explanations for the chunks of code they identified. As a consequence, they did not write explanations for sections six, seven, and eight. Instead, they grouped them all using a single explanation in section five, where they described the goal of these chunks: “gui where the user can control the temperature and initial omega value”.

The limited explainers depicted multiple instances of limited knowledge in sections one, two, four and six. These instances were mostly limited explanations of instructions where students only described part of the purpose of an instruction. This group only showed sporadic instances of schematic and strategic knowledge in the fourth section to make connections to the problem.

The procedural explainers focused on how this program would execute, writing multiple instances of the interaction between the user and the GUI in section seven. This group also used instances of procedural, schematic, and strategic knowledge in sections four and seven, but did not explained section five.

The fourth group corresponded to the declarative explainers, who consistently used declarative knowledge to explain what the program did throughout all sections, but section four. In section four, some of these explainers made connections to the problem either by talking what some variables represented: “the melting temperature at a, and the melting temperature at b”; or by clarifying

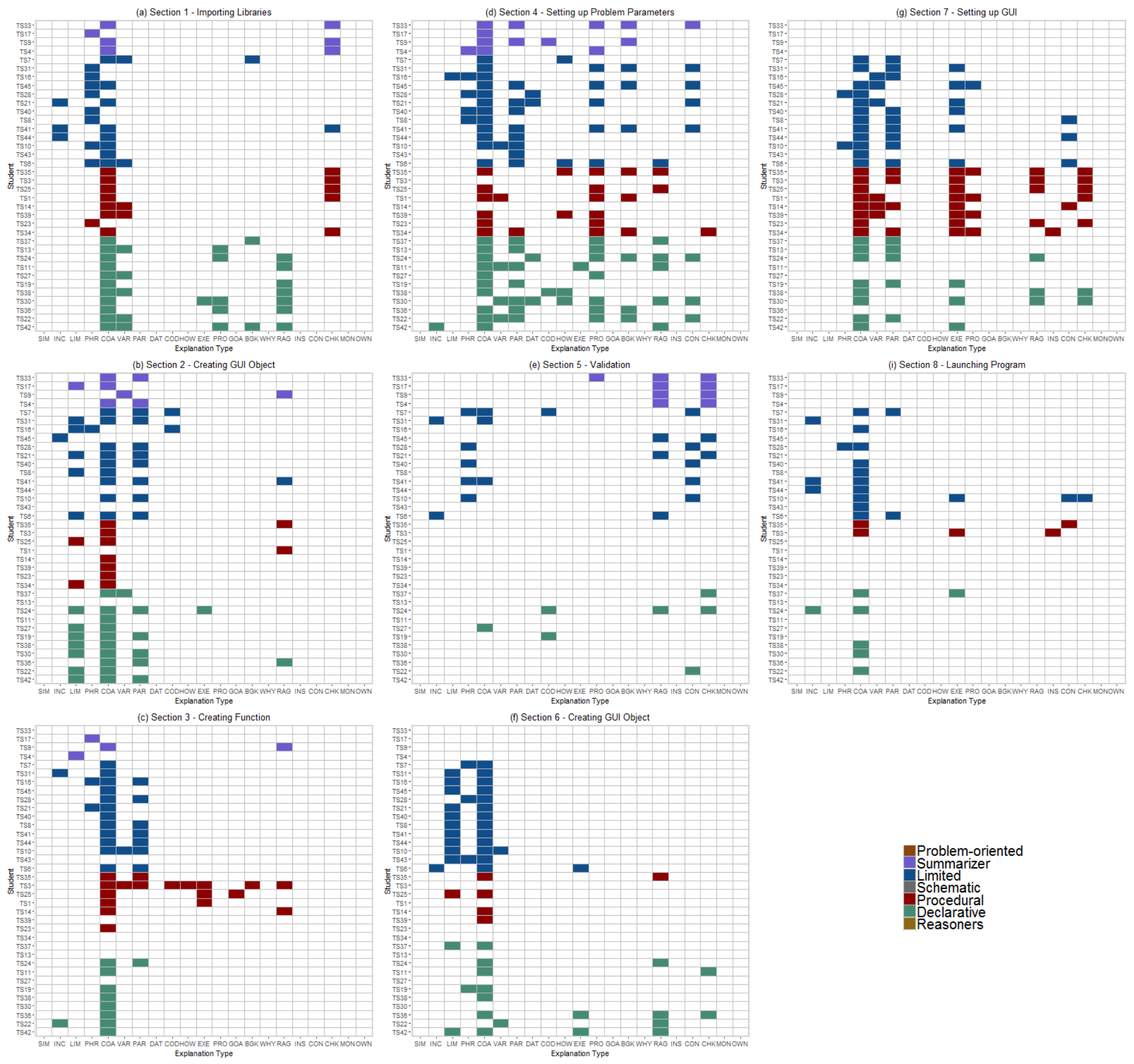


Figure 6.8. Patterns of students' explanations by section type: Module #2

what the plot was about: "plots the graph of Gibbs vs. Temperature in the GUI". However, this was the case for all the groups of explainers in this section.

Overall, this module had all students writing some kind of explanation for the first section (Figure 6.8(a)), where libraries where imported. This contrasts with the first module where only 40% of the students explained that section. However, 10 students paraphrased this instructions (PHR), and only seven students made connections with the goal of importing this libraries (RAG). Students from all groups showed instances of limited explanations in section two, where as described above, their explanations focused on setting a title and not creating a handler for the plot.

Only one student mentioned what was the actual goal of the function (GOA), and none of the students mentioned the rationale for the instructions (WHY). In general, the schematic knowledge was limited to the connections to the problem (PRO), the use of background knowledge (BGK) and the relationship action goals (RAG), especially in section four, when setting up problem parameters. Strategic knowledge was limited to the conditions of application of actions (CON) both in sections four and five.

Module #3

Thirty-five students submitted explanations for the example of this computational module. Figure 6.9 presents the types of knowledge students used within each section of the code. The most used type of knowledge was the declarative knowledge across the sections. Similar to what was described in the worked-example for Module #2, several students wrote limited explanations for sections two and nine, where the GUI objects were being created. However, four and eight students also showed limited knowledge in sections five and six correspondingly, where they paraphrased the configuration of the model, depicting unclear understanding of this configuration. Only four students (TS3, TS20, TS23, TS24) employed the four types of knowledge within one section.

The schematic knowledge was mostly used in sections four, six, seven, and eight, which purpose was to configuring parameters related to the problem, while the strategic knowledge was mostly employed in sections seven and eight, identifying chunks of code with certain purpose. Also, the use of procedural knowledge was

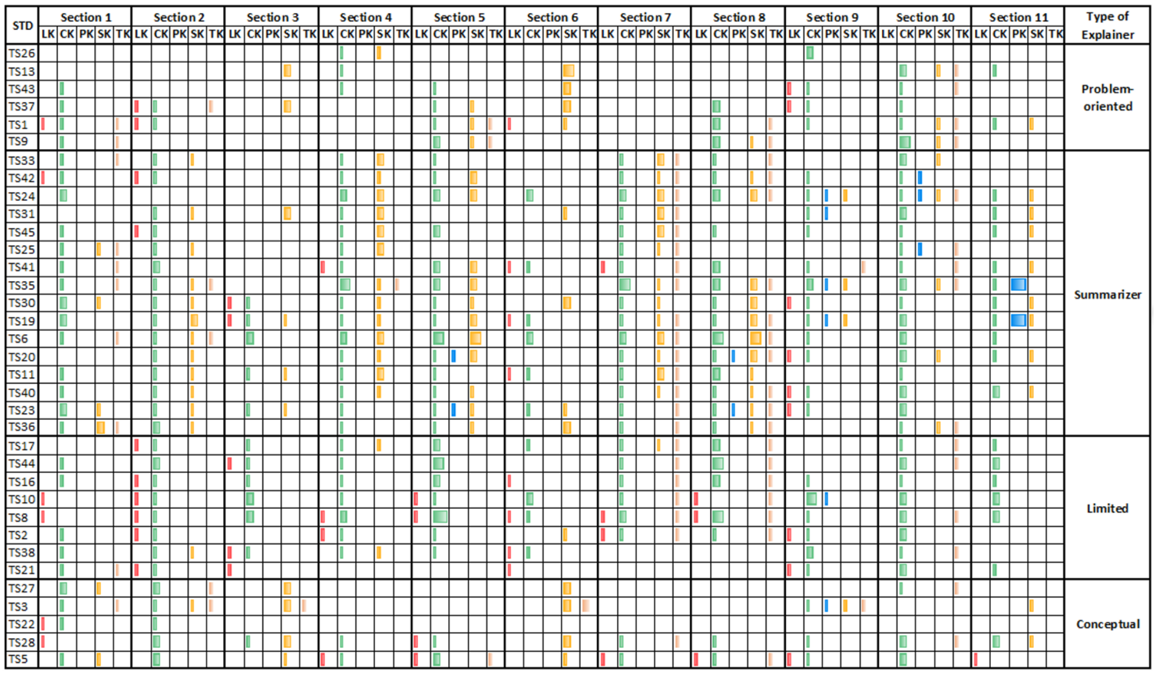


Figure 6.9. Students' use of the types of knowledge for each section of the code: Module #3

very low, with a higher frequency within the last three sections, which involved user interface, and execution time and almost non-existent in the rest of the sections.

Four groups of explainers where identified for this activity: problem-oriented (TS26, TS13, TS43, TS37, TS1, TS9), summarizer (TS33, TS42, TS24, TS31, TS45, TS25, TS41, TS35, TS30, TS19, TS6, TS20, TS11, TS40, TS23, TS36), limited (TS17, TS44, TS16, TS10, TS8, TS2, TS38, TS21), and declarative (TS27, TS3, TS22, TS28, TS5). The first group of explainers (problem-oriented) used multiple instances of declarative knowledge and made consistent connections to the problem in the sections where the parameters were being configured (i.e., sections four, five, seven, and eight). This group of explainers also used background knowledge to describe what certain variables represented. However, they were not considered schematic explainers because they did not explain why the instruction works as it is.

The summarizer explainers was the second group of explainers, who only wrote comments for six out of 11 sections of the code. These explanations were mostly focused on identifying chunks of code and describing the goal of these groups of instructions. The third group comprised the limited explainers because they often wrote limited or incorrect explanations, and these were merely based on declarative knowledge. The fourth and final group did not often use schematic nor strategic knowledge, and instead limited their explanations to simple descriptions of consequences of actions such as “solves equations” and “plots data”. The patterns of the explanations for the four clusters of students that were identified in this module are presented in Figure 6.10.

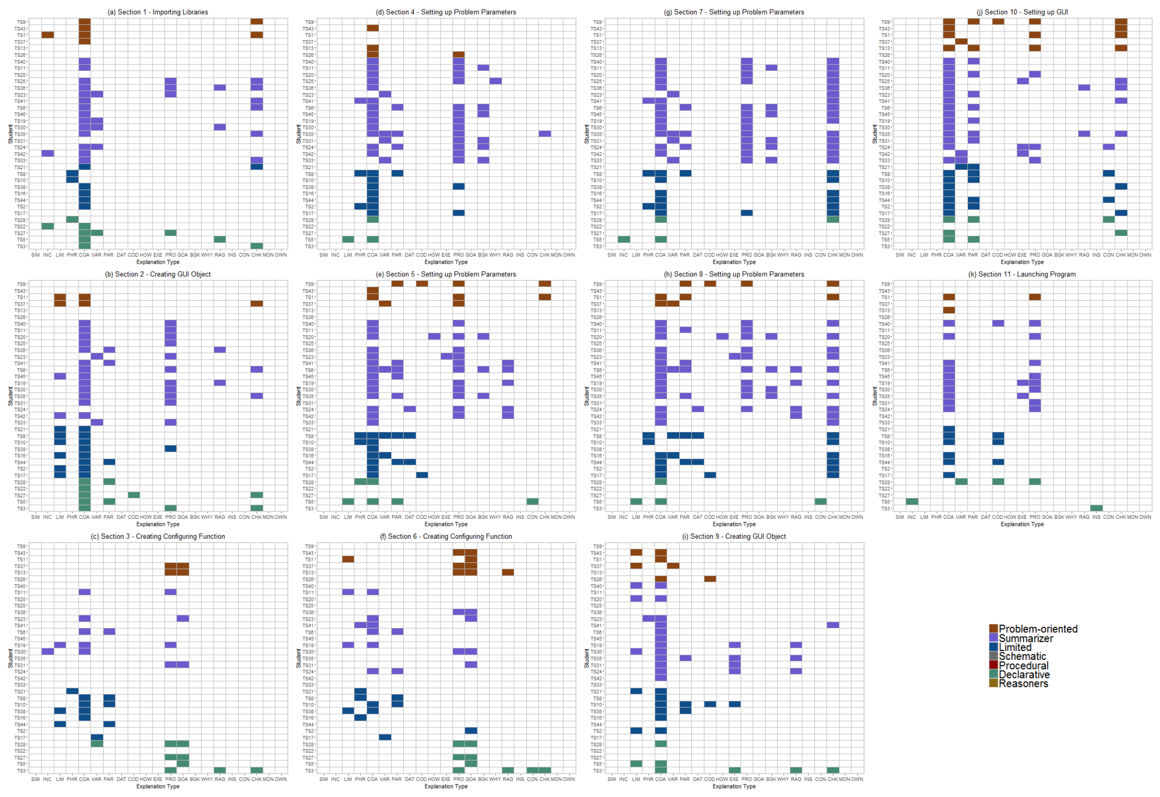


Figure 6.10. Patterns of students’ explanations by section type: Module #3

Overall, this activity had students actively explaining all sections, although a small group of them preferred to summarize their explanations in sections three, six, and ten (Figure 6.10 (c,f,j)). Sections related to setting up problem parameters (i.e., four, five, seven, and eight) involved explanations that made these connections to the problem. The number of limited explanations decreased as compared to the previous module, but again, no instances of the rationale (WHY) for the given instructions were identified.

6.2.3 Results

What are the characteristics of students' explanations in a black box approach to CSE education?

Students' explanations in the black box approach involved the use of all types of knowledge in different proportions (see Figure 6.11). Declarative knowledge was the most widely used type of knowledge across the three different computational modules being at least 50% for each explanation activity. Students often described what certain section of the code did, but did not explain how, why, or under which conditions it worked.

The schematic was the second most used type of knowledge, usually to make connections to the problem, and to a lesser extent, to bring background knowledge into their explanations and to describe the relationship action-goals. The limited knowledge peaked on the modules #2, where students showed a common misunderstanding of the use of parameters while creating objects, which remained present in a lower degree, for module #3. The strategic knowledge was consistently used around 10% of the times, and the procedural knowledge was the least used type of knowledge across all three modules.

There were also differences in the number of occurrences for each category within each type of knowledge (Figure 6.12). These values need to be taken into perspective considering the number of possible instances based on the number of commentable sections and the number students submitting explanations for each module: (1) 43 students, six sections; (2) 36 students, eight sections; and (3) 35

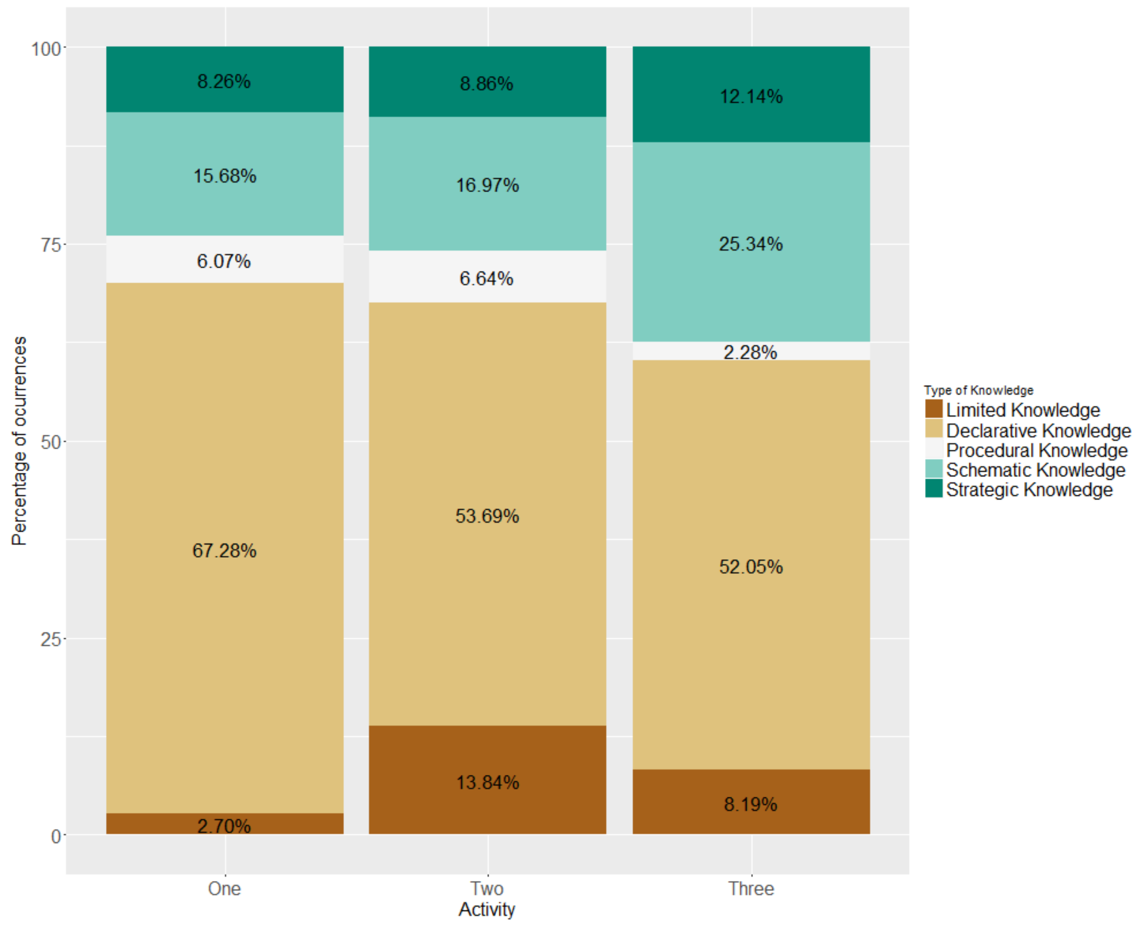


Figure 6.11. Percentage of occurrences of the types of knowledge within each module

students, 11 sections. Assuming that there was one section students did not comment in each exercise (e.g. importing libraries), the total possible number of instances would be (43 * 5) + (36 * 7) + (35 * 10) = 817.

Student use of declarative knowledge corresponded to more than 50% in all the three activities, and this was mostly thanks to the large number of explanations that involved consequences of actions (COA 653), which is almost three times the next category (PAR 234), and almost four times the number of connections to the problem (PRO 170). This result suggests that only one fourth of the students' explanations described the consequences of actions. These explanations were

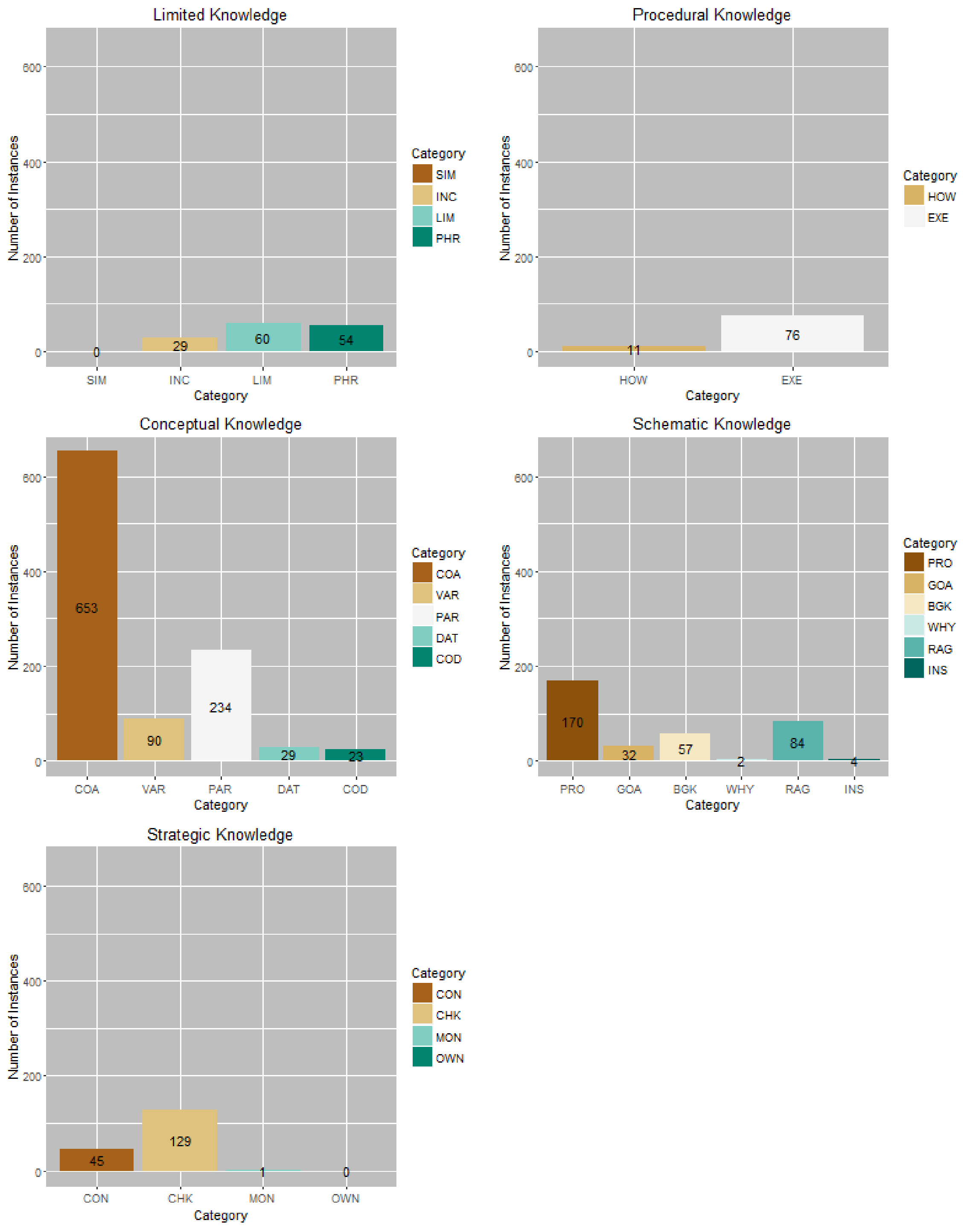


Figure 6.12. Number of instances of each category within the types of knowledge

connected to the problem, and only once every 10 times students connected to background knowledge (BGK 57). Furthermore, there were almost no explanations of the rationale (WHY 2), and only 32 descriptions of the goal of certain section (GOA).

The limited explanations (LIM 60 instances) and paraphrasing (PHR 54) where the most common ones within the identified instances of limited knowledge, but a significant number of incorrect statements (INC 29) were also identified. The few instances of procedural knowledge (87 in total) were mostly due to descriptions of user interaction in execution time (EXE 76). Likewise, three quarters of the instances of strategic knowledge were related to the identification of chunks of the program (CHK 129), and the remaining third focused on identifying the conditions for these instructions (CON 45).

How does the characteristics of students' explanations in a black box approach to CSE education change over time?

Five types of explainers were identified within this context: limited, declarative, problem-oriented, summarizer, and procedural. The limited explainers were those students who consistently showed limited or incorrect explanations. The declarative explainers focused on simply describing what an instruction or a section did, without actually explaining how, why, or under what conditions this worked. The third group, problem-oriented explainers, was similar to the declarative explainers, but made multiple connections to the problem when describing the consequences of actions, or what certain variables represented. The summarizer explainers usually did not write comments in all sections but summarized them in a single one, identifying chunks of code with certain purpose, while the procedural explainers described user interactions with the program during execution time.

Figure 6.13 depicts the distribution of the groups of explainers within each activity. The first module did not show any limited explainers, probably because the code was rather a simple sequence of instructions with no complex data structures, loops or functions. However, this group corresponded to 36.11% of the students in module 2 and 22.86% in module 3. Declarative explainers were 37.21% of the students for module 1 and 30.56% in module 3, but only 14.29% in module 2. One possible explanation is that this group of students moved to the limited explainers, who peaked in module 2 (36.11%) with a common misunderstanding, which was also present in module 3 (22.86%). Likewise, the group of students who made consistent connections to the problem was only present in modules one and three.

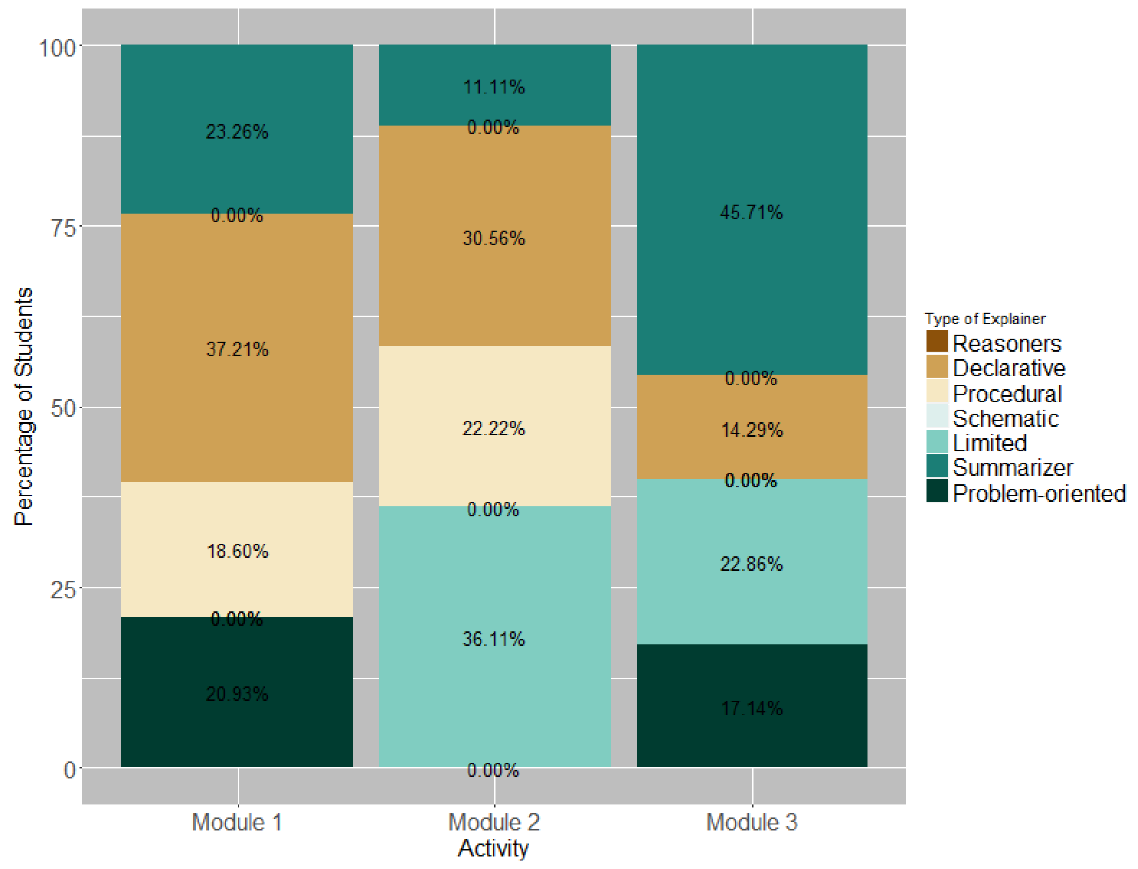


Figure 6.13. Distribution of types of explainers between activities: THERMO

What common misunderstandings in programming can be identified from students' explanations in a black box approach to CSE education?

Students in the black box context showed a large percentage of explanations depicting limited knowledge, especially for modules #2 and #3. These explanations were distributed among limited explanations, paraphrasing, and incorrect explanations. The first module did not include limited knowledge, probably because it was a simple sequence of instructions that did not require complex programming experience. Among the instances of limited knowledge, the most common one was a false assumption that the parameters determine the purpose of an instruction. For instance, in the lines of code such as this one: Viewer2= GnuplotViewer(title='Common Tangent'), many comments would be like “titles a graph window”, “creates the graph title”, or “give title to the plot in Viewer2:Common Tangent”. Although it is true that the program assigned a title by passing the parameter, the purpose of the instruction was actually to create a GnuplotViewer object, which would be titled Common Tangent, and would be used to invoke methods that showed the resulting plots.

Another difficult concept for this group of students was to understand the use of the callback function in the module #2. Seven students did not explain this function, 11 students either paraphrased or wrote simple consequences of actions (e.g., “Creates a function callback”, “callback variables”) and two students related this function that would be later connected to the graphical user interface to an actual loop statement: “Creates a loop in which variables pass between”, “starts callback loop with 5 variables”.

Finally, students did often not talk about objects or libraries in any of their explanations. In fact, one of the students considered that importing all elements from the Sympy library in module #1 corresponded to “Import [a] file”, other students talked about it as a database, a directory of functions, or an interface. Likewise, in modules #2 and #3 students assumed that importing the regular free energy variable from gibbs.variables was to "import value of RFE". However, this was actually a function from the library that would be later used with different parameters.

6.3 Relationship between Students' Explanations and Student Ability to Program

After identifying the different types of explainers in a black box context, we wanted to identify whether these characteristics had a relationship with student ability to do programming. Forty-three sophomore engineering students participated of this part of the study. These students completed a pre-survey at the beginning of the semester, and a pretest/posttest instruments before and after each computational module. The guiding research questions were:

- How do the characteristics of students' self-explanations in a black box context relate to relate to their perceived ability to program?
- How do the characteristics of students' self-explanations in a black box context relate to their performance in the course?

6.3.1 Data Collection

Each module started with a five-minute pretest that evaluated student understanding of the topics that would be represented in that session. These topics had already been presented during the two lectures sessions immediately before the computational module. The test consisted of a single open-ended question where students needed to identify certain properties or phases of a given material. For instance, in module #2 students created a free energy plot with modifying parameters: omega, enthalpy, and melting points. Students would analyze under which conditions this material would be stable and unstable. Therefore, the pretest/posttest instruments provided students with a similar plot and asked them to identify where the solution would be unstable (see Figure **??**).

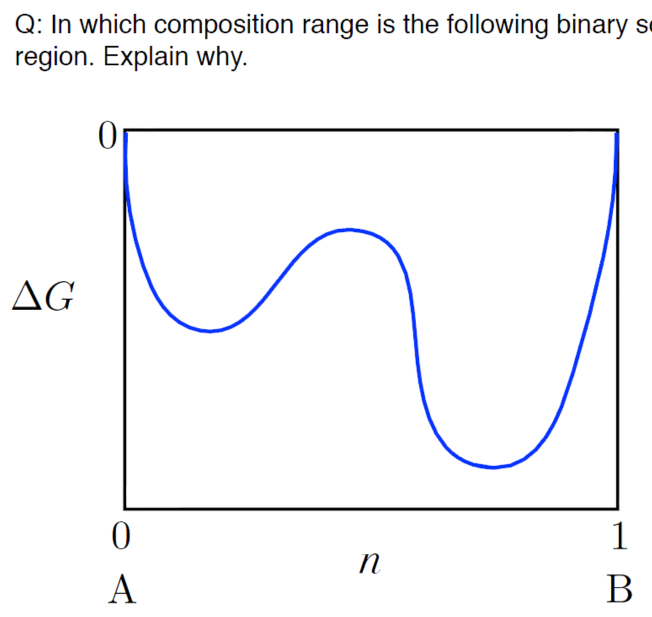

Q: In which composition range is the following binary solution unstable? Please shade the stable region. Explain why.

Figure 6.14. Distribution of types of explainers between activities: THERMO

After the pretest, the instructor of the course introduced the worked-example and the students played with the parameters to make inferences about the phenomenon. As part of a weekly homework assignment, students were required to write comments within the code to describe what the code was doing, as well as to modify part of the code to make it work for certain given conditions. One week after, when the homework was due, students started the next module with a posttest plus the pretest of the new module.

6.3.2 Data Analysis

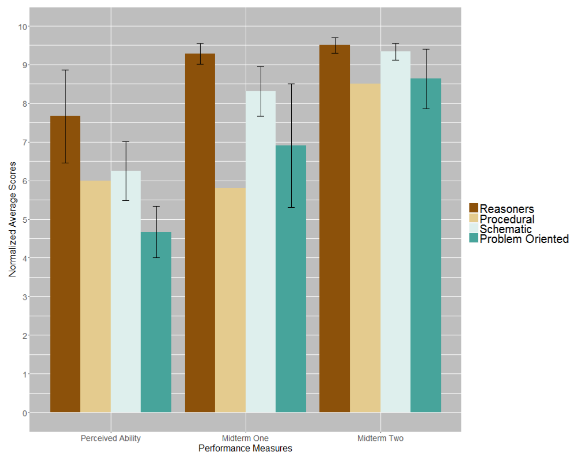
Students' explanations were qualitatively analyzed following the same process as in the glass box context. Students' in-code comments were categorized using the codes from Table 4.3, which were then used to identify types of explainers based on the types of knowledge they used in each section. The types of explainers that were identified for module #2 were: summarizer (TS4, TS9, TS17, TS33), limited (TS7, TS6, TS41, TS21, TS45, TS31, TS43, TS10, TS44, TS8, TS40, TS28, TS16), procedural (TS34, TS23, TS39, TS14, TS1, TS25, TS3, TS35), and declarative (TS22, TS36, TS27, TS11, TS24, TS13, TS30, TS38, TS42, TS19, TS37) explainers. Two new types of explainers were identified as compared to the ones previously described in the CPMSE course: summarizer and declarative explainers. The declarative explainers used mostly declarative knowledge in their explanations, saying what an instruction does. On the other hand, the summarizer explainers would regularly identify chunks of code with certain goal, and would highlight the relationship action goals. Similar types of explainers were identified in module #3: problem-oriented (TS26, TS13, TS43, TS37, TS1, TS9), summarizer (TS33, TS42, TS24, TS31, TS45, TS25, TS41, TS35, TS30, TS19, TS6, TS20, TS11, TS40, TS23, TS36), limited (TS17, TS44, TS16, TS10, TS8, TS2, TS38, TS21), and declarative (TS27, TS3, TS22, TS28, TS5).

The beginning of the semester survey asked the same two questions related to student ability to create a computer program: (1) [Likert Scale] I have the ability to design an algorithm; and (2) [Likert Scale] I have the ability to write a computer program. These were averaged to identify a composite score for student perceived ability to create computer programs. Both, the perceived ability and the pretest/posttest scores were employed to identify their relationship with the type of explainers.

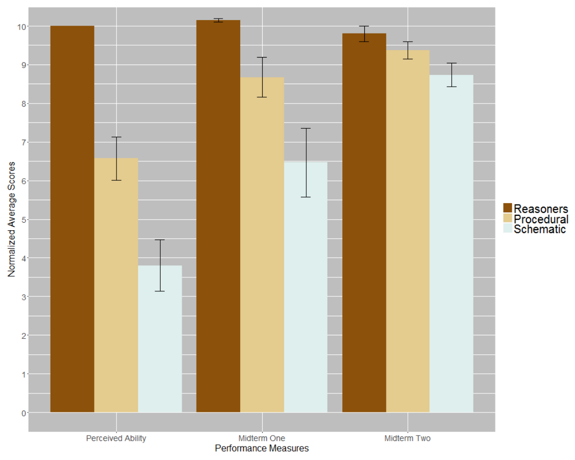
The pretest/posttest instruments were scored by the teaching assistant of the course in a scale 1-5. The gain for each student was computed as posttest-pretest. Furthermore, the gain score was employed to group students by low, mid, and high performers. A low gain score was considered to be zero or less, an intermediate gain comprised values between one and two, and a high gain score corresponded to gains of three or more.

6.3.3 Results

Figure 6.15 presents the comparison among these types of explainers on average student perceived ability, and average pretest/posttest performance. Although there were no significant differences among these groups regarding the student perceived ability, or pretest/posttest scores, we identified similar trends in both modules: (1) the limited explainers seem to have started with a lower perceived ability to program; (2) the declarative explainers showed the highest average performance on the pretest in both modules; and (3) the summarizers showed an intermediate gain from pretest to posttest. It is possible that students who felt confident after completing the pretest, did not see the value of reflecting deeply on the code. These students did not show a significant learning gain, in part because of the already high scores in the pretest.



(a) Module #2



(b) Module #3

Figure 6.15. Average differences in student performance by explainer type

The second part of the analysis consisted of identifying whether the extension of their explanations and the number of identified categories within their comments were different based on their ability and their disciplinary knowledge. Figure 6.16 shows the number of explanations based on the students' perceived ability (a,d), pretest score (b, e), and posttest-pretest gain (c, f). There were no significant differences nor clear patterns among students' explanations for any of the performance measures.

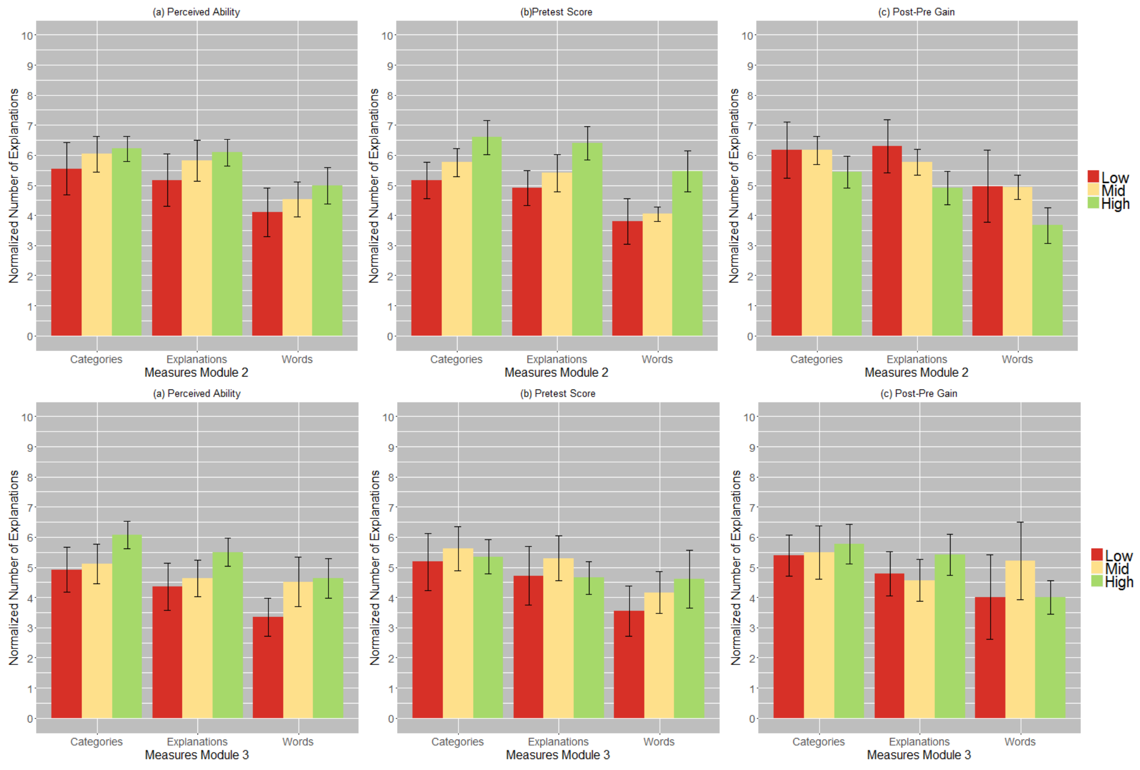


Figure 6.16. Number of categories, explanations, and words in students' explanations for module #2 and #3 based on: (a) perceived ability [Module #2]; (b) pretest performance [Module #2]; (c) gain from posttest to pretest [Module #2]; (d) perceived ability [Module #3]; (e) pretest performance [Module #3]; (f) gain from posttest to pretest [Module #3].

6.4 Summary of the findings

This chapter explored the characteristics of students' written explanations in a black box context. Students were exposed to three computational modules using a Python-based simulation tool to represent thermodynamics phenomena. As part of the homework assignments, students were asked to write in-code comments to explain how the Python code solved the problem. Three elements were explored during the implementation of these instructional approach to elicit students' self-explanations: (1) the affordances of in-code commenting activities for students; (2) the characteristics of students' explanations; and (3) the relationship between students' explanations and student ability to do computer programming.

Students in the black box context described the different ways in which they afforded the written explanations. While writing in-code comments, students commonly reflected on how the sample code worked, how these simulations solved problems in thermodynamics, and even learned about the Python syntax. To a lesser extent, these activities helped students to learn more about thermodynamics and to solve the course project. Nevertheless, nine students would also like to have access to the encapsulating libraries that implement the underlying mechanisms of the simulations.

Students' explanations in this context showed five different types of explainers: declarative, problem-oriented, summarizer, procedural, and limited. Three of these types (i.e., declarative, problem-oriented, and summarizer) mostly used declarative knowledge, with some variations. The problem-oriented explainers made connections to the problem, while the summarizer commented chunks of code with certain purpose. The procedural explainers described how the program would execute, and the limited explainers showed incorrect or limited explanations, and misunderstandings of the code. In general, there was very limited use of schematic knowledge, especially in terms of the rationale for any given section of the code, and almost no instances of monitoring activities.

The characteristics of students' explanations did not show a clear relationship to the students' perceived ability to do computer programming. There was a common pattern between modules two and three, where declarative explainers performed better in the pretest as compared to the other groups of explainers.

However, these result was not statistically significant. Further discussion on these findings is presented in the next chapter.

CHAPTER 7. DISCUSSION AND IMPLICATIONS FOR TEACHING AND LEARNING

Students' ability to explain is an important skill for science and engineering (Lombrozo, 2006). Humans communicate science through explanations that they build based on what they know (Sandoval & Millwood, 2005). Computer programmers usually communicate with collaborators using in-code comments. This study explored students' explanations in the form of in-code comments in the context of computational science and engineering.

Three elements of students' written explanations were studied (see Figure 7.1). We first identified the affordances of in-code commenting activities for students. We defined affordances in this context as the benefits derived by students in writing the in-code comments, and the ways individual students take advantage of them (Gibson, 2014). We then explored the characteristics of students' explanations, organizing them based on the type of knowledge they employed. This analysis allowed us to group students based on the types of explanations they used on different sections of the code. Finally, we compared these characteristics of students' explanations to their ability to do computer programming. The three following sections discuss these three elements under the lens of existing literature in explanations. Last section (8.4) discusses the implications for teaching and learning.

Two different contexts were studied. The first group was the glass box approach, a programming course applied to disciplinary knowledge from materials science and engineering. The second group was the black box approach, in which students enrolled in a thermodynamics of materials course that used computational tools to represent disciplinary phenomena. Students were assigned to write in-code comments within programming worked-examples to explain what the programming

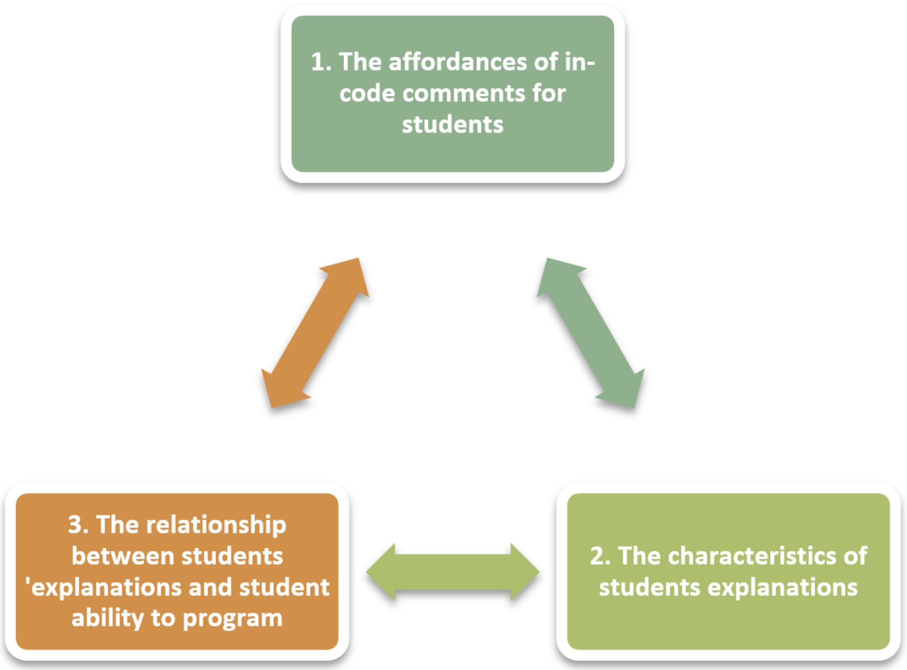

1. The affordances of in-code comments for students
2. The characteristics of students explanations
3. The relationship between students 'explanations and student ability to program

Figure 7.1. Three main elements of students' explanations explored in this study

code was doing. These explanations where qualitatively analyzed and characterized based on the types of knowledge students employed on them.

7.1 Affordances of in-code commenting activities for students

This first part of the study explored the affordances of in-code commenting activities for students in the context of CSE. Self-explaining can be beneficial to learning because students engage in constructive learning activities (Chiu & Chi, 2014). Students use their background knowledge to make inferences about the worked-examples, creating, or adapting their own schemata. Eliciting self-explanations in the context of CSE education had not been previously explored. Moreover, the use of in-code comments as the self-explanation strategy is also an innovative contribution of this study.

Computing education can be approached as a glass box or a black box approach. While the glass box provides students with access to the underlying mechanisms of the computer simulations, it also involves higher cognitive demands. In the following sections we present the affordances of in-code commenting activities for students participating in one of these two contexts.

7.1.1 What is the effect of using in-code commenting activities on students' engagement with the worked-examples in the context of black box and glass box approaches to computational science and engineering?

Using in-code comments as self-explanation strategy engaged students to access the worked-examples more often and more deeply, especially in the glass box approach. Several students in the glass box context submitted the in-code comments for extra-credit and became aware of the existence of the worked-examples. When the activity became graded in 2016 for the first few in-class exercises, a higher percentage of students kept submitting their written explanations during the whole semester.

The black box approach showed that in 2015 several students did the first extra-credit activity, but this rapidly decreased. Students argued that they were already too busy with the homework assignments and that the extra-credit was not completely real, given this was a curved class. In addition, they might not have seen the value of accessing the code, since the graphical user interface already helped them to better understand the THERMO-related concepts. Furthermore, previous studies have suggested that while novices take advantage of worked examples, experts prefer to focus on problem solving activities (Atkinson et al., 2000; Vieira et al., 2015). Hence, a possible explanation for the decrease in the number of submissions is that students were increasingly acquiring schemata that allowed them to face the problems on their own. Overall, although writing in-code comments can engage both the glass box and the black box approaches to study the worked-examples, students in the glass box approach showed a higher level of engagement, given that the purpose of the class was to learn programming skills rather than to represent phenomena.

Another effect of the in-code comments for the black box approach relates to the transparency paradox (Magana et al., 2010). Students in this context were provided with certain level of access to the underlying mechanisms of the simulations. Although the representations of the THERMO phenomena during the modules would not usually require students to access the Python code, students were encouraged to write in-code comments. This process certainly had an effect that represents the paradox. While some students would like to have additional transparency by being able to access the code inside the GIBBS libraries, other students felt under-prepared and under-supported with Python programming skills, necessary to better understand these mechanisms. This result suggests that a higher transparency is important for students, but educators need to consider additional instruction or scaffolding strategies so that students can benefit from this transparency (Magana et al., 2012).

7.1.2 What are affordances of in-code comments self-explanation activities in the contexts of black box and glass box approaches to computational science and engineering?

Self-explaining can be beneficial for learning in several ways (Chiu & Chi, 2014; Williams & Lombrozo, 2010). Self-explaining can integrate new learning materials to existing knowledge, or it can help students to fill the gaps of missing information in the learning materials relying on previous knowledge. This constructive learning activity also engages students in meta-cognitive processes that help to identify and modify misconceptions. Moreover, self-explaining can help students to focus on individual parts of a learning material, scaffolding their learning process.

Students either in the black box or the glass box context identified some of these and other affordances of in-code comments as a self-explanation strategy. The most common affordance in both contexts was the better understanding the worked-examples, while connecting individual lines of code to the overall purpose of the program. Writing comments in the glass box context gave students familiarity with MATLAB syntax, helped them to understand difficult concepts that were taught during the class, and to learn new functions. Going line-by-line writing comments helped students understand how the computer works, to practice algorithm design by seeing experts' solutions to these problems, and even to practice commenting skills.

On the other hand, students in the black box approach to computation were able to identify underlying mathematical equations that model thermodynamic phenomena. Likewise, students engaging in self-explanations were able to better see the connection between thermodynamics and computation, while identifying how they could use these simulation programs for their own projects. The explanation process commonly involves the generalization of the explained phenomenon to common patterns, which allow students to use this knowledge in other contexts (Lombrozo, 2006). In a lesser extent than for the glass box approach, writing comments also helped students in the black box approach to become familiar with the programming language syntax.

A small portion of both groups did not find the in-code comments activities useful. They argued that doing so required additional time that they could better spend working on the projects. One of these students confessed having previous experience in programming, so the activities only helped them at the beginning of the semester to get familiar with the MATLAB syntax. This result was also expected. Previous research has shown that worked-examples are usually useful for novices, who do not have the background knowledge to solve problems on their own from scratch (Sweller et al., 2011). Conversely, more experienced students prefer to learn by problem solving, using their schemata and testing their own ideas.

7.2 Characteristics of Students' Explanations

The second part of this study explored the characteristics of students' written explanations in the two contexts. Students' in-code comments were analyzed using categories within the four types of knowledge for science assessment, and this analysis was used to group students based on their explanation style. The following sections discuss the types of knowledge students used, the types of explainers that were identified, and the common misunderstandings that were identified from students' written explanations.

7.2.1 What are the characteristics of students' explanations in a glass box and a black box approaches to CSE education?

Student use of the four types of knowledge varied among contexts, activities, and sections. The declarative knowledge was the most used one, in part, because the most common type of explanation is to say what an instruction does. The procedural knowledge did not appeared very often in either context, and when did, it was mostly to describe the execution of the program, or the interaction with the user. However, only on very few occasions students explained how an instruction was doing what it did. The schematic knowledge was more often used to connect to background knowledge or to the problem statement (i.e., use of schemata) than to describe the rationale of actions. Likewise, the strategic knowledge was mostly focused on the conditions of applications of actions, and identifying chunks of instructions, especially in the black box context.

Different types of explanations were also identified for different types of sections. For instance, the connections to the problem often emerged when the section involved setting up problem parameters. Meanwhile, the use of background knowledge was employed when the example actually involved some type of disciplinary knowledge, independently of whether the section had direct connection with a problem variable or not (e.g., when setting up supporting variables).

Likewise, the identification of conditions was present when a loop statement or an if-clause were present, but also when the section involved different meanings for its result. Two examples of this case correspond to the penultimate section of activity #5, and the final section of module #1. In the former example, students made connections between the stopping conditions of the program and the resulting output. In the latter example, students talked about the condition of a subtraction between two derivatives, and what was the meaning of a zero or non-zero value (i.e., state function or not). The identification of chunks of instructions, as well as their rationale, and how they relate to the overall goal, comprise important evidence of student understanding of programming code (M. Chi et al., 1989; Lombrozo, 2006; Soloway, 1986). However, students often focus on the most visible aspects of the phenomenon, and therefore, fail to see the interactions among different parts (or chunks) (Watson, Prieto, & Dillon, 1997). This was the case for students' explanations in these two contexts. The explanation of the rationale for a given section was only present in the glass box approach, while the identification of chunks of code with a given purpose was more common in the black box approach. In both contexts, the number of instances for these categories was rather small.

In general, the least identified categories within students' explanations were those who could actually demonstrate students' ability to transfer what they understood from one example to another context. The use of laws and principles has been demonstrated to be an important factor in effective explanation processes (Chiu & Chi, 2014; Kuhn & Katz, 2009), and being able to explain why the program solves a problem represent the third level of programming expertise (Lister et al., 2012). However, the number of explanations including the rationale in the black box context was almost zero, the connection to background knowledge was only present in about 20% of the times for the glass box context, and 10% of the times for the black box context.

On average, students' explanations in the black box approach showed lower quality (Küchemann & Hoyles, 2003; Sandoval & Millwood, 2005) as compared to the glass box approach. First, only the glass box approach showed students using meta-cognitive strategies (MON and OWN). These students may have had a strong background knowledge in programming that enabled them to come up with their own solution (Renkl, 1997). Second, the black box context presented more instances of limited knowledge (INC and LIM), and no instances of rationales for almost any section in the code (WHY). In this context, student use of schematic knowledge was mostly focused on making connections to the problem and to the background principles. In fact, two groups of explainers that connected with the goals and with the problem were identified in the black box context. Students may be more interested on actually thinking about what the program solves and represents, rather than actually understanding how or why it is built in a certain way. Conversely, students in the glass box approach might be looking more into actually understanding the code to a level of being able to build it themselves. Another possible explanation for these differences between approaches might be related to the format of the worked-example. While the glass box approach provided students with a problem to be solved, a strategy on how to solve it, and the MATLAB code, the black box example was introduced by the course instructor and only consisted of the Python code. These instructional designs can affect the way students generate explanations (Schworm & Renkl, 2006). Hence, these students might have not seen the need for understanding what the algorithm design process was like to get to their solution.

These findings need to be taken with caution because usually the explanations we give depend on the audience (Southerland, Abrams, Cummins, & Anzelmo, 2001). These students may have assumed that their audience (i.e. the course instructor) actually understood the lines of code from the example, and therefore, the might have not seen the relevance of rich descriptions. Students may have considered that simple comments and the code would "spoke for themselves", an effect that as has been shown in other contexts (Sandoval & Millwood, 2005). Nevertheless, although this effect might explain the results, especially in the black box context, it is important that students learn to effectively communicate to their audience. Further discussion on the implications for teaching and learning is presented in Section 8.4.

7.2.2 How does the characteristics of students' explanations in a glass box and a black box approaches to CSE education change over time?

Seven types of explainers were identified in these two contexts of CSE education. Students in the glass box approach fell into one out of these five categories: limited, problem-oriented, procedural, schematic, and reasoners. The limited explainers were small groups of students identified in activities #5 (7.69% of the students) and #11 (12.5%), where they included either very few and simple descriptions of the code, or incorrect and limited explanations. The problem-oriented explainers used declarative knowledge to describe what a section of code did, focusing on the most visible sections of the code (Pirolli & Recker, 1994), and not engaging in a reflective process. This group was found in activities #2(16.67%) and #11(18.75%), when students made connections to the problem, which was explicitly related to their background knowledge in mathematics. Conversely, activity #5 showed the largest percentage of procedural explainers (61.54%), those who approach their written explanations making constant connections to how the program will execute, and how the interaction with the user is.

The schematic explainers and the reasoners where only identified in the glass box approach. The schematic explainers would have instances describing the rationale of at least one of the sections in the code of the three activities (58.33%, 23.08%, and 50%). These reasoners usually made connections to laws or principles, which according to the self-explanation effect can help them to better understand the examples (M. Chi et al., 1989; Renkl, 1997). The reasoners comprised a specific group of students that explained their own solution to the problem instead of the provided sample code in activities #2 (20.83%) and #5 (7.69). This type of explainers used their background knowledge to come up with their own solution before validating it with the worked-out solution (Renkl, 1997).

Neither the reasoners nor the schematic explainers were identified into the black box context. However, two additional groups of explainers were identified here: declarative and summarizer explainers. The declarative explainers were identified in all three modules of the THERMO course (37.21%, 30.56%, and 14.29%), and mostly used declarative knowledge to explain all sections of each example. Meanwhile, the summarizer explainers (23.26%, 11.11%, and 45.71%) did not usually write explanations for all sections but used a chunking strategy that allowed them to identify group of instructions or sections and describe the goal of these chunks. Identifying these chunks is an important step towards the development of programming expertise (Lister et al., 2012; Mayer, 1981; Soloway, 1986).

Although it is expected that the overall characteristics of students' explanations as a group change for different activities, the distribution of students among these groups was reasonably similar between activities. For instance, the schematic explainers dominated the glass box context for all activities, but activity #5. The difference of this activity as compared with activities #2 and #11 is that it did not involve any disciplinary or mathematical knowledge in its solution. Moreover, activity #5 was a programming problem, which stopping condition was dependent on the user input. Thus, students focused more in the procedural part of the code, and how it interacted with the user.

Likewise, the groups who mostly used declarative knowledge (declarative, problem-oriented, summarizer, and limited explainers) were the most common ones in the black box approach. The limited explainers where only present in modules #2 and #3, which involved a more complex program. The group of summarizer explainers increased in size in module #3, where the program was bigger, and when students had already explained a simpler version of a similar program in module #2. However, even in this third module, there were not schematic explainers.

Students learning from examples need to connect their background knowledge to make sense of these learning materials (Chiu & Chi, 2014). It is important, especially for novices, that the self-explanation activities elicit these schemata and promote sense-making and meta-cognitive activities from students (Pea et al., 1987; Williams et al., 2010). This was more often the case in the glass box context, where students were interested on actually learning algorithm design, as opposed to THERMO students who might have been more interested in using the computational representation. Two additional factors might be eliciting this effect in the glass box context: the students' level of expertise and the level of transparency of the course. Students in the black box context might not have the necessary background knowledge to make connections to while explaining programming code. The next section discusses the common misunderstandings students showed in their written explanations, especially in the black box context.

7.2.3 What common misunderstandings in programming can be identified from students' explanations in a glass box and a black box approaches to CSE education?

The characterization of students' explanations in both contexts led to several instances of incorrect or limited explanations. Although this was more often the case in the black box approach, students in the CPMSE course also showed some misunderstandings. The higher percentage of instances of limited knowledge for the black box context can be explained by the fact that students did not have specific programming instruction, but instead used programming as a means to understand THERMO-related phenomena.

Table 7.1 describes the common misunderstandings or difficult concepts that were identified in each context. For instance, the concept of objects and the scope of parameters were a common misunderstanding in the black box approach. This is a very limited explanation that can be understood as a misclassification of a formal cause of explanation (Lombrozo, 2006). Students may have incorrectly assumed that the parameter is the feature that describes what this instruction is, omitting the creation of an object that would be later used for a more important purpose. Furthermore, novice programmers in this case not only focused on an individual line of code (Mselle & Twaakyondo, 2012), but on a surface feature of this instruction (Pirolli & Recker, 1994). This misunderstanding demonstrates that students in the black box approach may require additional support to understand the programming code behind the simulations.

Table 7.1.

Misconceptions and difficult concepts in computer programming

Source of misconceptions or difficult concept	**Misconception or Difficult Concept**	**Glass Box**	**Black Box**
Lack of understanding underlying mechanisms	Input-output commands and memory management (Bayman & Mayer, 1983; Goldman et al., 2010; Kaczmarczyk et al., 2010)		
	Non-linear sequence of program (Bayman & Mayer, 1983)	X	
	Equal sign: equation vs. assignment (Bayman & Mayer, 1983)		
	Name of a variable compared to the value in that variable. (Bayman & Mayer, 1983)		X
Superbug (Pea, 1986)	Assuming that the computer understands human language (Pea, 1986)		
	The order in which the instructions are executed (Kaczmarczyk et al., 2010; Pea, 1986)	X	
	Intentionality of the computer program (Pea, 1986)		
	Language-dependent bugs: syntax and semantics (Pea et al., 1987)		
	Lack of meta-cognitive strategies (monitoring learning) (Pea et al., 1987)		X
Systematic errors (Confrey, 1990)	Difficulty to identifying chunks of code with certain purpose (Mselle & Twaakyondo, 2012; Whalley & Lister, 2009)		
	Use of correct knowledge that is incorrectly applied in a broader domain (Fleury, 2000)		X
Difficult and important concepts (relevant to this study)	Objects (Kaczmarczyk et al., 2010)		X
	Loops (Kaczmarczyk et al., 2010)	X	
	Arrays (Taylor et al., 2014)	X	X
	Scope of Parameters (Goldman et al., 2010)		X
	Procedures (Goldman et al., 2010)		X
	Local and Global Variables (Goldman et al., 2010)		

Likewise, the use of loops demonstrated to be a challenging concept for students in both contexts. Students in the CPMSE course were confused regarding the sequence of execution of the instructions inside the loop (Bayman & Mayer, 1983; Kaczmarczyk et al., 2010; Pea, 1986), and their relationship to the condition inside the parenthesis. Meanwhile, students in the THERMO course described the call back function from the module #2 as a loop. Loops are a difficult concept (Goldman et al., 2008, 2010; Kaczmarczyk et al., 2010), and students might be trying to apply their knowledge of loops to an incorrect context. This phenomenon was also previously identified when students used loops to describe a recursive function (Fleury, 2000; Sorva, 2013).

All these misconceptions that were evident from students' explanations of code did not necessarily affected their disciplinary learning in the black box approach. Moreover, we did not identify any explicit misunderstanding related to students' disciplinary knowledge. The examples in THERMO involved a graphical user interface that represented THERMO phenomena and that did not required them to actually write the code themselves; this approach limited students' explanations to little use of schematic and strategic knowledge. However, if the level of transparency is increased, as some students requested, additional support needs to be provided for them to succeed. This support could be in the form of prior programming courses applied to students' disciplinary knowledge (Magana, Falk, Vieira, & Reese, 2016) or multiple forms of scaffolding within the course that allow them to manage the cognitive loads of such complex learning (Magana et al., 2012).

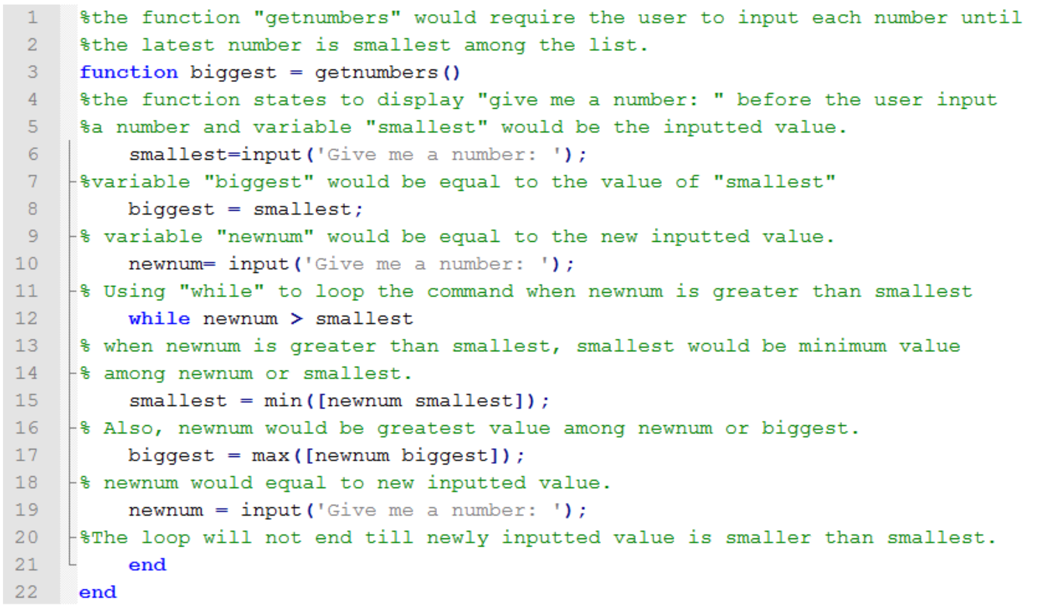
7.3 Relationship between Students' Explanations and Student Ability to Program

The third and final part of the study explored the relationship between the characteristics of students' written explanations, their ability to do computer programming, and their course performance. Survey questions and performance tests were used to compare among groups of explainers.

Students' explanations in the glass box context showed a common pattern when compared to the students' perceived ability to program. Students who started with an average lower perceived ability to do computer programming wrote longer, and more complex explanations than students with an average higher perceived ability. Schematic explainers, those who described the rationale for several sections in the code, had an average lower perceived ability, and an average lower score on the first midterm. An example that compares two of these different approaches to self-explain is presented in Figure 7.2. The first set of explanations (Figure 7.2a) had simple explanations, without rationale for the instructions, but with one monitoring statement in line six. This student (S4) was among the high-perceived ability group, and the high performers in the midterm one. Conversely, the second set of explanations (Figure 7.2b) corresponds to a student (S16) with an initial low perceived ability and a low performance on the first midterm. This student wrote longer explanations, explaining the rationale for each section of the code.

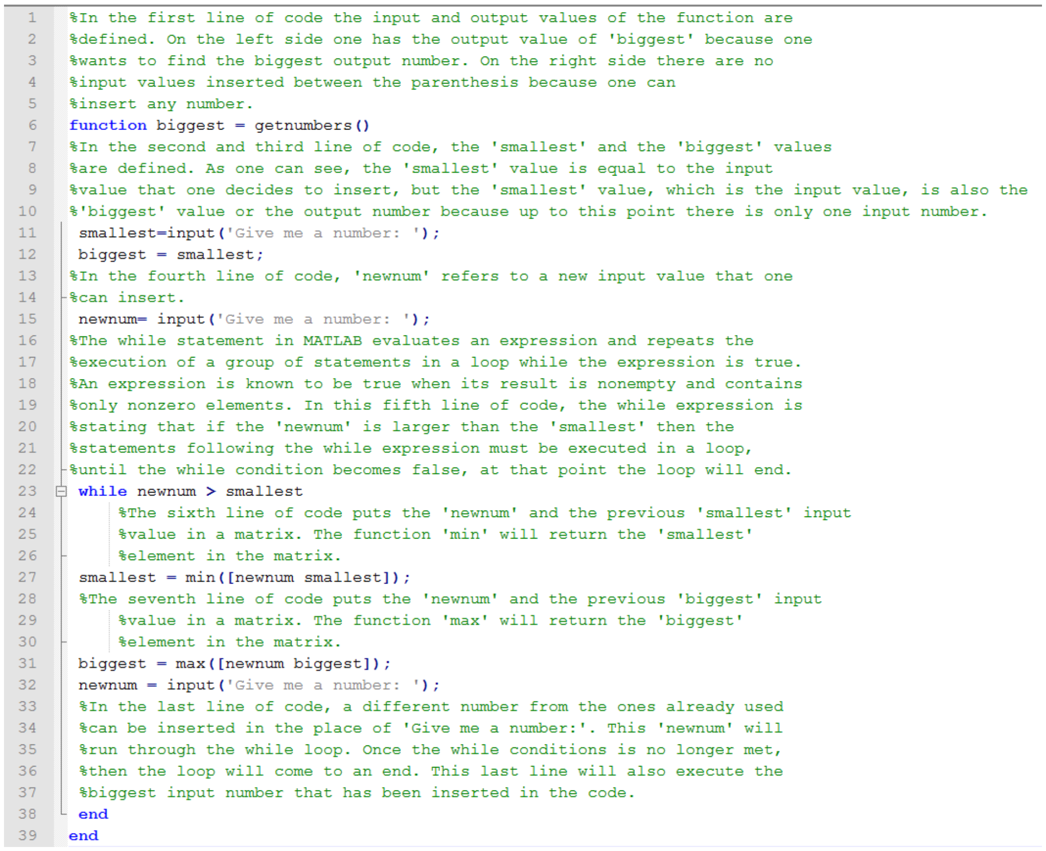
Learners with different skills and background knowledge may benefit differently from the self-explanations (Chiu & Chi, 2014). As described in the affordances of these activities for students, there are different ways in which students may have benefited from the in-code commenting activities. When we compared these affordances to the types of explainers, we also identified that students who used them to learn concepts and not only to familiarize with the syntax, were often schematic explainers. These results suggests that those students in the glass box approach who had limited programming experiences used these explanation activities to engage in a reflective process of understanding of the examples. Conversely, students with high ability to do computer programming wrote simple comments that although depicted their understanding of the example, corresponded to a limited explanation for communicating in CSE. As we mentioned before, the explanations we generate depend on the audience (Southerland et al., 2001). These high ability students may have just decided to complete the task for the course instructor as the audience.

From the perspective of the worked-examples, this result is also expected: being an expert solution to a problem, the worked-examples are usually useful for novices who do not have a relevant background knowledge to allow them to solve the problems on their own; the more experienced students prefer to engage in problem solving rather than in worked-example exploration (Kalyuga et al., 2001; Kester et al., 2010; Renkl, 2005; Sweller et al., 2011).



```
%the function "getnumbers" would require the user to input each number until
%the latest number is smallest among the list.
function biggest = getnumbers()
%the function states to display "give me a number: " before the user input
%a number and variable "smallest" would be the inputted value.
    smallest=input('Give me a number: ');
%variable "biggest" would be equal to the value of "smallest"
    biggest = smallest;
% variable "newnum" would be equal to the new inputted value.
    newnum= input('Give me a number: ');
% Using "while" to loop the command when newnum is greater than smallest
    while newnum > smallest
% when newnum is greater than smallest, smallest would be minimum value
% among newnum or smallest.
    smallest = min([newnum smallest]);
% Also, newnum would be greatest value among newnum or biggest.
    biggest = max([newnum biggest]);
% newnum would equal to new inputted value.
    newnum = input('Give me a number: ');
%The loop will not end till newly inputted value is smaller than smallest.
    end
end
```

(a) High Perceived Ability



```
%In the first line of code the input and output values of the function are
%defined. On the left side one has the output value of 'biggest' because one
%wants to find the biggest output number. On the right side there are no
%input values inserted between the parenthesis because one can
%insert any number.
function biggest = getnumbers()
%In the second and third line of code, the 'smallest' and the 'biggest' values
%are defined. As one can see, the 'smallest' value is equal to the input
%value that one decides to insert, but the 'smallest' value, which is the input value, is also the
%'biggest' value or the output number because up to this point there is only one input number.
 smallest=input('Give me a number: ');
 biggest = smallest;
%In the fourth line of code, 'newnum' refers to a new input value that one
%can insert.
 newnum= input('Give me a number: ');
%The while statement in MATLAB evaluates an expression and repeats the
%execution of a group of statements in a loop while the expression is true.
%An expression is known to be true when its result is nonempty and contains
%only nonzero elements. In this fifth line of code, the while expression is
%stating that if the 'newnum' is larger than the 'smallest' then the
%statements following the while expression must be executed in a loop,
%until the while condition becomes false, at that point the loop will end.
 while newnum > smallest
     %The sixth line of code puts the 'newnum' and the previous 'smallest' input
     %value in a matrix. The function 'min' will return the 'smallest'
     %element in the matrix.
 smallest = min([newnum smallest]);
 %The seventh line of code puts the 'newnum' and the previous 'biggest' input
     %value in a matrix. The function 'max' will return the 'biggest'
     %element in the matrix.
 biggest = max([newnum biggest]);
 newnum = input('Give me a number: ');
 %In the last line of code, a different number from the ones already used
 %can be inserted in the place of 'Give me a number:'. This 'newnum' will
 %run through the while loop. Once the while conditions is no longer met,
 %then the loop will come to an end. This last line will also execute the
 %biggest input number that has been inserted in the code.
 end
end
```

(b) Low Perceived Ability

Figure 7.2. Students' explanations for activity #5 (a) high perceived ability, high performer in midterm one; (b) low perceived ability, low performers in midterm two

On the other hand, students' explanations in the black box context did not show a clear relationship to students' ability to do computer programming. The different types of explainers had a similar average perceived ability to do computer programming at the beginning of the semester. A possible explanation for this effect can be given by the format in which these computational modules were introduced. Students had limited access to the underlying mechanisms, and they did not have any explicit instruction on Python programming as part of the thermodynamics course. These led into limited explanations, where students incorrectly described some of the sections, and they did not explain the rationale for any given section. Furthermore, the worked-examples in this context were introduced by the course instruction, and when instructional explanations are present, the self-explanation effect is diminished (Schworm & Renkl, 2006).

7.4 Implications for Teaching and Learning

7.4.1 Learning

Using in-code comments as a self-explanation strategy demonstrated to be an effective approach to engage students in the active exploration of worked-examples in CSE. Students afforded these explanation activities in different ways. Students in the glass box context dissected the worked-examples, learned about effective algorithm design techniques, got familiar with the MATLAB syntax, and practiced their commenting abilities. In this context, the use of graded activities at the beginning of the semester allowed students to identify how they could benefit from them, and decide whether it would be useful for them to continue submitting their explanations for extra-credit.

In the black box context, students also understood how each line of the code was related to the overall goal of the program, and got familiar with the Python syntax. Furthermore, students in this context were able to see the connection between the computational modules and the disciplinary problems. However, although students would like to have additional transparency on the encapsulated functions, some of them also advocated for additional instruction in Python programming to be able to take advantage of the activities. This result suggests that while a higher transparency is important for students, this transparency cannot be given without proper instruction or scaffolding strategies.

From the learning perspective, it is important that students, especially novice programmers, reflect and use meta-cognitive strategies while studying the examples. Students in the black box approach did not explain the rationale for how the code was designed. The learning outcomes for this context focused on student understanding of disciplinary phenomena, and therefore, the design of the course did not involve explicit instruction in programming. It is important then to provide further instruction, prompting, or scaffolding, to engage students in higher quality explanations of the programming code. Explicit instruction on what these explanations should comprise, might have an effect both in student explanations, and in student learning (Renkl, 2005).

Besides the use of in-code commenting activities for the benefit of student learning, these comments can also be analyzed to identify students' ability to communicate through their code, as well as students' misunderstandings of programming concepts. This study showed that different students, activities, sections, and contexts, produce different explanations.

In the two contexts we explored, students' explanations relied mostly in declarative knowledge. These simple statements did not effectively communicate how the code solves a problem or why it was built in certain way, but only what it did. From the instructional perspective, we would like to have students writing comprehensive in-code comments that would enable them to collaborate in CSE projects. Also, as a communication strategy, we would expect that high quality students' explanations go beyond describing what an individual line of code does, to say: what the goals for certain chunks of code are, the conditions under which these chunks work, and the rationale for writing these instructions. All this, while making connections to appropriate principles or laws from their background knowledge. As a consequence, one implication of this study lies on the importance of educating students in how to create these explanations that can be effectively used as a communication strategy. These effective characteristics do not only need to consider the types of knowledge students used, but also the length of these explanations. Lengthy comments within a programming code negatively impacts readability of the code, and so students usually prefer simple but informative comments (Vieira et al., 2015).

7.4.2 Teaching

Another implication of this study is that instructors can potentially use this mechanism of in-code comments as an assessment strategy. First, we were able to identify students' misunderstandings with programming from their written explanations. These activities can also potentially be used with incorrect, incomplete, or inefficient examples where students could evidence their monitoring strategies to identify these mistakes. Seven students in the glass box approach included monitoring statements for an instruction that was not incorrect but was unnecessary. Providing incorrect examples could promote students' ability to monitor and revise their own understanding of the program, the code, or the phenomenon (Chiu & Chi, 2014).

Also, different types of sections led to different types of knowledge (e.g. setting up problem parameters showed more connections to the problem). An instructor can just look specific sections to focus the assessment process on certain types of knowledge. For example, if an instructor in the glass box approach is interested in assessing declarative knowledge, they might choose to look at students' comments in the function definition type of section. On the other hand, if the instructor would like to elicit schematic and strategic knowledge, looking at sections of the code where there are more connections to the problem and background principles would be more appropriate.

7.4.3 Instructional Principles of Worked-Examples in CSE

Building upon Atkinson et al. (2000), we adapted the instructional principles of worked-examples for the specific contexts that we explored. The two cases we described in this document (i.e. Chapter 5 and Chapter 6) and the pilot study (Vieira et al., 2015) produced empirical results that can support the implementation of programming worked-examples. For instance, having multiple representations of the worked-examples helped students in the glass box context. While some students watched the video explanations more than once for certain exercises, other students preferred the textual and mathematical representations. In this regard, one of the representations we used in (Vieira et al., 2015) was the in-code comments. When we provided these comments, students often suggested us to reduce their length, because lengthy comments reduced the readability of the code.

This representation (i.e., in-code comments) was also used to elicit students' explanations of the worked-examples. While students did not access the worked-examples in the glass box context in 2014, the implementation of this self-explanation strategy started to make them aware of the different benefits this could have. Furthermore, it was important to have the first few explanation activities graded in 2016 so that all students could at least see whether this process would help them or not. After these graded activities were finished, giving students the possibility of submitting additional explanations kept half of the group engaged in the active exploration of the worked-examples. These were especially useful for novices in the glass box context, who reflected on their understanding of the worked-examples and made sense out of them.

The use of in-code commenting activities in the black box context was also an effective approach to have students actively explore the Python code behind the visual simulation. Students benefited from these activities by relating individual lines of code to the overall functionality, seeing the connection between computation and thermodynamics, and getting familiar with the Python syntax. In this context, it was also important to have the activities individually graded so that students could see the benefit of completing them. However, students in this context may have needed additional support to deal with the additional complexity that having access to the code might provide. Table 8-2 summarizes the instructional principles with our additions for the specific context.

Table 7.2.

Design characteristics for effective worked examples

Feature	Description
Intra-Example	• The easy mapping guideline: Students take advantage of different forms of representation of a worked-out example (e.g., videos, diagrams, mathematical model, and programming code). However, different formats should be fully integrated to avoid extra cognitive load generated by the split attention effect (Renkl, 2005). Furthermore, when in-code comments are used as a representation to explain the example, these should be simple and comprehensive to avoid affecting the readability of the code. • The meaningful building-blocks guideline (Renkl, 2005): The example should be divided in sub goals or steps to make it easier for the student to understand. Labels and visual separation of steps can be used for this purpose. One approach for this guideline can be the use of the problem-solving steps, as we did in the glass box context.
Inter-Example	• The structure-emphasizing guideline (Renkl, 2005): The use of multiple worked examples (at least two of them) with structural differences can improve the learning experience. The worked examples should be presented with similar problem statements that encourage the students to build schemas based on analogies and the identification of declarative and procedural rules. • The programming worked-examples help students to learn about algorithm design, the syntax of the programming language, and the connection to their discipline. Worked-examples in these context should provide these meaningful elements to be valuable for students: provide useful strategies of algorithm design to solve similar problems; make connections between the sample code and relevant disciplinary problems.
Environ-mental	• Self-explanation effect (M. Chi et al., 1989): Students should be encouraged to self-explain the worked examples in order to be actively engaged with them. Writing in-code comments is an effective strategy to engage students in the active study of worked-examples, and brings several benefits to the learning process. • The fading effect of the self-explanation strategy can promote students' use of this scaffolding strategy on demand. For instance, providing credit for the first few activities and leaving the rest of them for extra-credit has demonstrated to keep at least half of the students engaged in this process throughout the semester. • Students with limited instruction on programming concepts may need extra-support to be able to create meaningful explanations.

CHAPTER 8. CONCLUSIONS AND FUTURE WORK

The use of in-code comments as a self-explanation strategy was explored in the context of two approaches for CSE education. The glass box approach involved a programming course for materials science and engineering students, while the black box approach was implemented in a thermodynamics course using computational tools. The in-code commenting activities increased the students awareness of worked-examples, and the active exploration of these materials.

Students found that these commenting activities were useful to better understand the examples, to connect the programming code to the disciplinary problems and mathematical models, to practice algorithm design and MATLAB® syntax, and to improve their commenting skills. Students in the black box approach would like to have higher transparency of the underlying mechanisms, but they would also like to have additional support to be able to take advantage of these activities. Hence, future research should explore what other forms of support can be provided in a black box context, where the learning outcomes regarding the computational tools is not at a level of creating but at a level of applying. Although students would like to have more transparency, the main learning outcomes in this context are usually the ones related to the disciplinary knowledge (i.e., thermodynamics of materials), and not to the computational component.

Students explanations were characterized in both contexts by the abundant use of declarative knowledge, and the limited use of schematic and strategic knowledge. While some of the students in the glass box context explained the rationale for the instructions and used meta-cognitive strategies, this was not the case in the black box context. Looking at students explanations as their ability to communicate programming code, these written comments were very limited. However, this could be the case because students may have felt that they were writing comments for the course instructor, and he already knew the examples. Thus, a future study will compare students explanations of their own code to the ones for the worked-examples. Furthermore, having extensive written explanations is a detriment to the readability of the code. Hence, we would like to explore how expert programmers create their explanations in a similar context, and what are the best practices that need to be taught to the students to better communicate through written comments.

This study also showed that different students created explanations differently, and that at least in the glass box context, students prior knowledge was an important factor to determine the way they created explanations. Students with low perceived ability to do computer programming wrote explanations that involved the use of schematic knowledge more often that students with high perceived ability. Future research should also focus on understanding how certain type of explanations affects student understanding of the learning materials. In other contexts, the elements students use in their explanations have demonstrated to affect how much they understand the materials (Chiu & Chi, 2014). Hence, identifying the characteristics of effective explanations for learning could potentially benefit student learning. Once detected, these strategies can be taught and promoted among students to further increase the effectiveness of their study of worked-examples (Sweller et al., 2011).

These criteria for effective explanations should also consider how the use of programming or disciplinary background knowledge in the explanations affects the learning process. The principles in the context of Computational Science and Engineering, especially for the black box approach can have two different sources: the programming principles and the disciplinary principles. While some students may fail to identify the programming principles, they might still get the disciplinary understanding, and vice versa.

Identifying these effective self-explanations strategies can be also used for intelligent tutoring systems. Several semantic analysis techniques have been explored to provide automatic feedback to students when they provide limited explanations (Aleven & Koedinger, 2002; Makatchev et al., 2004). The large number of existing massive open online courses (MOOC) related to computer programming could benefit from this instructional strategy, both as a learning material and as automatic assessment.

Some of the relevant research questions that can be further explored after this study are:

- What are the characteristics of students explanations that lead to a better understanding of worked-examples in different contexts of CSE education?
- How do expert computer programmers write in-code comments to communicate with other people working on the same code?
- What instructional strategies can be used to promote students ability to communicate through written in-code comments?
- What are the characteristics of students explanations of their own computer programs?
- What are effective instructional strategies that can support student learning of transparent computational tools in a black box context?
- How can we promote the use of schematic and strategic knowledge in the black box context to CSE education?
- How can the in-code commenting activities be integrated into programming massive open online courses?

APPENDICES

CHAPTER A. SAMPLE WORKED EXAMPLE CPMSE - EXAMPLE 11 - ATOMIC BONDS

A.1 Problem Statement

Write a function called atomicbonds that determines which atoms from a list are closer than a distance called cutoff. The function should accept as input a matrix of atomic positions (x, y, z) where each row represents a different atom, in a N by 3 matrix where N is the number of atoms.

The output should be a sparse connectivity matrix similar to the one discussed in the podcast, where the row and column represent atom numbers and the value for each pair is set to the separation distance between the atoms if the two are within the cutoff or 0 if they are not. To avoid redundancy only the upper triangular part of the matrix should be non-zero (those values where row ¡ column).

A.1.1 For a video explanation of this example see:

Part 1 https://www.youtube.com/watch?v=AlLGsGgUCPQ

Part 2 https://www.youtube.com/watch?v=ENgHexFqoc4

See the correct version of the code at the end of this document

A.2 What is the problem asking us to do?

Suppose we have a matrix with the position of a set of atoms. Each row in the matrix corresponds to one atom. The columns are the coordinates x, y, z of a given position of the atom.

If you have large tables or figures to include and need to use margin space, use the right margin and bottom margin.

The problem asks us to write a program to identify the pairs of atoms closer than a given cutoff distance.

The result should be another matrix relating each atom to the rest of them.

A.3 Addressing the Problem

How should we do it?

Let's take the matrix again. It needs to be transformed as follows:

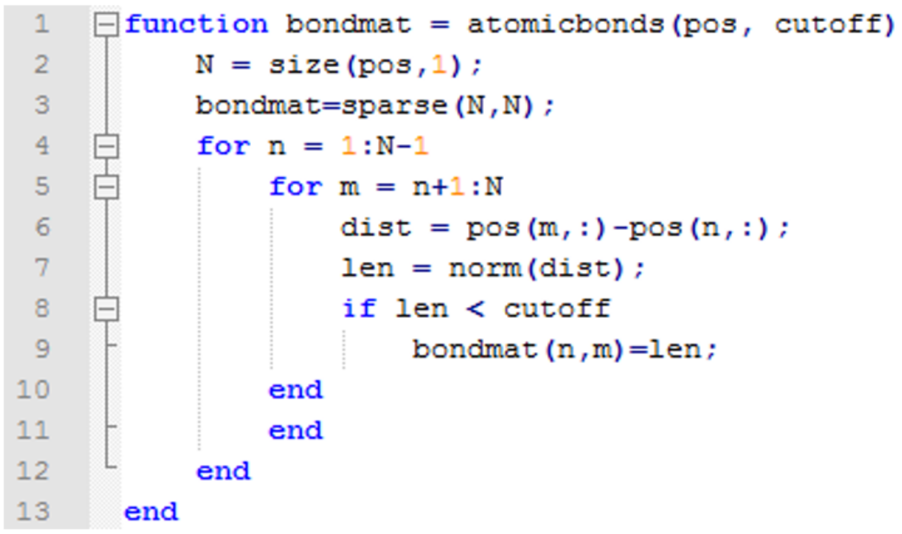
$$\begin{array}{c} 1\\ .\\ .\\ .\\ n \end{array}\left[\begin{array}{cccc} \quad & \quad & \quad & \quad \\ & & & \\ & & & \\ & & & \\ & & & \end{array}\right] \Rightarrow \begin{array}{cc} & \begin{array}{cccccc} 1 & 2 & \ldots & & n\text{-}1 & n \end{array} \\ \begin{array}{c} 1\\ 2\\ \ldots\\ n\text{-}1\\ n \end{array} & \left[\begin{array}{cccccc} n/a & 0 & \ldots & d1 & 0 & d2 \\ . & n/a & \ldots & 0 & d3 & 0 \\ . & . & \ldots & n/a & 0 & 0 \\ . & . & .\ldots & . & n/a & 0 \\ d2 & . & \ldots & . & . & n/a \end{array}\right] \end{array}$$

Please notice the following elements on the result:

- The diagonal is not important because we only want to identify distances between atoms.
- Only one of the triangular parts of the matrix will be important. Otherwise, the result will be redundant. In this case, we care for the upper part.
- The smaller is the cut off, the more zeros the matrix has. Using a matrix to store this data would be very inefficient. More than half of the matrix consists of irrelevant values.

Therefore, this is more efficiently carried out by having a sparse matrix which only stores the non-zero values.

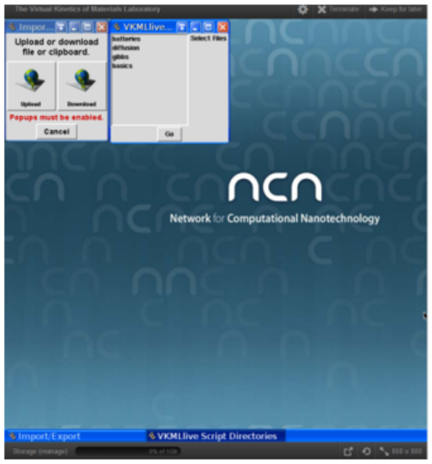
MATLAB Code

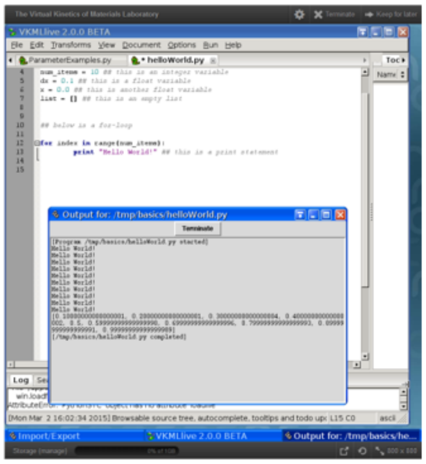


```
function bondmat = atomicbonds(pos, cutoff)
    N = size(pos,1);
    bondmat=sparse(N,N);
    for n = 1:N-1
        for m = n+1:N
            dist = pos(m,:)-pos(n,:);
            len = norm(dist);
            if len < cutoff
                bondmat(n,m)=len;
        end
        end
    end
end
```

CHAPTER B. SAMPLE MODULE IN THERMODYNAMICS

The lab session lasts for 50 minutes in total. The learning objective of this first computational module is to enable students to create and run a new program using Python, as well as to solve simple state function problems using the computational approach. The tool used during the class is the Virtual Kinetics of Materials Laboratory (VKML), hosted in Nanohub (https://nanohub.org/tools/vkmllive).


The Virtual Kinetics of Materials Laboratory
Import...
VKMLlive...
Upload or download file or clipboard.
Upload
Download
Popups must be enabled.
Cancel
batteries
diffusion
gibbs
basics
Select Files
Go
Network for Computational Nanotechnology
Import/Export
VKMLlive Script Directories
Storage (manage)


The Virtual Kinetics of Materials Laboratory
Terminate
Keep for later
VKMLlive 2.0.0 BETA
File Edit Transforms View Document Options Run Help
ParameterExamples.py
* helloWorld.py
Output for: /tmp/basics/helloWorld.py
Terminate
Log
[Mon Mar 2 16:02:34 2015] Browsable source tree, autocomplete, tooltips and todo up: L15 C0
Import/Export
VKMLlive 2.0.0 BETA
Output for: /tmp/basics/he...
Storage (manage)

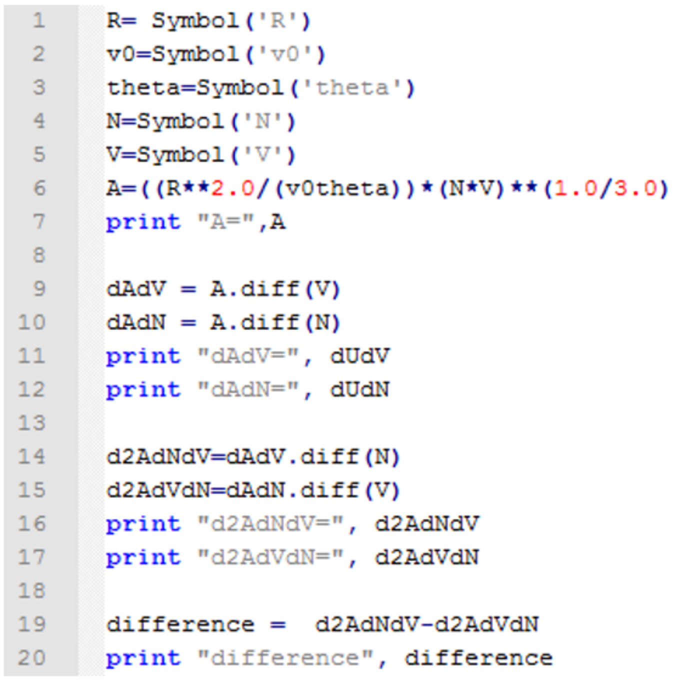
```
num_items = 10 ## this is an integer variable
dx = 0.1 ## this is a float variable
x = 0.0 ## this is another float variable
list = [] ## this is an empty list


## below is a for-loop

for index in range(num_items):
        print "Hello World!" ## this is a print statement
```

```
[Program /tmp/basics/helloWorld.py started]
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
[0.10000000000000001, 0.20000000000000001, 0.30000000000000004, 0.40000000000000
002, 0.5, 0.59999999999999998, 0.69999999999999996, 0.79999999999999993, 0.89999
999999999991, 0.99999999999999989]
[/tmp/basics/helloWorld.py completed]
```

The lab session starts with a ten-minute pretest. The test consists of a single question asking whether a provided function was a state function or not. Students are allowed to use any method they considered appropriate to find the correct response. A correct answer of the test would give the students five extra-points on the weekly homework.

The professor lectures the students on simple actions such as: create a new file; declare variables; write a hello world program; write a for loop; execute a program; import Python packages; and write comments within the code. The professor also highlights the importance of commenting the code to be able to collaborate and better understand a program.

The next part of the activity is intended to identify whether a given equation was a state function or not using VKML. First, the group discussed the activity conceptually and mathematically. Then, the professor starts to guide the implementation of the program. The class may end before the professor finish activity. Therefore, the example is published on blackboard. The student assignments for the following week include commenting the code to describe each steps, and modify the program so that is evaluates a different function. Both assignments are optional and provided extra credit in the homework. The code that was provided to the student was:



```
R= Symbol('R')
v0=Symbol('v0')
theta=Symbol('theta')
N=Symbol('N')
V=Symbol('V')
A=((R**2.0/(v0theta))*(N*V)**(1.0/3.0)
print "A=",A

dAdV = A.diff(V)
dAdN = A.diff(N)
print "dAdV=", dUdV
print "dAdN=", dUdN

d2AdNdV=dAdV.diff(N)
d2AdVdN=dAdN.diff(V)
print "d2AdNdV=", d2AdNdV
print "d2AdVdN=", d2AdVdN

difference =  d2AdNdV-d2AdVdN
print "difference", difference
```

CHAPTER C. SAMPLE QUOTES FROM STUDENTS' EXPLANATIONS

Activity #2

ToK	Code	Type of explanation
LK	SIM	*N/A*
	INC	*%Set the value of variable 'm','m' is the* ***a variable in the quadratic formula*** *%Set the value of variable 'n','n' is the* ***a variable in the quadratic formula*** *%Set the value of variable 'p','p' is the* ***a variable in the quadratic formula***
	LIM	*N/A*
	PHR	*The variable "a" is assigned the value ...sqrt(b1^2 + c1^2 - 2*b1*c1*cos(pi*A1/180))*
CK	COA	*Define function Cosinelaw*
	VAR	*'a' is the length of diagonal*
	PAR	*There are 5* ***input parameters b1, b2 c1, A1, & A2, and 2 outputs, c2a & c2b.***
	DAT	*Describe the* ***data type*** *of the variable*
	COD	*N/A*
PK	HOW	*This function is* ***using law of cosines*** *to compute the length of side c2 four sided figure has missing side c2, law of cosines will be used to determine this side.*
	EXE	*This line of code will be hidden from the command window.*
SK	PRO	*This is for solving the law of cosines when* ***we have two triangles with shared side a****. top triangle is b1, c1, a. bottom is b2, c2,a angles opposite sides are capital liters (a1, b1, c1, a2...)*
	GOA	*Generate a function that can calculate the length of side c2*
	BGK	*Two answers are possible since the quadratic equation comes into play*
	WHY	*The cosine of a1* ***is written as cos(pi*a1/180) because*** *we are given the angle in degrees and we need to convert it into radians.*
	RAG	*In this line we utilize the law of cosines* ***to*** *define the variable of the common side 'a' of the top and bottom triangle utilizing some of our input variables*
	INS	*This code is setting a1 to be equivalent* ***to 120 which is in degrees***
TK	CON	*Sets value m to 1 for use in quadratic formula* ***(m will always be 1 in this scenario)***
	CHK	***In the last three lines of code*** *we again utilize a law of cosines in addition to the quadratic formula to define variables we can use to calculate our two output variables*
	MON	*N/A*
	OWN	*N/A*

Activity #5

ToK	Code	Type of explanation
LK	SIM	*The while statement in matlab evaluates an expression and repeats the execution of a group of statements in a loop while the expression is true ➔ Copied from MATHWORKS*
	INC	*This part of loop will not run if newnum<smallest from the first line of the body of the loop*
	LIM	*N/A*
	PHR	*N/A*
CK	COA	*the smallest number is determined*
	VAR	*biggest, which is the largest inputted number.*
	PAR	*Outputs biggest, no inputs in call*
	DAT	*Of the 2-number array consisting of the values of "newnum" and "smallest," "smallest"*
	COD	*In the first line of code the input and output values of the function are defined*
PK	HOW	*This line places the new number and the previous smallest number in an array together and utilizes the min function to output the lowest number of the two value*
	EXE	*The function states to display "give me a number: " before the user input a number*
SK	PRO	*The function will end because it has already had a smaller value input than all numbers so far and it will output the biggest.*
	GOA	*Create a function that keeps asking numbers until it's less than all the numbers so far, then output the biggest number.*
	BGK	*Creates a loop structure that will repeat as long as newnum is greater than smallest after every iteration of the contents of the loop*
	WHY	*On the right we leave our inputs empty **because** the user of the code will input their own values.*
	RAG	*We use the input function that says 'Give me a number' again so that the user will put in another number to compare with the first number.*
	INS	*So if the previous biggest was 4 and the newnum is 5, 5 will now be the value for biggest*
TK	CON	*While the most recently entered number is larger than the smallest, this loops continues*
	CHK	*Here we set up a while loop **to know when to stop asking** the user for more numbers.*
	MON	*This line is unecessary i think*
	OWN	*N/A*

Activity #11

ToK	Code	Type of explanation
LK	SIM	*Defines a function bondmat that accepts the input of an atomic positons matrix(x,y,z) where each row represents a different atom, in a n by 3 matrix where n is the number of atoms*
	INC	*The value of variable n is equal to a row matrix of size of the position matrix*
	LIM	*This function will find the distance between each different atom*
	PHR	*The variable N is the size of a pos by 1 column matrix.*
CK	COA	*get the total number of atoms*
	VAR	*Parameters pos and cuttoff, the number of atoms and the cutoff distance,*
	PAR	*Takes in matrix with atom positions and cutoff number %takes in matrix with atom positions and cutoff number*
	DAT	*Assign bondmat to an nxn sparse matrix.*
	COD	*The first 'for' loop states that for n=1:n-1*
PK	HOW	*determines how many atoms there are by asking the size of the matrix*
	EXE	*N/A*
SK	PRO	*An input to the cutoff seperation between two atoms if two atoms are seperated by a distance greater then the cutoff then a zero will be input in the sparse matrix*
	GOA	*We create a function called "atomicbonds" that returns a matrix that shows the distance between two atoms if the distance is less than the cutoff value*
	BGK	*The norm function will square each individual value of the distance vector and take the square root*
	WHY	*Since the previous line only determined the difference between each coordinate vector (x,y,z), this line finds the magnitude of the distance.*
	RAG	*Essentially we are finding the number of atoms involved so we can set up our sparse matrix.*
	INS	*E.G. ((X,Y,Z) ==> SQRT(X^2 + Y^2 + Z^2)*
TK	CON	*we let n be the row number from 1 to n-1, and m to be from n+1 to n*
	CHK	*We then use nested for loops to iterate through …*
	MON	*N/A*
	OWN	*N/A*

Module #1

ToK	Code	Type of explanation
LK	SIM	*N/A*
	INC	*Import file*
	LIM	*Imports symbols*
	PHR	*define u as R*(NV)^2*
CK	COA	*import library*
	VAR	*Defines the character R which represents the gas constant*
	PAR	*Differentiates the function u with respect to v*
	DAT	*Defines t as a symbol*
	COD	*The * command symbolizes multiplication **(2) represents raising the quantity (n * v) to the power of 2*
PK	HOW	*Using same sympy functions to take cross derivatives of dudn and .dudv.*
	EXE	*When the program is run, the equation for u will be shown on the screen*
SK	PRO	*Defines symbolic variables that represent each thermodynamic variable used in the derivatives necessary to calculate maxwell relations and prove state functions*
	GOA	*This is the function we are verifying is a state function*
	BGK	*Define letters as variables, keeping consistent with the conventions used in class.*
	WHY	*if the difference equals 0, the function is a state function because the second derivatives are equal*
	RAG	*Take cross derivatives of dudn and .dudv. to check is maxwells equations hold*
	INS	*Importing library sympy to be able to perform calculations needed and use various commands such as u.diff*
TK	CON	*Subtracts the two, should be 0 if state fn*
	CHK	*Defines the symbols to be used*
	MON	*takes the first derivative with respect to the variable in parentheses and more can be added for as many variables as necessary*
	OWN	*N/A*

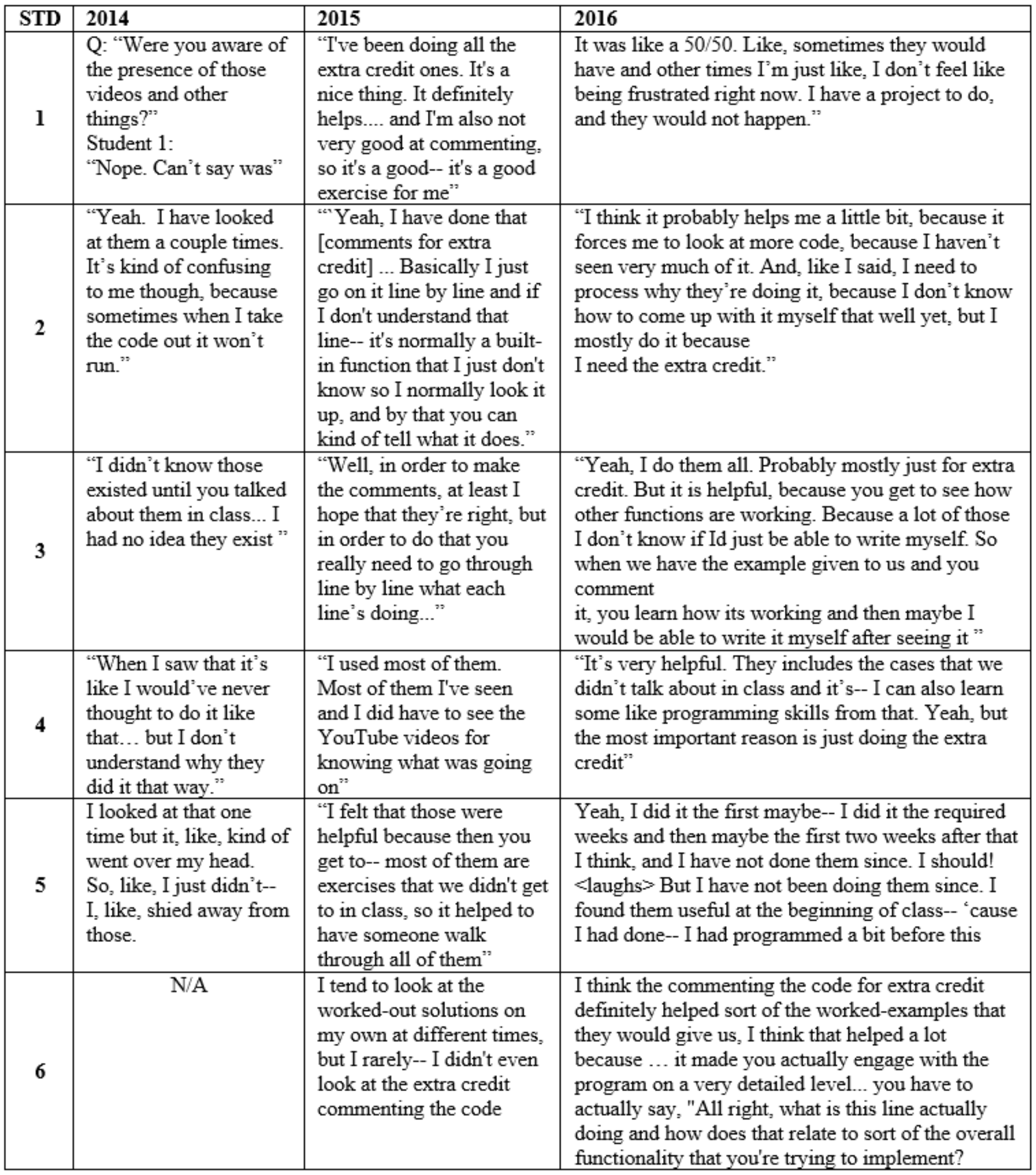
Module #2

ToK	Code	Type of explanation
LK	SIM	*N/A*
	INC	*Import calculation method from database*
	LIM	*Creates title for GUI*
	PHR	*if the name is main then ...*
CK	COA	*import a function*
	VAR	*Solver for chosen variables*
	PAR	*Define gnuplotv as the module with title 'temperature'*
	DAT	*N/A*
	COD	*Begins if statement*
PK	HOW	*the callback is how inputs are read by the computer. when the user adjusts the sliders (Temperature or omega) or input text into the text boxes (path, format, and title) the callback function takes this information and updates the viewer.*
	EXE	*Creates a input for path which the user can enter values into*
SK	PRO	*Imports the general variables for the gibbs free energy equation for phase diagram thermodynamics*
	GOA	*Takes value changed made by the user back into the solver and changed the plot accordinly*
	BGK	*Import regular free energy variable from gibbs library and store with handle rfe*
	WHY	*N/A*
	RAG	*Makes sliders to adjust omega and temperature between the given ranges in order to give flexibility and test many different situations*
	INS	*Setting the initial value as 0 (for question 6, this value set to 22 800)*
TK	CON	*Creates a dial for omega which the user can manipulate from -30000 to 30000*
	CHK	*This section creates the gui and allows the user to control the parameters controlling the equations*
	MON	*N/A*
	OWN	*N/A*

Module #3

ToK	Code	Type of explanation
LK	SIM	*N/A*
	INC	*Finds omega one and two*
	LIM	*names a viewer*
	PHR	*Creates a variable DGl that has 5 variables*
CK	COA	*imports graphic user interphase*
	VAR	*The constrains and default value of of temperature*
	PAR	*This defines the free enerqy of the liquid phase based on omega 2, delta ha and hb, tma and tmb*
	DAT	*This creates the matrix "variables" to define the plot variables*
	COD	*Solves for phase diagram from given hardcoded inputs*
PK	HOW	*Solves for the phase diagram using the Gibbs energy curve equations for the two phases*
	EXE	*The next lines of code, 65-73, create the slider menu that the user interact*
SK	PRO	*Starts to define the energy diagram*
	GOA	*Displays phase diagram plot*
	BGK	*Model construction for drawing one gibbs mixing free energy function*
	WHY	*Defining two different initial conditions because this is a two component system.*
	RAG	*Inputs DGl and DG2 into a Phase Diagram solver in order to create values for the Phase Diagram.*
	INS	*Peritectic phase diagram created using omega 1= 15000 omega 2 = 16765 ·*
TK	CON	*Create varibable min_y in p belong to ml with range from 0 to 4000, initial as 300*
	CHK	*Solves equations & updates gui*
	MON	*N/A*
	OWN	*N/A*

CHAPTER D. APPENDIX D STUDENTS' RESPONSES TO INTERVIEW QUESTIONS GLASS BOX - CPMSE



STD	2014	2015	2016
1	Q: "Were you aware of the presence of those videos and other things?" Student 1: "Nope. Can't say was"	"I've been doing all the extra credit ones. It's a nice thing. It definitely helps.... and I'm also not very good at commenting, so it's a good-- it's a good exercise for me"	It was like a 50/50. Like, sometimes they would have and other times I'm just like, I don't feel like being frustrated right now. I have a project to do, and they would not happen."
2	"Yeah. I have looked at them a couple times. It's kind of confusing to me though, because sometimes when I take the code out it won't run."	"'Yeah, I have done that [comments for extra credit] ... Basically I just go on it line by line and if I don't understand that line-- it's normally a built-in function that I just don't know so I normally look it up, and by that you can kind of tell what it does."	"I think it probably helps me a little bit, because it forces me to look at more code, because I haven't seen very much of it. And, like I said, I need to process why they're doing it, because I don't know how to come up with it myself that well yet, but I mostly do it because I need the extra credit."
3	"I didn't know those existed until you talked about them in class... I had no idea they exist "	"Well, in order to make the comments, at least I hope that they're right, but in order to do that you really need to go through line by line what each line's doing..."	"Yeah, I do them all. Probably mostly just for extra credit. But it is helpful, because you get to see how other functions are working. Because a lot of those I don't know if Id just be able to write myself. So when we have the example given to us and you comment it, you learn how its working and then maybe I would be able to write it myself after seeing it "
4	"When I saw that it's like I would've never thought to do it like that… but I don't understand why they did it that way."	"I used most of them. Most of them I've seen and I did have to see the YouTube videos for knowing what was going on"	"It's very helpful. They includes the cases that we didn't talk about in class and it's-- I can also learn some like programming skills from that. Yeah, but the most important reason is just doing the extra credit"
5	I looked at that one time but it, like, kind of went over my head. So, like, I just didn't-- I, like, shied away from those.	"I felt that those were helpful because then you get to-- most of them are exercises that we didn't get to in class, so it helped to have someone walk through all of them"	Yeah, I did it the first maybe-- I did it the required weeks and then maybe the first two weeks after that I think, and I have not done them since. I should! <laughs> But I have not been doing them since. I found them useful at the beginning of class-- 'cause I had done-- I had programmed a bit before this
6	N/A	I tend to look at the worked-out solutions on my own at different times, but I rarely-- I didn't even look at the extra credit commenting the code	I think the commenting the code for extra credit definitely helped sort of the worked-examples that they would give us, I think that helped a lot because … it made you actually engage with the program on a very detailed level... you have to actually say, "All right, what is this line actually doing and how does that relate to sort of the overall functionality that you're trying to implement?

LIST OF REFERENCES

LIST OF REFERENCES

Aleven, V. A. W. M. M., & Koedinger, K. R. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based Cognitive Tutor. *Cognitive Science*, *26*(2), 147–179. doi: 10.1016/S0364-0213(02)00061-7

Anderson, J. R., Fincham, J. M., & Douglass, S. (1997). The role of examples and rules in the acquisition of a cognitive skill. *Journal of experimental psychology: learning, memory, and cognition*, *23*(4), 932.

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from examples: Instructional principles from the worked examples research. *Review of educational research*, *70*(2), 181–214.

Ayres, P., & Sweller, J. (1990). Locus of difficulty in multistage mathematics problems. *The American Journal of Psychology*, 167–193.

Ayres, P. L. (1993). Why goal-free problems can facilitate learning. *Contemporary Educational Psychology*, *18*(3), 376–381.

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Communications of the ACM*, *26*(9), 677–679. Retrieved from `internal-pdf://bayman-3226972160/bayman.pdf` doi: http://doi.acm.org/10.1145/358172.358408

Bell, P. (2000). Scientific arguments as learning artifacts: Designing for learning from the web with kie. *International Journal of Science Education*, *22*(8), 797–817.

Biggs, J. B., & Collis, K. F. (1982). *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome).* New York: Academic Press, 1982.

Bransford, J. D., Brown, A. L., & Cocking, R. (2000). *How People Learn.* Washington DC: National Academy Press.

Carroll, W. M. (1994). Using worked examples as an instructional support in the algebra classroom. *Journal of Educational Psychology*, *86*(3), 360.

Case, J. M., & Light, G. (2011). Emerging methodologies in engineering education research. *Journal of Engineering Education*, *100*(1), 186.

Chesnais, A. (2012, January). Acm's annual report. *Commun. ACM*, *55*(1), 9–13. Retrieved from `http://doi.acm.org/10.1145/2063176.2063179` doi: 10.1145/2063176.2063179

Chi, M., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive science*, *13*(2), 145–182.

Chi, M. T., Glaser, R., & Rees, E. (1981). *Expertise in problem solving.* (Tech. Rep.). DTIC Document.

Chi, M. T., & Roy, M. (2010). How adaptive is an expert human tutor? In *International conference on intelligent tutoring systems* (pp. 401–412).

Chi, M. T. H. (2009). Active-Constructive-Interactive: A Conceptual Framework for Differentiating Learning Activities. *Topics in Cognitive Science*, *1*(1), 73–105. doi: 10.1111/j.1756-8765.2008.01005.x

Chi, M. T. H., De Leeuw, N., Chiu, M. H., & Lavancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, *18*(3), 439–477. doi: 10.1016/0364-0213(94)90016-7

Chiu, J. L., & Chi, M. T. (2014). Supporting self-explanation in the classroom. *Acknowledgments and Dedication*, 91.

Cierniak, G., Scheiter, K., & Gerjets, P. (2009). Explaining the split-attention effect: Is the reduction of extraneous cognitive load accompanied by an increase in germane cognitive load? *Computers in Human Behavior*, *25*(2), 315–324.

A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. (2009). *Proceedings of the Fifth International Workshop on Computing Education Research Workshop - ICER '09*(2009), 117–128. Retrieved from `http://portal.acm.org/citation.cfm?doid=1584322.1584336` doi: 10.1145/1584322.1584336

Confrey, J. (1990). A review of the research on student conceptions in mathematics, science, and programming. *Review of research in education*, *16*, 3–56.

Cowan, N. (2001). The magical number 4 in short term memory. A reconsideration of storage capacity. *Behavioral and Brain Sciences*, *24*(4), 87–186.

Cowan, N. (2010). The magical mystery four how is working memory capacity limited, and why? *Current directions in psychological science*, *19*(1), 51–57.

De Jong, T. (2010). Cognitive load theory, educational research, and instructional design: some food for thought. *Instructional science*, *38*(2), 105–134.

Difficulties in Learning and Teaching Programming Views of Students and Tutors. (2002). *Education and Information Technologies*, *7*, 55–66. Retrieved from `http://www.springerlink.com/index/h1u587b0r3e9eja3.pdf` doi: 10.1023/A:1015362608943

Du Boulay, B. (1986). Some Difficulties of Learning to Program. *Journal Of Educational Computing Research*, *2*(1), 57. doi: 10.1017/CBO9781107415324.004

EDUCATION, S. W. G. O. C. U., Co-Chairs, P. T., Petzold, L., Shiflet, A., Vakalis, I., Jordan, K., & John, S. S. (2011). Undergraduate computational science and engineering education. *SIAM review*, *53*(3), 561–574.

Ericsson, K. A., & Kintsch, W. (1995). Long-term working memory. *Psychological review*, *102*(2), 211.

Exter, M. (2014). Comparing educational experiences and on-the-job needs of educational software designers. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 355–360).

Exter, M., & Turnage, N. (2012). Exploring experienced professionals reflections on computing education. *ACM Transactions on Computing Education (TOCE)*, *12*(3), 12.

Fleury, A. E. (2000). Programming in Java: Student-Constructed Rules. *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, 197–201. Retrieved from `http://portal.acm.org/citation.cfm?doid=331795.331854` doi: 10.1145/331795.331854

Gibson, J. J. (2014). *The ecological approach to visual perception: classic edition.* Psychology Press.

Glotzer, S. C., Kim, S., Cummings, P. T., Deshmukh, A., Head-Gordon, M., Karniadakis, G., ... Shinozuka, M. (2009). *International assessment of research and development in simulation-based engineering and science. panel report* (Tech. Rep.). DTIC Document.

Goldman, K., Gross, P., Heeren, C., Herman, G., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2008). Identifying important and difficult concepts in introductory computing courses using a delphi process. *ACM SIGCSE Bulletin*, *40*(1), 256. doi: 10.1145/1352322.1352226

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Transactions on Computing Education*, *10*(2), 1–29. doi: 10.1145/1789934.1789935

Gray, S., St Clair, C., James, R., & Mead, J. (2007). Suggestions for graduated exposure to programming concepts using fading worked examples. In *Proceedings of the third international workshop on computing education research* (pp. 99–110).

Guzdial, M. (2011). From science to engineering. *Communications of the ACM*, *54*(2), 37–39.

Hafner, J. (2000). Atomic-scale computational materials science. *Acta Materialia*, *48*(1), 71–92.

Hartley, J. (2004). Case study research.

Hattie, J., & Timperley, H. (2007). The power of feedback. *Review of educational research*, *77*(1), 81–112.

Hausmann, R. G. M., & Chi, M. T. H. (2002). Can a computer interface support self-explaining. *Cognitive Technology*, *7*, 4–14.

Joint Task Force on Computing Curricula, A. f. C. M. A., & Society, I. C. (2013). *Computer science curricula 2013: Curriculum guidelines for undergraduate degree programs in computer science.* New York, NY, USA: ACM. (999133)

Kaczmarczyk, L. C., Petrick, E. R., East, J. P., & Herman, G. L. (2010). Identifying student misconceptions of programming. *Proceedings of the 41st ACM technical symposium on Computer science education - SIGCSE '10*, 107. Retrieved from `http://portal.acm.org/citation.cfm?doid=1734263.1734299` doi: 10.1145/1734263.1734299

Kalyuga, S., Ayres, P., Chandler, P., & Sweller, J. (2003). The expertise reversal effect. *Educational psychologist*, *38*(1), 23–31.

Kalyuga, S., Chandler, P., Tuovinen, J., & Sweller, J. (2001). When problem solving is superior to studying worked examples. *Journal of educational psychology*, *93*(3), 579.

Kalyuga, S., & Sweller, J. (2004). Measuring knowledge to optimize cognitive load factors during instruction. *Journal of educational psychology*, *96*(3), 558.

Kapur, M., & Bielaczyc, K. (2012). Designing for productive failure. *Journal of the Learning Sciences*, *21*(1), 45–83.

Kester, L., Paas, F., & van Merriënboer, J. J. G. (2010). Instructional Control of Cognitive Load in the Design of Complex Learning Envinronments. In J. Plass, R. Moreno, & R. Brünken (Eds.), *Cognitive load theory* (pp. 109–130).

Khan, S., & VanWynsberghe, R. (2008). Cultivating the under-mined: Cross-case analysis as knowledge mobilization. In *Forum qualitative sozialforschung/forum: Qualitative social research* (Vol. 9).

Kirschner, P. A., Ayres, P., & Chandler, P. (2011). Contemporary cognitive load theory research: The good, the bad and the ugly. *Computers in Human Behavior*, *27*(1), 99–105.

Kohlbacher, F. (2006). The use of qualitative content analysis in case study research. In *Forum qualitative sozialforschung/forum: Qualitative social research* (Vol. 7).

Küchemann, D., & Hoyles, C. (2003). The quality of students'explanations on a non-standard geometry item. In *Proceedings of cerme* (Vol. 3).

Kuhn, D., & Katz, J. (2009). Are self-explanations always beneficial? *Journal of Experimental Child Psychology*, *103*(3), 386–394.

Lister, R. (2011). Concrete and other neo-piagetian forms of reasoning in the novice programmer. *Conferences in Research and Practice in Information Technology Series*, *114*, 9–18.

Lister, R., Corney, M., Curran, J., Souza, D. D., Fidge, C., Gluga, R., … Teague, D. (2012). Toward a Shared Understanding of Competency in Programming : An Invitation to the BABELnot Project. *14th Australasian Computing Education Conference*, *123*(February), 53–60.

Lombrozo, T. (2006). The structure and function of explanations. *Trends in Cognitive Sciences*, *10*(10), 464–470. doi: 10.1016/j.tics.2006.08.004

Lopez, M., Whalley, J., Robbins, P., & Lister, R. (2008). Relationships between reading, tracing and writing skills in introductory programming. *Proceeding of the fourth international workshop on Computing education research - ICER '08*, 101–112. Retrieved from `http://portal.acm.org/citation.cfm?doid=1404520.1404531` doi: 10.1145/1404520.1404531

Magana, A. J., Brophy, S. P., & Bodner, G. M. (2010). the Transparency Paradox: Computational Simulations As Learning Tools for Engineering Graduate Education. In *Aera meeting: Understanding complex ecologies in a changing world.*

Magana, A. J., Brophy, S. P., & Bodner, G. M. (2012). Student Views of Engineering Professors Technological Pedagogical Content Knowledge for Integrating Computational Simulation Tools in Nanoscale. *International Journal of Enginnering Education*, *28*(5), 1033–1045. Retrieved from `http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Student+Views+`

Magana, A. J., Falk, M. L., & Reese Jr, M. J. (2013). Introducing discipline-based computing in undergraduate engineering education. *ACM Transactions on Computing Education (TOCE)*, *13*(4), 16.

Magana, A. J., Falk, M. L., Vieira, C., & Reese, M. J. (2016). A case study of undergraduate engineering students ' computational literacy and self-beliefs about computing in the context of authentic practices. *Computers in Human Behavior*, *61*, 427–442. Retrieved from `http://dx.doi.org/10.1016/j.chb.2016.03.025` doi: 10.1016/j.chb.2016.03.025

Magana, A. J., & Mathur, J. I. (2012). Motivation, Awareness, and Perceptions of Computational Science. *Computing in Science & Engineering*, 74–79.

Magana, A. J., Vieira, C., Polo, F. G., Yan, J., & Sun, X. (2013). An exploratory survey on the use of computation in undergraduate engineering education. In *2013 ieee frontiers in education conference (fie)* (pp. 94–100).

Makatchev, M., Jordan, P. W., & VanLehn, K. (2004). Abductive theorem proving for analyzing student explanations to guide feedback in intelligent tutoring systems. *Journal of Automated Reasoning*, *32*(3), 187–226. doi: 10.1023/B:JARS.0000044823.50442.cd

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *ACM Computing Surveys (CSUR)*, *13*(1), 121–141.

Merril, M. D. (2002). First Principles of Instruction. *Educational Technology Research and Development*, *50*(3), 43–59.

Miller, C. S., Lehman, J. F., & Koedinger, K. R. (1999). Goals and learning in microworlds. *Cognitive Science*, *23*(3), 305–336.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological review*, *63*(2), 81.

Moreno, R., & Park, B. (2010a). Cognitive load theory: Historical development and relation to other theories. *Cognitive load theory*, 9–28.

Moreno, R., & Park, B. (2010b). Cognitive load theory: historical development and relation to other theories. In J. Plass, R. Moreno, & R. Brünken (Eds.), *Cognitive load theory* (pp. 9–28). New York: Cambridge University Press.

Moreno, R., Reisslein, M., & Ozogul, G. (2009). Optimizing worked-example instruction in electrical engineering: The role of fading and feedback during problem-solving practice. *Journal of Engineering Education*, *98*(1), 83–92.

Mselle, L. J., & Twaakyondo, H. (2012). The impact of Memory Transfer Language (MTL) on reducing misconceptions in teaching programming to novices. *International Journal of Machine Learning and Applications*, *1*, 1–6. doi: 10.4102/ijmla.v1i1.3

Mulder, Y. G., Bollen, L., de Jong, T., & Lazonder, A. W. (2016). Scaffolding learning by modelling: The effects of partially worked-out models. *Journal of research in science teaching*, *53*(3), 502–523.

NSF. (2011). *Empowering the Nation Through Discovery and Innovation* (Tech. Rep.). National Science Foundation.

Owen, E., & Sweller, J. (1985). What do students learn while solving mathematics problems? *Journal of Educational Psychology*, *77*(3), 272.

Paas, F., & Kirschner, F. (2012). Goal-free effect. In *Encyclopedia of the sciences of learning* (pp. 1375–1377). Springer.

Paas, F., Renkl, A., & Sweller, J. (2003, mar). Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist*, *38*(1), 1–4. Retrieved from `http://www.tandfonline.com/doi/abs/10.1207/S15326985EP3801_1` doi: 10.1207/S15326985EP3801_1

Paas, F., Tuovinen, J. E., Tabbers, H., & Van Gerven, P. W. (2003). Cognitive load measurement as a means to advance cognitive load theory. *Educational psychologist*, *38*(1), 63–71.

Paas, F. G. (1992). Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive-load approach. *Journal of educational psychology*, *84*(4), 429.

Paas, F. G., & Van Merriënboer, J. J. (1993). The efficiency of instructional conditions: An approach to combine mental effort and performance measures. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, *35*(4), 737–743.

Paas, F. G., & Van Merriënboer, J. J. (1994). Instructional control of cognitive load in the training of complex cognitive tasks. *Educational psychology review*, *6*(4), 351–371.

Paas, F. G. W. C., & Van Merrinboer, J. J. G. (1994). Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive-load approach. *Journal of Educational Psychology*, *86*(1), 122–133. doi: 10.1037/0022-0663.86.1.122

Patton, M. Q. (2002). *Qualitative Research & Evaluation Methods* (2nd ed.). Thousand Oaks: Sague Publications.

Pea, R. D. (1986). Language-independent conceptual" bugs" in novice programming. *Journal of Educational Computing Research*, *2*(1), 25–36. doi: 10.2190/689T-1R2A-X4W4-29J2

Pea, R. D., Soloway, E., & Spohrer, J. C. (1987). The buggy path to the development of programming expertise. *Focus on Learning Problems in Mathematics*, *9*(1), 5–30.

Pirolli, P., & Recker, M. (1994). Learning strategies and transfer in the domain of programming. *Cognition and instruction*, *12*(3), 235–275.

PITAC. (2005). *Computational Science: Ensuring Americas Competitiveness* (Tech. Rep.). President's Information Technology Advisory Committee. Retrieved from `http://www.nitrd.gov/pitac/reports/20050609_computational/computational.pdf`

Prasada, S., & Dillingham, E. M. (2006). Principled and statistical connections in common sense conception. *Cognition*, *99*(1), 73–112.

Radermacher, A., & Walia, G. (2013). Gaps between industry expectations and the abilities of graduates. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 525–530).

Reisslein, J., Atkinson, R. K., Seeling, P., & Reisslein, M. (2006). Encountering the expertise reversal effect with a computer-based environment on electrical circuit analysis. *Learning and instruction*, *16*(2), 92–103.

Renkl, A. (1997). Learning from worked-out example: A study on individual differences. *Cognitive sciences*, *21*, 1–29.

Renkl, A. (2005). The worked-out examples principle in multimedia learning. In R. E. Mayer (Ed.), *The cambridge handbook of multimedia learning.* New York: Cambridge University Press.

Renkl, A., & Atkinson, R. K. (2003). Structuring the transition from example study to problem solving in cognitive skill acquisition: A cognitive load perspective. *Educational psychologist*, *38*(1), 15–22.

Renkl, A., Atkinson, R. K., Maier, U. H., & Staley, R. (2002). From example study to problem solving: Smooth transitions help learning. *The Journal of Experimental Education*, *70*(4), 293–315.

Resnick, M., Berg, R., & Eisenberg, M. (2000). Beyond black boxes: Bringing transparency and aesthetics back to scientific investigation. *The Journal of the Learning Sciences*, *9*(1), 7–30.

Rogalski, J., & Samurçay, R. (1990). *Acquisition of programming knowledge and skills* (Vol. 18). Retrieved from `http://128.232.0.20/teaching/1011/R201/ppig-book/ch2-4.pdf\npapers3://public`

Sahami, M., Roach, S., Cuadros-Vargas, E., & LeBlanc, R. (2013). Acm/ieee-cs computer science curriculum 2013: Reviewing the ironman report. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 13–14).

Salden, R. J., Aleven, V., Schwonke, R., & Renkl, A. (2010). The expertise reversal effect and worked examples in tutored problem solving. *Instructional Science*, *38*(3), 289–307.

Sandoval, W. A., & Millwood, K. A. (2005). The quality of students' use of evidence in written scientific explanations. *Cognition and instruction*, *23*(1), 23–55.

Schwonke, R., Renkl, A., Salden, R., & Aleven, V. (2011). Effects of different ratios of worked solution steps and problem solving opportunities on cognitive load and learning outcomes. *Computers in Human Behavior*, *27*(1), 58–62.

Schworm, S., & Renkl, A. (2006). Computer-supported example-based learning: When instructional explanations reduce self-explanations. *Computers and Education*, *46*(4), 426–445. doi: 10.1016/j.compedu.2004.08.011

Shafto, P., & Coley, J. D. (2003). Development of categorization and reasoning in the natural world: novices to experts, naive similarity to ecological knowledge. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, *29*(4), 641.

Shavelson, R. J., Ruiz-Primo, M. A., Li, M., & Ayala, C. C. (2003). Evaluating new approaches to assessing learning. *Retrieved October*, *21*, 2003.

Sheard, J., Carbone, A., Lister, R., Simon, B., Thompson, E., & Whalley, J. L. (2008). Going solo to assess novice programmers. In *Acm sigcse bulletin* (Vol. 40, pp. 209–213).

Soloway, E. (1986). Learning to program= learning to construct mechanisms and explanations. *Communications of the ACM*, *29*(9), 850–858.

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education*, *13*(2), 1–31. doi: 10.1145/2483710.2483713

Southerland, S. A., Abrams, E., Cummins, C. L., & Anzelmo, J. (2001). Understanding students' explanations of biological phenomena: Conceptual frameworks or p-prims? *Science Education*, *85*(4), 328–348.

Stark, R., Mandl, H., Gruber, H., & Renkl, A. (2002). Conditions and effects of example elaboration. *Learning and Instruction*, *12*(1), 39–60.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science*, *12*(2), 257–285.

Sweller, J. (2010). Cognitive load theory: Recent theoretical advances. *Cognitive load theory*, 29–47.

Sweller, J., Ayres, P., & Kalyuga, S. (2011). *Cognitive load theory.* New York: Springer.

Sweller, J., & Chandler, P. (1994). Why some material is difficult to learn. *Cognition and Instruction*, *12*(3), 185–233.

Sweller, J., Mawer, R. F., & Ward, M. R. (1983). Development of expertise in mathematical problem solving. *Journal of Experimental Psychology: General*, *112*(4), 639.

Sweller, J., van Merriënboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive architecture and instructional design. *Educational Psychology Review*, *10*(3), 251–296.

Tan, G., & Venables, A. (2010). Wearing the Assessment "BRACElet". *Journal of Information Technology Education*, *9*, IIP–25–IIP–34. Retrieved from `http://search.ebscohost.com/login.aspx?direct=true&AuthType=cookie,ip,uid&db`

Taylor, C., Zingaro, D., Porter, L., Webb, K. C., Lee, C. B., & Clancy, M. (2014). Computer science concept inventories: past and future. *Computer Science Education*, *24*(4), 253–276.

Tew, A. E., & Guzdial, M. (2011). The fcs1: a language independent assessment of cs1 knowledge. In *Proceedings of the 42nd acm technical symposium on computer science education* (pp. 111–116).

Thompson, E., Luxton-Reilly, A., Whalley, J. L., Hu, M., & Robbins, P. (2008). Bloom's taxonomy for CS assessment. *Conferences in Research and Practice in Information Technology Series*, *78*(January), 155–161.

Trafton, J. G., & Reiser, B. J. (1994). *The contributions of studying examples and solving problems to skill acquisition.* Unpublished doctoral dissertation, Citeseer.

Turner, P. R., Shiflet, A. B., Cunningham, S., Stewart, K., Phillips, A. T., & Vakalis, I. E. (2002). Undergraduate computational science and engineering programs and courses. *ACM SIGCSE Bulletin*, *34*(1), 96–97.

Van Merrienboer, J. J. G., & Sluijsmans, D. M. A. (2009). Toward a synthesis of cognitive load theory, four-component instructional design, and self-directed learning. *Educational Psychology Review*, *21*(1), 55–66. doi: 10.1007/s10648-008-9092-5

VanLehn, K. (1996). Cognitive skill acquisition. *Annual review of psychology*, *47*(1), 513–539.

Van Merriënboer, J. J., Clark, R. E., & De Croock, M. B. (2002). Blueprints for complex learning: The 4c/id-model. *Educational technology research and development*, *50*(2), 39–61.

Van Merriënboer, J. J., Kirschner, P. A., & Kester, L. (2003). Taking the load off a learner's mind: Instructional design for complex learning. *Educational psychologist*, *38*(1), 5–13.

Van Merrienboer, J. J., & Sweller, J. (2005). Cognitive load theory and complex learning: Recent developments and future directions. *Educational psychology review*, *17*(2), 147–177.

van Merriënboer, J. J. G., Clark, R. E., & Croock, M. B. M. (2002). Blueprints for complex learning: The 4C/ID-model. *Educational Technology Research and Development*, *50*(2), 39–64. doi: 10.1007/BF02504993

Vieira, C., Roy, A., Magana, A. J., Falk, M. L., & Reese Jr., M. J. (2016, June). In-code comments as a self-explanation strategy for computational science education. In *2016 asee annual conference & exposition.* New Orleans, Louisiana: ASEE Conferences. (https://peer.asee.org/25642)

Vieira, C., Yan, J., & Magana, A. J. (2015). Exploring Design Characteristics of Worked Examples to Support Programming and Algorithm Design. *Journal of Computational Science Education*, *6*(1), 2–15.

Watson, J. R., Prieto, T., & Dillon, J. S. (1997). Consistency of students' explanations about combustion. *Science Education*, *81*(4), 425–443.

Whalley, J. L., & Lister, R. (2009). The BRACElet 2009 . 1 (Wellington) Specification. In *Eleventh australasian computing education conference (ace2009)* (Vol. 1). doi: 10.1113/jphysiol.2007.148254

Wickens, C. D., Hollands, J. G., Banbury, S., & Parasuraman, R. (2015). *Engineering psychology & human performance.* Psychology Press.

Williams, J. J., & Lombrozo, T. (2010). The role of explanation in discovery and generalization: Evidence from category learning. *Cognitive Science*, *34*(5), 776–806. doi: 10.1111/j.1551-6709.2010.01113.x

Williams, J. J., Lombrozo, T., & Rehder, B. (2010). Why does explaining help learning? insight from an explanation impairment effect. In *Proceedings of the 32nd annual conference of the cognitive science society* (pp. 2906–2911).

Wirth, J., Künsting, J., & Leutner, D. (2009). The impact of goal specificity and goal type on learning outcome and cognitive load. *Computers in Human Behavior*, *25*(2), 299–305.

Xun, G., & Land, S. M. (2004). A conceptual framework for scaffolding iii-structured problem-solving processes using question prompts and peer interactions. *Educational Technology Research and Development*, *52*(2), 5–22.

Yaar, O. (2013). Computational Math, Science, and Technology (C-MST) Approach to General Education Courses. *Cognition and Instruction*, *1*(4), 2–10.

Yin, R. K. (2009). Case study research: Design and methods 4th ed. In *United states: Library of congress cataloguing-in-publication data.*

VITA

VITA

Camilo Vieira, M. E.

Ph.D. Student and Research Assistant of Computer and Information Technology Department of Computer and Information Technology at Purdue University

Professional Preparation:

- B.S., Universidad Eafit, Medellin-Colombia, in Systems Engineering, June 2008.
- M.E., Universidad Eafit, Medellin-Colombia, in Engineering - Educational Technologies, June, 2013.

Appointments:

- Purdue University, West Lafayette, Indiana, Research Assistant, Department of Computer and Information Technology. January 2013-Present.
- Universidad Eafit, Medellin-Colombia, Continuous Lecturer, Department of Systems Engineering. January 2012-December 2013
- Universidad Eafit, Medellin-Colombia, Research Assistant, Department of Systems Engineering. January 2011-December 2012
- Universidad Autonoma Latinoamericana, Medellin-Colombia, Continuous Lecturer, Department of Informatics Engineering. January 2012, December 2012
- Servicio Nacional de Aprendizaje (SENA) Medellin-Colombia, Continuous Lecturer, Centro de Servicios de Salud.
- Suramericana de Seguros S.A. Medellin-Colombia, Software Analyst, September 2007 December 2008

- Trebol Software S. A. (AXEDE), Medellin-Colombia, Development Engineer, January 2006-August 2007