Algorithmic pedagogy: Using code analysis to deliver targeted supplemental instruction

by

Nathan H. Bean

B.S., Kansas State University, 2003
M.Sc., University of Abertay Dundee, 2005
M.A., Kansas State University, 2009

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Curriculum and Instruction
College of Education

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2022

Abstract

Learning to program has long been known to be a difficult task, requiring a student to develop both fluency in the syntax and grammar of a formal programming language and learn the problem-solving approaches and techniques of computational thinking. The successful teaching strategies of the past have involved maintaining small teacher-student ratios and large amounts of supplemental instruction in lab courses. However, recent growth in the demand for programming courses from both computer science major and nonmajor students has drastically outpaced the expansion of computer science faculty and created a shortage in available lab space and time across American universities.

This study involved creating a software tool for automatically delivering targeted supplemental instruction to students based on a real-time algorithmic analysis of the program code they were writing. This approach was piloted with students enrolled in a sophomore-level object-oriented software development course. The majority of students reported finding the detection and reporting of issues in their code helpful. Moreover, students who were less proficient programmers entering the course who utilized the tool showed statistically significant improvement in their final exam grade over those who did not. Thus, adopting the strategy piloted in this study could improve instruction in larger classes and relieve some of the strain on overburdened computer science departments while providing additional learning benefits for students.

Algorithmic pedagogy:
Using code analysis to deliver targeted supplemental instruction

by

Nathan H. Bean

B.S., Kansas State University, 2003
M.Sc., University of Abertay Dundee, 2005
M.A., Kansas State University, 2009

A DISSERTATION

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Curriculum and Instruction
College of Education

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2022

Approved by:

Major Professor
Jacqueline Spears

Copyright

Abstract

Learning to program has long been known to be a difficult task, requiring a student to develop both fluency in the syntax and grammar of a formal programming language and learn the problem-solving approaches and techniques of computational thinking. The successful teaching strategies of the past have involved maintaining small teacher-student ratios and large amounts of supplemental instruction in lab courses. However, recent growth in the demand for programming courses from both computer science major and nonmajor students has drastically outpaced the expansion of computer science faculty and created a shortage in available lab space and time across American universities.

This study involved creating a software tool for automatically delivering targeted supplemental instruction to students based on a real-time algorithmic analysis of the program code they were writing. This approach was piloted with students enrolled in a sophomore-level object-oriented software development course. The majority of students reported finding the detection and reporting of issues in their code helpful. Moreover, students who were less proficient programmers entering the course who utilized the tool showed statistically significant improvement in their final exam grade over those who did not. Thus, adopting the strategy piloted in this study could improve instruction in larger classes and relieve some of the strain on overburdened computer science departments while providing additional learning benefits for students.

Table of Contents

List of Figures

List of Tables

Chapter 1 - Introduction

This chapter provides an overview of the context and motivation of this research project, describes the theoretical framework with which it was framed, identifies the problem it sought to address, describes its goals and design, and provides a list of definitions for domain-specific terms used throughout the proposal.

Motivation

That the development of the digital computer has had a transformative effect on society, politics, and economics in the latter half of the 20th century is hard to deny. Indeed, many historians and economists describe these years as a shift from the "industrial age" to an "information age," made possible by the ability of computer networks to readily store, process, and share vast quantities of information. This has enabled the expansion of multinational corporations, an increase in federalization in nation-state collations, such as the United States and the European Union, and provided new educational opportunities for much of the world's population.

The National Science Foundation's Blue-Ribbon Advisory Panel recognizes the power of these new technologies and their concrete implementations, which they collectively refer to as *cyberinfrastructure*, to transform science and engineering endeavors:

> [… cyberinfrastructure …] enables more ubiquitous, comprehensive knowledge environments that become functionally complete for specific research communities in terms of people, data, information, tools, and instruments and that include unprecedented capacity for computational [*sic*], storage, and communication. Such environments enable teams to share and collaborate over time and over geographic, organizational, and

> disciplinary distance. They enable individuals working alone to have access to more and better information and facilities for discovery and learning. They can serve individuals, teams and organizations in ways that revolutionize *what they can do*, *how they do it*, and *who participates*. (Atkins et al., 2003, pp. 12-13)

This cyberinfrastructure, in turn, supports and enables those persons who Douglas Engelbart, the computer scientist whose work foreshadowed the development of desktop operating systems, the Internet, and cloud computing, referred to as *intellectual workers.* This new class of worker uses computing technology to communicate, structure, store, and retrieve data; develop models and simulations; and otherwise augment their natural problem-solving abilities (Engelbart, 1962). In the information age, all industries have come to rely on such intellectual workers to some degree, and growth in these areas has led to greater efficiencies and innovative approaches within their fields.

The United States has a tremendous demand for both the cyberinfrastructure creators and the intellectual workers they support. In 2014, the United States Bureau of Labor Statistics estimated that over 3.5 million workers were directly employed in creating cyberinfrastructure and projected high job growth within the industry (Bureau of Labor Statistics, 2014a)[1]. President Obama made this concern a priority for his administration, participating in the national *Hour of Code* activity and launching his *CS For All* initiative in 2016, expanding federal education

[1] The Bureau of Labor Statistics predicts faster than average job growth for all but two categories of Computer and Information Technology Occupations, and average job growth for the remaining two (Bureau of Labor Statistics, 2014b).

spending and NSF programs and funding, with the goal of bringing Computer Science to all the nations' K-12 students (Smith, 2016).

Jeanette Wing mustered the knowledge, skills, and abilities that Douglas Engelbart saw as core competencies for his intellectual workers under the conceptual banner of computational thinking, "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent (Wing, 2010)." Wing argued that computational thinking is transforming the way we approach problems in multiple fields beyond computer science:

> We have witnessed the influence of computational thinking on other disciplines. For example, machine learning has transformed statistics. Statistical learning is being used for problems on a scale, in terms of both data size and dimension, unimaginable only a few years ago. […] Computer science's contribution to biology goes beyond the ability to search through vast amounts of sequence data looking for patterns. The hope is that data structures and algorithms – our computational abstractions and methods – can represent the structure of proteins in ways that elucidate their function. Computational biology is changing the way biologists think. Similarly, computational game theory is changing the way economist think; nanocomputing, the way chemists think; and quantum computing, the way physicists think. (Wing, 2006, p. 34)

Wing's paper set off a flurry of discussion in the literature involving the identification of more instances of computational thinking's influence on fields of human endeavor, clarifying what computational thinking might consist of and encouraging the adoption of computational thinking as a core competency for all 21st century learners (Bundy, 2007; Wing, 2008; Lu & Fletcher, 2009; Barr et al., 2011).

A consensus seems to be emerging from both the academic literature and governmental reports that developing computational thinking and leveraging computing technology to its fullest potential are critical 21st century workforce skills for all disciplines. It would seem Douglas Engelbart's vision of an intellectual workforce is coming to fruition. Or will be, if we can overcome some not-insignificant challenges in preparing this intellectual workforce to develop and use these skills.

Programming and Computational Thinking

In 2010-2011, The National Science Foundation convened multiple meetings and workshops to identify exactly what constituted computational thinking and how it could be made accessible to various audiences (Lin, 2010; Barr et al., 2011). Emerging from this discussion was an evolving conception of the relationship between programming and computational thinking. Many researchers and educators saw learning programming as an entry point into learning to think computationally, as Ursa Wolz did when arguing that "programming is a language for expressing ideas. You have to learn how to read and write that language in order to be able to think in that language" (Lin, 2010, p. 13), and Mitchel Resnick, who suggested "programming, like writing, is a means of expression and an entry point to developing new ways of thinking" (Lin, 2010, p. 13).

Lu and Fletcher (2009) offer a counter perspective, as they propose that programming is a *formal expression* of computation. Much like a mathematical proof is an argument based in mathematical reasoning and established mathematical facts, a program is a computational process for solving a particular problem, *a crystallization of computational thought in precise and unambiguous terms*. With this understanding, students must first be able to think computationally before they will be effective at programming – just as developing a geometric proof requires facility with the foundations of geometry.

One of the most robust attempts to teach computer science and computational thinking without programming, and indeed, without computing technology at all is Computer Science Unplugged (CS Unplugged). Developed by the CS Education Group of the University of Canterbury, CS Unplugged is a curriculum for teaching computer science and computational thinking without computers, often through very hands-on activities. However, some research has suggested that students participating in CS Unplugged activities often miss the connections between the activity and the field of computer science, and indeed find the field less interesting after participating in activities (Taub et al., 2012). These findings raise questions about the viability of preparing students to program by teaching computational thinking without programming.

The Challenge of Teaching Programming

The nature of this chicken-and-egg relationship between computational thinking and programming may provide an explanation for why learning to program is such a hard task, a reality that has long been recognized by the computer science education community (Guzdial, 2004; Bennedsen & Caspersen, 2007). Cognitive Load Theory suggests that only a handful of novel concepts can be held in a learner's mind at any given moment, and overloading this capacity severely reduces the likelihood of learning taking place (Paas, Renkl, & Sweller, 2003). If the dominant paradigm for learning computational thinking is through learning to program, but facility with computational thinking is necessary to carry out programming effectively, then we are very likely overwhelming the cognitive resources of our students. Indeed, this may account for introductory college programming courses typically featuring a pass rate of 67% (Watson &

Li, 2014). Students completing their first year of a computer science degree program often are unable to author simple programs – in many cases, unable to write code that would not crash (McCraken et al., 2001).

One effective and well-known strategy for teaching challenging and deeply connected sets of skills (like mathematical proofs and literature analysis) is to spend years preparing students by gradually building a foundation of skills and concepts needed to support these higher-level skills. For example, students have 10 or more years of mathematical learning before proofs are introduced and have experienced a childhood filled with stories before attempting literature analysis. However, this is not the case for computer science students. The Computer Science Teachers' Association highlighted the issue in their 2010 report *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age*:

> At a time when computing is driving job growth and new scientific discovery, it is unacceptable that roughly two-thirds of the entire country has few computer science standards for secondary school education, K-8 computer science standards are deeply confused, few states count computer science as a core academic subject for graduation, and computer science teacher certification is deeply flawed. (Wilson et al., 2010, p. 9)

Without this background, fledgling computer scientists are forced to compress many years' worth of learning into a few short semesters.

There has been strong advocacy for treating computational thinking as equally important as reading, writing, and arithmetic in the K-12 curriculum (Wing, 2006; Lu & Fletcher, 2009; Barr et al., 2011). Many states have responded by adopting K-12 CS Standards and promoting computer science in K-12, which will likely ease the challenge for undergraduate CS majors in the future. However, that does not address the challenge of students entering postsecondary computer science courses now.

The Enrollment Surge

It should hardly be surprising that the recognition of the growing role of computational thinking and programming in our modern society should be matched by increased enrollment in computer science courses. Indeed, the field of computer science is now facing its third major surge in enrollment numbers. The National Academy of Sciences, Engineering, and Medicine reports that "the average number of CS majors increased for large departments (25 or more tenure-track faculty) from 341 to 970 and for small departments from 158 to 499 majors," (2018, p. 2) since 2006. The trend accelerated in 2012, roughly doubling of the number of enrolled students in the time since; a growth far more pronounced than the earlier two booms associated with the introduction of the personal computer in the 1980s and the Internet in the 1990s (Camp et al., 2017a).

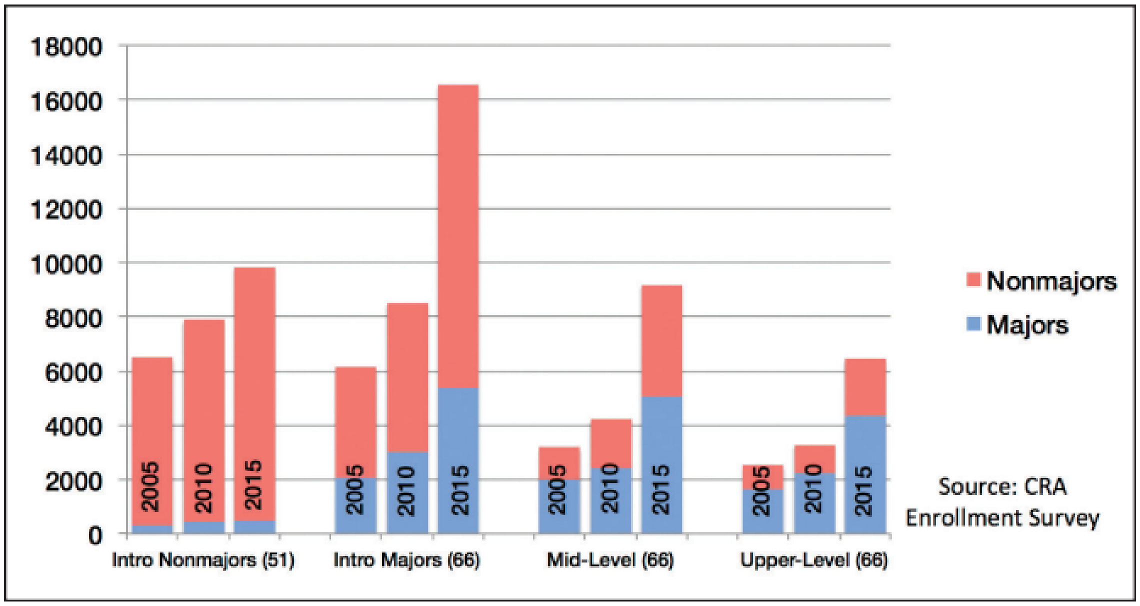
The enrollment growth trend is not limited to computer science majors; increasing numbers of nonmajors also have begun enrolling in computer science courses:

> Between 2005 and 2015, in representative courses primarily intended for majors, the number of nonmajors in computing courses increased at a rate equal to or greater than the increase in majors. For the intro majors course, majors increased by 152% and nonmajors by 177%; for the mid-level course, majors increased by 152% and nonmajors by 251%; and for the upper-level course, majors increased by 165% and nonmajors by 143%. (Camp et al., 2017a, p. 48)

This growth can be visually seen in Figure 1.1.

Figure 1.1

Nonmajor vs. Major Enrollment in CS Courses

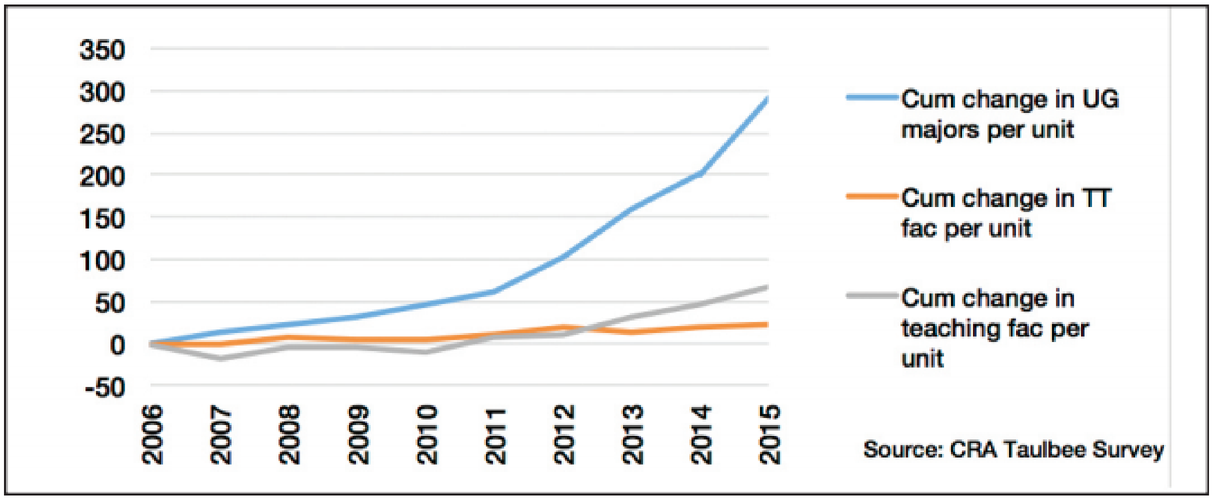


Note: Major enrollments are in blue and nonmajors in red. Parenthesized numbers are sample sizes. Reprinted from "Generation CS: The Growth of Computer Science," by Camp et al., 2017, *ACM Inroads, 8*(2), 48. Copyright 2017 by the Authors. Reprinted with permission.

As might be expected, this unprecedented enrollment growth has created strains on Computer Science departments, including inadequate classroom space, insufficient faculty numbers, and increased faculty workloads (Camp et al, 2017c). These problems are exacerbated by the lack of a corresponding trend in faculty size, which has been growing at about a tenth the rate of student enrollments, as can be seen in Figure 1.2.

Figure 1.2

Enrollment Growth Trends in CS Majors and Faculty since 2006



Note: Reprinted from "Generation CS: The Growth of Computer Science," by Camp et al., 2017, *ACM Inroads, 8*(2), 46. Copyright 2017 by the Authors. Reprinted with permission.

This tremendous demand has resulted in many departments limiting enrollments for high-demand courses (~50%), restricting them to their own majors (45%), and encouraging underperforming students to leave the major (33%) (Camp et al., 2017c). Overburdened faculty are also reducing involvement in non-course related educational activities like club advising, undergraduate research support, reading groups, and mentoring, which may result in a less welcoming environment for students (Camp et al., 2017c). While enrollment growth has been accompanied by slight gains in the number of women and underrepresented minorities, there is "concern that the actions departments take to managed increased enrollments might have the side effect of reducing diversity" (Camp et al., 2017b). These trends are especially troubling at a time when both raw numbers and diversity are sorely needed in the field.

The COVID-19 Pandemic

While it is too early to say what long-term impact the global COVID-19 pandemic may ultimately have on Computer Science departments, there are a few observations that can be made, and perhaps some inferences drawn:

1. Postsecondary institutions are reporting a tremendous fiscal loss due to the necessary actions taken to protect their students, faculty, and staff.
2. The pandemic served to highlight both the need for, and relative security of, the kinds of jobs that represent Douglas Engelbart's "Intellectual Worker." As factory and retail workplaces shut doors and laid off employees, intellectual workers shifted to working from home, supported by technology infrastructure of the Internet. We will likely see a shift towards these kinds of jobs post-pandemic, further amplifying the need for postsecondary training in these skills.
3. The pandemic further demonstrated the need for additional software developers, both to support businesses leveraging Internet technologies to stay relevant in a shuttered world, and to upgrade and replace aging computing infrastructure strained by the additional demands brought on by the pandemic.

It is therefore likely that in the post-pandemic world, we will continue to see growing demand for postsecondary Computer Science instruction that Computer Science departments will be unable to meet merely by hiring more faculty. Thus, it will be even more important to be able to meet the demands created by growing enrollment numbers without significant increases in faculty numbers.

Theoretical Perspective

In educational research, it has become increasingly common for researchers to explicitly disclose their research paradigm, the combination of ontology, epistemology, and theoretical perspectives that are guiding their work. This section therefore seeks to briefly elucidate the influences guiding this study.

I have found myself drawn to William Doll's conception of a postmodern curriculum, in which he draws upon current cognitive science, Piaget's genetic epistemology, Dewey and Whitehead's educational philosophies, and Prigogine's work with self-organizing and dissipative structures. He posits that a postmodern curriculum should be *rich, recursive, relational,* and *rigorous* (Doll, 1993). To understand the value of this approach though, one must first understand the foundations upon which he is building, which arises from Piaget's work but incorporates new understandings emerging from current cognitive science research. Collectively this body of exploration is known as neo-Piagetian theory, and it has been applied to the domain of computer science in the *developmental epistemology of programming* theory developed by Raymond Lister and Donna Teague. Thus, this section will explore each of these foundations before circling back to Doll's conception of a post-modern curriculum.

Classical Piagetian Theory

There are two aspects to Piaget's work on learning: his stage theory of cognitive development, and his theories of the developmental processes that underlay learning (Teague, 2015). Piaget focused on learning in children. He also organized his observations on their cognitive growth into a series of age-related stages based on their ability to reason abstractly: sensorimotor, preoperational, concrete operational, and formal operational. His stage theory is well-known in American educational circles; research citing that only 30-35% of adolescents achieve formal operational stages of reasoning (and some adults never do) was once argued as a basis for testing students for abstract reasoning ability before admittance to a postsecondary computer science program (Lister, 2011).

Piaget’s insights into learning – which he called *genetic epistemology* – grew from his work as a biologist studying Limnaea, a species of snail found in Swiss lakes. He found the expressed phenotype of the snails’ shells was influenced by environmental conditions. A snail transplanted to a different lake started to grow its shell in a different shape; one which matched the Limnaea living in that environment. After he turned his attention to psychology, Piaget proposed the human brain likewise builds cognitive structures in response to its interactions with the environment (Doll, 1993).

For Piaget, these structures were the basis of learning, which occurred through two mechanisms: accommodation and assimilation (Piaget, 1970). Assimilation is the simpler of the two and involves adding new knowledge that fits within the learner’s existing cognitive structures. For example, once a learner had a structure for multiplication, adding additional multiplication “facts” would be an assimilation task. Similarly, once a learner has a structure for “mammal,” adding a new animal to that classification is straightforward.

In contrast, accommodation involves the creation of new cognitive structures to allow the learner to reason with new concepts. Accommodation is not an easy or automatic process – learners need to be repeatedly confronted with knowledge that does not conform to their existing cognitive structures, creating a sense of discomfort and disequilibrium between the learner’s conception of the world and their experiences of it (Piaget, 1970). With just the right amount of disturbance, the brain creates new structures to allow assimilation of new knowledge (Doll, 1993). This development of cognitive structure provides an explanation of the eureka moment when a concept a student has been struggling with suddenly becomes clear.

Neo-Piagetian Theory

Researchers have continued to expand upon and refine Piaget's work, incorporating new findings in cognitive science. This has led to some important refinements – one of the most critical was a movement away from a belief that Piaget's stages were caused by the biological maturation of the brain but are rather due to an increase in the effective capacity of working memory (Lister, 2011). Working memory has a small capacity, typically 7 ± 2 unique items, but by grouping disparate elements into sets in the process of 'chunking,' the brain can treat a structure as a single element in memory (National Research Council, 2000). With this understanding, Piaget's accommodation process can be seen as developing the neural structures to support chunking. Neo-Piagetian theory further suggests "that people, *regardless of their age,* are thought to progress through increasingly abstract forms of reasoning *as they gain expertise in a specific problem domain"* (Lister, 2011). Thus, in neo-Piagetian learning, the learner progresses through Piaget's stages with each unique knowledge domain.

Developmental Epistemology of Programming

Lister (2016) applied neo-Piagetian ideas to the domain of computer programming, laying the foundations of what he refers to as a *developmental epistemology of programmin*g. In this framework, he associates the four Piagetian stages with students' ability to trace code – "manually execute a piece of code and determine the values in the variables when the execution is finished" (Lister, 2016). Earlier research established that novices need to trace code with at least 50% accuracy before they begin to understand it (Clear et al., 2008; Lister et al., 2009). In contrast, experts often do not trace code at all – rather they infer the purpose of the code from its structural patterns (Lister et al., 2006). The neo-Piagetian stages of cognitive development identified in Lister's developmental epistemology of programming and their characteristics are detailed in Table 1.1.

Table 1.1

Neo-Piagetian Programming Stages

Developmental Stage	Characteristics
Sensorimotor (pre-tracing)	Student cannot trace code with >= 50% accuracy Looks for recognizable words in the code Dominant strategy is trial and error
Preoperational (tracing)	Student can trace code Often writes down the effect of each line or draws a diagram of program state Can describe what the code does, but the purpose of the code is not inferred, and may be guessed at based on input/output
Concrete Operational (abstract tracing)	Student can reason about code deductively and recognizes patterns in the code Dominant programming strategy is hasty design and futile patching First stage where student displays purposeful approach to writing code
Formal Operational (post-tracing)	Student can reason logically, consistently, and systematically about code, and can reflect on their own thinking Displays hypothetico-deductive reasoning (formulating and testing hypothesis)

These neo-Piagetian stages can be understood in the framework of the student developing cognitive structures to support 'chunking,' reducing the cognitive load of reasoning about programming. This is supported by studies of novice vs. expert programmers' mental representations of programming. Experts tend to develop structures based around function, while novices' structures primarily relate to syntax (Teague, 2015).

Thus, a sensorimotor programmer has no structures to speak of other than those drawn from other domains. This can help explain common novice misconceptions like assuming the computer can deduce the *intention* of the programmer, assuming the symbol 'while' indicates temporal consistency as it does in the English language, or assuming the symbol '=' indicates two values are equal as it does in mathematics (Teague, 2015).

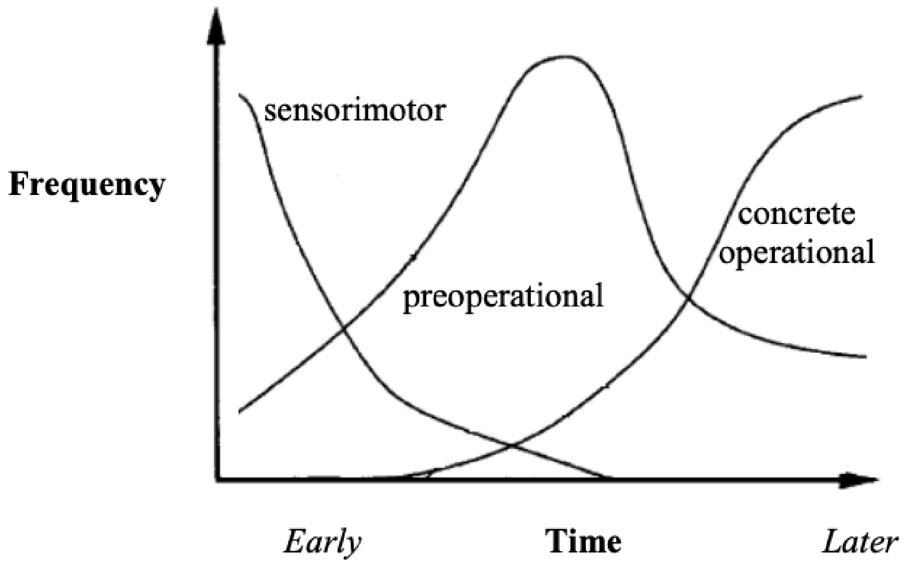
Preoperational programmers have built the cognitive framework to reason about program syntax, enabling them to trace code, but do not yet have the cognitive structures to see the 'big picture' of the program – concepts like invariants are still out of reach. "Until the concrete operational stage, novices are able to develop neither a mental representation of the code's function, nor the role of individual statements and relationships between them" (Teague, 2015, p. 49).

A concrete operational programmer has built the cognitive structures that help reason about programming abstractly and can begin to recognize strategies from patterns in code. A formal operational programmer has developed *many* cognitive structures about not just syntax and code structure, but also strategies for planning and debugging, and reflecting on their own efforts.

In a longitudinal study that formed part of her doctoral research, Teague (2015) found that students often concurrently manifested two or more developmental stages as they were learning programming. This is consistent with the neo-Piagetian overlapping waves model, visually depicted in Figure 1.3.

Figure 1.3

Neo-Piagetian Overlapping Waves Model



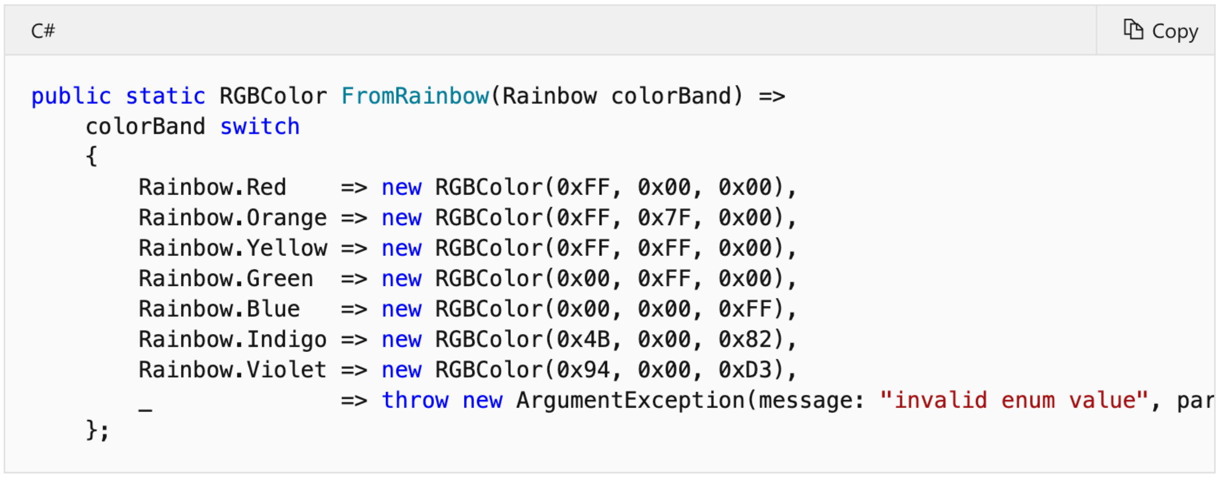
Note: Reprinted from "Toward a Developmental Epistemology of Computer Programming," by R. Lister, 2016, *WiPSCE '16: Proceedings of the 11th Workshop in Primary and Secondary Computing Education* (p. 7). Copyright 2016 by the ACM. Reprinted with permission.

Conceptually, it makes sense. Consider a student learning the syntax for an unfamiliar operation, like the pattern-matching switch expression, added to C# in version 8 (see Figure 1.4 for an example). Although the student has no cognitive structures specific to this new syntax (and thus operate at the sensorimotor stage in relation to it), the student will retain and be actively using the other chunks for C# programming they have developed (thus, operating at those stages in relationship to those portions of the code).

Moreover, if the student has already developed cognitive structures that are related to the new syntax (such as an understanding of the older case statement, the use of lambda expressions, or pattern matching operators from other languages), these can be incorporated into the new cognitive structures being built, allowing the student to progress more quickly through the stages. This is arguably why learning new syntax in a well-known language or learning a second programming language goes so much faster than the first. The learner possesses existing mental structures that can be adapted to incorporate the new knowledge.

Figure 1.4

Code Listing Demonstrating the C# Switch Expression



```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _              => throw new ArgumentException(message: "invalid enum value", par
    };
```

The Notional Machine

In thinking about these cognitive structures, it is useful to consider the concept of the *notional machine* – an idealized computer that combines aspects of the hardware and programing language to describe how programs are executed (Sorva, 2013). The notional machine is an abstraction; ideally it captures just enough detail to explain how a program is executed and inform the programmer's design decisions. A notional machine is specific to a language and execution environment (the physical hardware of the computer).

While notional machines can be expressed in formal terms, from the neo-Piagetian standpoint we can infer that the cognitive structures a student builds for reasoning about programming represent a mental model of a specific notional machine unique to that student. This "allows [the programmer] to make inferences about program behavior and to envision future changes to the programs they are writing" (Sorva, 2013, p. 9).

This mental model of a notional machine is undergoing constant revision as the student develops new understandings of how programs execute. Ben-Ari (2001) argues the computer represents an *accessible ontological reality*, in the sense that when students write and execute a program utilizing a mental model of a notional, they receive immediate evidence if their conception was correct – either the program behaves as expected, or some other form of error manifests.

Post-Modern Curriculum

With its foundations described, we return to William Doll's conception of a post-modern curriculum. In many ways, Doll's work is a refutation of Tyler's Rationale[2], suggesting it be replaced with curriculum that embodies Doll's four R's: *rich, recursive, relational,* and *rigorous* (Doll, 1993). Richness refers to the opportunity to play with and explore ideas through

[2] Tyler's Rationale is one of the best-known approaches to designing and organizing curriculum and involved "(a) the identification of purposes; (b) the organization of instruction attendant to such purposes; and (c) the design of evaluation mechanisms to determine if the stated purposes have been attained (Hlebowitsh, 2021)."

experiences. In learning to program I would argue this means the opportunity to read, write, and refactor (revise) many programs with varying goals, organization, and structure. Recursiveness refers to the opportunity to revisit and apply ideas in new contexts. It is a central organizing tenet in a spiral curriculum. For learning programming, I would suggest that this means concepts that are introduced to students need to be revisited and utilized in a variety of exercises and assignments (again in rich ways) and across courses. Relations involves building understandings that bridge concepts and domains; the student's mental model of the notional machine is a manifestation of the relations between code syntax and execution, type representation and operations, and much more. Additionally, the development of software involves solving problems from other domains, so using examples drawing from those domains (which may be less known to students) is also a way of building relations into the curriculum. Finally, rigor applies to the process of developing of the curriculum itself, with the recognition that significant thought and investigation needs to go into the design, there is no 'one way to get it right' and that it should be continually evolving (Doll, 1993).

Statement of the Problem

The growing recognition of the role computer science skills are playing in transforming how diverse fields like business, engineering, governance, medicine, politics, and science conduct their work have increased demand for Computer Science classes from majors outside the field. This increased demand is compounded by record growth in Computer Science major enrollments as well. But there has been limited growth in the resources and faculty available to meet this growing demand. Many of the pedagogical techniques we use are most effective for the small class sizes and a largely homogenous student bodies of the past.

To put it succinctly, we need to develop new approaches to teaching very large computer science classes with limited faculty. This need is echoed in the recommendations provided by the National Academies' report, *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments,* specifically recommendation 2.5:

> Institutions should pursue innovative strategies for using technology to deliver high-quality instruction at scale to large numbers of students, and pursue additional, creative strategies for meeting demand for quality computer science courses and skills development among the entire student body. (National Academies of Sciences, Engineering, and Medicine , 2018, p. 9)

This study proposes using code analysis techniques to assess student code algorithmically and use the results to drive the delivery of as-needed supplemental instruction. This would be supplied through the user interface of the student's development environment, allowing such guidance to be provided as the students write their code mimicking the kinds of guidance offered by instructors and teaching assistants in computer labs.

Purpose of this Study

This study can best be described as *educational exploratory research,* the purpose of which is to "investigate approaches to education problems to establish the basis for design and development of new interventions or strategies" (Institute of Education Sciences, 2013, p. 12). This differs from educational confirmation research, the most common form of which seeks to prove that an educational intervention leads to improved educational outcomes. The core difference lies in the nature of the theories involved. For confirmation research, well-established existing theories are drawn upon, whereas in experimental research, new theories are being developed. Confirmation research focuses on creating hypotheses from the theory that can be tested; in exploratory research, the theory itself is the hypothesis.

My working hypothesis for this study is that code analysis techniques could be used to perform real-time analysis on student code as it is written, and from this analysis recognize patterns in that code that indicate either a mistake or misconception on the part of the student, an assertion supported by several recent studies (De Ruvo et al., 2018; Ahmed, et al., 2018). From this recognition, an associated tutoring agent could provide a targeted instructional response – effectively explaining the mistake and why it *is* a mistake, at the point the student has made it.

Such a system could be used by students at any point and place they are working on their code, not just in a departmental computer lab when a tutor is normally available. Moreover, it could detect, and allow the student to correct, mistakes at the moment they are made, rather than at a future arbitrary point, which may well lead to improved educational outcomes. Thus, such a system could conceivably improve the efficacy of Computer Science instruction while reducing requirements for both lab space and instructional faculty time.

Moreover, this approach embraces the four 'Rs of a post-modern curriculum applied to programming. Such an approach supports *rich* experiences, as it can be used with any programming work, not just tightly constrained problems or a predefined set of assignments. It is also *recursive*, as the supplemental instruction is targeted towards the very concept students need to grasp at whatever point in their education they encounter it, and also at *every* point at which they encounter it. It is also *relational* if the instruction helps explain the concepts behind the error students are encountering, helping them to develop a correct mental model of the notional machine represented by the programming language and platform they are developing for. Finally, such a system is *rigorous* if it is constantly evaluated and updated.

But before any questions of efficacy can be addressed, more pressing questions must be answered, which this study seeks to do.

Research Questions

As expressed earlier, the primary goal of this study is to determine if algorithmically detected student mistakes could be used to provide supplemental, real-time tutoring from within the student's development environment. The study's research questions arise from this aim:

1. Can the proposed system for providing targeted contextual instruction be built with the software development tools currently available?
2. Can such a system effectively provide targeted contextual instruction corresponding to the difficulties the student is experiencing?
3. Can such a system be easily integrated into existing postsecondary programming instructional practice?
4. Will students perceive the system as useful for aiding in their learning process and achieving their learning goals?

Definitions

[Abstract] Syntax Tree (AST) is a representation of a program's syntactic structure as a tree and the result of a *lexical analysis*; it is a simplified form of a *parse tree* that has been pruned of unnecessary information (Aho & Ullman, 1977).

Backus-Naur Form (BNF) is a formal syntax for defining context-free grammars (Aho & Ullman, 1977). A parser can be algorithmically created from a BNF (Parr, 2013).

Compilation is the process of translating a program into a machine-executable form; it is usually a multiphase process with phases including *lexical analysis, syntax analysis,* and *semantic analysis* (Sunitha, 2020).

Compilation Error is an error in a program detected during the compilation process, during the phases of *lexical analysis, syntax analysis*, or *semantic analysis* and indicates an error in the structure of the program (Sunitha, 2020). This is in contrast to a run-time error, which only occurs during the execution of a program and indicates an error in the logic of the program.

Computational Thinking (CT) is a relatively new term in the literature that seeks to capture the kind of problem-solving process that would be employed by a computer scientist or intellectual worker, leveraging the tools and techniques made possible by ubiquitous computing. There are many competing definitions, but the one chosen for this study comes from Jeannette Wing, refined through discussions with Jan Cuny, and Larry Snyder: "**Computational Thinking** is the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent" (Wing, 2010).

Developmental Epistemology of Programming is a theory of how students learn to program derived from Jean Piaget's theories as elaborated by the neo-Piagetians and supported by significant qualitative investigation (Lister et al. 2006; Lister et al. 2009; Lister 2011; Lister, 2016; Teague, 2015.

Git is a version control system – a program that is used to 'commit' (save) changes to files within a directory and its subdirectories (collectively referred to as a 'repository') and allow those files to be reverted to a prior state. Git also offers the ability to create 'branches,' versions of the files that are edited independently of the original 'main' branch creating a new version, and to ‘merge’ versions represented by different branches back together. Finally, it allows the stored files and history of changes to be easily moved from computer to computer.

GitHub is a website that hosts Git repositories, serving as a central hub from which these repositories can be ‘cloned’ (copied) from, and synchronized with other cloned versions through ‘pushes’ and ‘pulls’ (essentially uploads and downloads that combine the local and remote versions of the repository).

Incremental Program Development is a programming practice where the programmer proceeds to develop a program iteratively towards a specified goal, refining both the understanding of the problem and the code intended to solve it in parallel (Wirth, 1983).

Integrated Development Environment (IDE) an application for authoring and editing code. An IDE offers functionality much like that of a regular text editor, but typically includes additional functionality helpful to programmers. Such additional features include syntax highlighting, autocompletion of common words and phrases, integration with a compiler or interpreter, and debugging and profiling tools.

Lexical Analysis is the first stage in the compilation process where source code is transformed into a list of tokens defined by the languages’ grammar; in this phase lexical errors (i.e. illegal composition of symbols) can be found and reported (Aho & Ullman, 1977). It is followed by *syntax analysis* and *semantic analysis.*

Notional Machine is an idealized computer that is an abstraction of a programming language and the hardware it runs on that describes how programs are executed; it can also refer to the mental model programmers develop to reason with as they learn to program (Sorva, 2013).

Parse Tree is a hierarchical representation of a program's syntax generated from the *semantic analysis* stage of compilation; an abstract syntax tree is a more refined form of a parse tree that omits unnecessary and redundant information (Aho & Ullman, 1977).

Semantic Analysis is the third stage in the compilation process where the program syntax (in the form of a parse tree or syntax tree) is evaluated to determine if it is meaningful or not; primarily this involves *type checking* (Sunitha, 2020). It is proceeded by *lexical analysis* and *syntax analysis.*

Semantic Style Indicators are semantical code patterns that, while they will compile and accomplish the intended goal, suggest the programmer does not fully comprehend the programing language (De Ruvo et al., 2018).

Syntax Analysis is the second stage in the compilation process where the token list is transformed into a *parse* or *syntax tree*; it is in this phase that errors in syntax construction are found (Aho & Ullman, 1977). It is proceeded by *lexical analysis* and followed by *semantic analysis.*

[Static] Type Checking is the process of determining if a program operation can be performed on the supplied data type; it is static if this check occurs during the compilation process, which can only be done with certain types of programming languages (Aho & Ullman, 1977).

Visual Studio is an Integrated Development Environment (IDE) developed by Microsoft for its flagship .NET programming languages, including C#. It combines a code editor (a text editor for writing program source code), a compiler, and a debugger, and may other tools for supporting a professional software developer. It is also *extensible*, allowing additional functionality to be added to the program by installing extensions – additional programs that add new features to the IDE.

Chapter 2 - Review of the Literature

The goal of this study was to investigate if algorithmic detection of code patterns could be used to detect student mistakes in authoring programs that could be indicative of a lack of understanding, and then use the output from this process to provide targeted supplemental instruction within the development environment. Taken separately, each of these concepts have a long academic history. However, they have not been brought together to the degree I am attempting, though several recent papers have proposed doing so (De Ruvo et al., 2018; Ahmed et al., 2018).

This chapter is therefore divided into sections corresponding to each research strand. First, I briefly discuss higher order programming languages and how they are compiled into a form that the computer can execute, as grasping the basics of this process is important to understanding code analysis. The second section discusses how code analysis is used in professional tools, and the third how it has been used pedagogically. Then I discuss other assessment strategies currently employed in the discipline of computer science.

Higher Order Programming Languages

The use of programs to analyze other programs is almost as old as the field of computer science itself. Higher order programming languages (like C#) rely on another program (a compiler or interpreter) to convert a program's code into instructions a computer can carry out. Grace Hopper released FLOW-MATIC in 1954, only nine years after the construction of the first American general-purposed digital computer, the ENIAC (Gürer, 2002). Since that time, the development of techniques for developing compilers and interpreters have been a major focus of research.

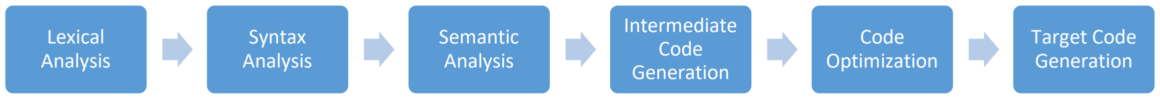
A detailed review of this body of knowledge is well out of the scope of this project, but there are several important concepts emerging from the literature that do warrant some expounding to better explain the thrust of my research. The next section will briefly explore the concepts of the compilation process that bear on the rest of our discussion.

The Compilation Process

A compiler translates program code written in a higher order programming language into instructions a computer can execute. This process is complex and normally broken into a series of phases to manage that complexity. Further, each phase may be executed multiple times (passes) on separate portions of the code and combined into an intermediate or final form (Aho & Ullman, 1977). The techniques used in each of these phases have continued to evolve, but the phases themselves have remained remarkably consistent (Figure 2.1.)

Figure 2.1

Compilation Process Phases



The first phase, *lexical analysis*, breaks the source code into logical groupings known as tokens (Sunitha, 2020). These tokens typically represent keywords in the language, symbols defined by the programmer, or raw data the program uses. The structure of allowable tokens is defined by the grammar of the programming language, typically embodied as a context-free grammar and often expressed in Backus-Naur Form (Aho & Ullman, 1977).

The second stage, *syntax analysis*, generates a tree structure representing the syntactic structure of the program from the tokens identified in the lexical analysis (Sunitha, 2020). This tree may be a *parse tree*, which is an exact representation of the code as written by the programmer, or a more concise representation that eliminates redundant and unnecessary information known as a *syntax tree* or *abstract syntax tree* (Aho & Ullman, 1977).

The third stage, *semantic analysis*, determines if the program constructs are meaningful, in the sense that they can be carried out as part of the program execution. In most languages, this consists primarily of *static type checking* – making sure that the operation specified can be carried out on the data type it is being asked to execute on (Sunitha, 2020) as well as scope and declaration issues (Aho & Ullman, 1977). For example, it does not make sense to compute the absolute value of a string, i.e.:

$$Math.Abs(\text{"The Taj Mahal"});$$

Although the above line of code is grammatically correct C#, it represents a semantic mistake by the programmer as they are trying to carry out an operation that makes no sense. It should be noted that static type checking can only be carried out in a *strongly typed* language; that is, a language where “the type of every name [variable] and expression can be computed at compile-time” (Aho & Ullman, 1977, p. 387).

The remaining stages, while important to converting the program source code into a machine-executable form, do not expose opportunities for identifying problems with program structure, and as such are not important for understanding this proposal. However, the first three stages offer the opportunity to identify a number of issues in code that are usually reported by the compiler in the form of *compilation errors*. These are errors that are reported when the program code is compiled – and in most cases, halt the compilation process (as there is no point to continuing while the program contains errors). Table 2.1 provides a listing of common compilation errors organized by the compilation phase in which they are detected.

Table 2.1

Compilation Phase of Identification and Common Errors

Compilation Phase	Possible Errors
Lexical Analysis	Numeric literals that are too long Ill-formed literals and symbols Reserved keywords used as variable names Unrecognized groups of characters
Syntax Analysis	Missing operator/operands in expressions Missing symbol definitions (i.e., undeclared variables or functions) Bracket mismatch issues
Semantic Analysis	Incompatible types of operands

Although compilers report the compilation errors they detect to the programmer, the messages report the error *from the standpoint of compilation* and are written with the expectation the audience is an experienced programmer. For novice programmers these error messages are cryptic and difficult to decipher, requiring a process of

> firstly, understanding what the compiler says through error/warning messages; secondly guessing what these messages really mean; thirdly, figuring out to do to fix such a warning; fourthly, learning how to act and avoid them thereafter; and fifthly, recognizing recurring messages and remembering how they were fixed in the past (Traver, 2010, p. 5)

From the neo-Piagetian standpoint, decoding the compilation error messages therefore requires the student to have developed a notional machine that includes the generation of error messages and what context(s) can cause them. Thus, for deciphering compiler errors, the student must progress through the four cognitive stages, starting afresh for each new message *and* for messages that reappear in a different context. For example, the ubiquitous ‘stack overflow’ error occurs when a program runs out of available memory. Students typically first encounter this error when they accidentally create a loop that does not terminate, and later when they create a recursive method that calls itself infinitely many times. Novice students who have learned that ‘stack overflow’ means they have created an infinite loop, now must extend that understanding to infinite recursion. Only after the students have encountered the concept of the program stack, well after their introductory programming courses, do they see the source of the cryptic name ‘stack overflow.’

Professional Code Analysis Tools

Compilers are not the only kinds of programs that have been developed to analyze source code and identify mistakes and other issues. Software development is a large and thriving professional domain and numerous tools have been developed to support programmers at their work. This section focuses on a subset of those tools that utilize code analysis techniques.

Linters

Linters, named after the Unix tool *lint* (one of the earliest and best known of this class of program) are programs that analyze source code for stylistic issues and reinforce commonly accepted best practices. The purpose of a linter message therefore differs from those of a compiler. The compiler messages are focused on making sure the code can execute, while a linter is focused on making sure the code is well-written.

Poorly written code can have serious impact on the maintainability, performance, and security of a program (Keuning et al., 2019). Much as human languages can express the same concepts in a multitude of ways, programming languages can be used to achieve the same results in very different approaches. For example, I once had student teams write code to generate and serve a webpage displaying a list of participants in a competition by their score. For most teams the page was generated in under a second, but one team's code took over an hour to execute. Programs are also rarely written by a single programmer, rather they are typically developed and maintained by a team. Much as a well-written essay is easier for an audience to read and more effective at conveying that meaning to other people, well-written code is easier for other programmers to understand and therefore maintain.

From the neo-Piagetian perspective, we can derive an even stronger argument for why poor code quality is problematic. The cognitive structures that we develop to support chunking are related to the brain's natural pattern-recognition ability. Remember that expert programmers look at the code *as a whole* and recognize its meaning by recognizing common structural patterns (Teague, 2015). These patterns are obscured by poor structuring (inconsistent indentation, bad sequencing of operations, lack of naming conventions, and over- or under-abstraction) making it harder for an expert programmer to understand intuitively. Moreover, as students write their own code, *they are developing their own cognitive structures*. If these structures use patterns that do not match the conventions adopted by other programmers, the students will likewise have trouble understanding other programmers' work.

While some educators have adopted the use of linters in their courses, the linters adopted are targeted at professional programmers and suffer from the same limitations as the compiler warning and error messages mentioned previously (Keuning et al., 2019). These linters provide no context for *why* the suggestions offered are an improvement, and ultimately may only add another layer of ‘noise’ students must contend with in their learning process.

ANTLR and Other Parsers

One benefit of defining a formal grammar for a language is that a parser can be algorithmically generated from it. Such a parser and its output has far more uses than just an initial stage for compilation. It can be used to convert data files into a useable form, support the development of domain-specific languages, identify security issues in code, and even be used to verify program correctness. ANTLR, an open-source recursive-descent parser written in Java, is perhaps the best known of these parsing tools, and sees much use in industry and academia (Parr, 2013).

The Roslyn Compiler and API

The specific compiler I used in this study, Microsoft’s *Roslyn*, combines the traditional aspects of a compiler and a linter in one open-source platform. It implements the compiler pipeline described at the beginning chapter 2, and for each phase “exposes an object model that allows access to the information at that phase …the parsing phase exposes a syntax tree, the declaration phase exposes a hierarchical symbol table, the binding phase exposes the result of the compiler’s semantic analysis, and the emit phase is an API that produces IL byte code” (Wagner et al., 2020).

In working with the Roslyn compiler, a developer can interact with the early stages of compilation. It exposes all the data generated during each stage of the compilation process in the form of immutable data structures through its Compiler API, also builds upon this layer an extensible Diagnostics API (Wagner et al., 2020). This second API allows for the easy creation of *analyzers* and *code fixes*. Analyzers leverage the compiler data to identify grammar, syntax, and semantic errors, and code fixes provide potential solutions to the problems identified (Wenzel, et al., 2020). Microsoft uses this platform to implement compilation error and linting-style suggestions in their Visual Studio and VS Code integrated development environments. Third-party linters like ReSharper (referenced in Keuning et al., 2019) also are built upon this platform.

Integrated Development Environments

[Integrated] development environments (IDEs) are programs that are used to write programs. Essentially an IDE is a text editor that is built to edit code, but usually also integrates a compiler to build and execute that code, along with instrumentation to support programmers in the debugging process. As with compilers and linters, the vast majority of IDEs are targeted at professional programmers and are feature rich. They "have a high learning curve and are confusing to novice programmers due to complex interfaces and a multitude of tools" (Vogts et al., 2008).

The Visual Studio and VS Code applications mentioned above are two professional IDEs developed by Microsoft to compile programs, including those written in C#. Visual Studio is a computational brute composed of multiple concurrently executing processes. This includes running a lexical, syntax, and semantic analysis every time a programmer makes a change, allowing Visual Studio to detect and report programmer mistakes almost as soon as those mistakes are made. VS Code is a lighter weight cousin built on using the web technologies of HTML/CSS/JavaScript. Both of these use the Roslyn compiler and can be extended with custom-built 'plug-in' programs. It is through this mechanism that I proposed to develop the pedagogic tools to detect student mistakes and offer pedagogical support in correcting them.

Pedagogical Code Analysis Tools

Although code analysis has long been a mainstay of professional development tools, the development and adoption of pedagogical code analysis tools have had a complicated history. A great many tools are developed, make an appearance in one conference publication, and then quietly fade away. Nicklas Wirth identifies several factors he sees as contributing to this trend:

1. Academics are not judged by the quality of their teaching, but rather by their research funding and publications, making the development of new pedagogical tools to improve their teaching low priority.
2. Universities have adopted a business model of education and emphasize teaching what their customers (students and industry) see as valuable, hence *professional* tools.
3. Faculty have abdicated their responsibility to lead and surrendered to industry trends (Wirth, 2002).

That said, there have been a few impactful pedagogical tools developed, and many more that have at least contributed some findings before fading into obscurity. I will examine those most related to my efforts in this next section.

Pedagogic Programming Languages

Pedagogic programing languages are those that are created primarily with the purpose of being used for teaching and learning. The most impactful of these at the postsecondary level was the language Pascal, developed by Nicolas Wirth and seeing widespread release in 1974 (Wirth, 1985). For a significant period, it was an overwhelmingly popular choice for a first course in programming and even made inroads into industry but was gradually replaced by professional programming languages (Dingle & Zander, 2001). While Wirth followed up with new pedagogical languages that sought to incorporate concurrency and other advanced features these new languages did not see the same widespread adoption as Pascal. Frustrated, he turned his attention to hardware design (Wirth, 1985; Wirth, 2002).

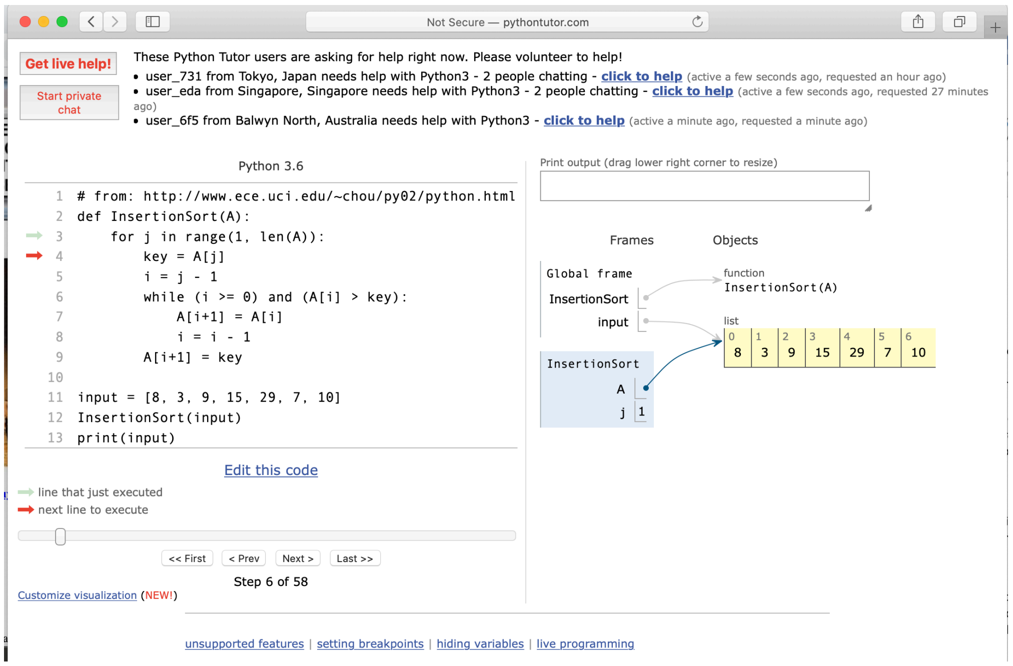
Aside from Pascal the only significant pedagogical languages that remain in common use are primarily intended for elementary students. These include the popular drag-and-drop block-based Scratch (Resnick, et al., 2009), the visual programming environment AgentCubes and its ancestor AgentSheets (Repenning et al., 2012), and some descendants of the LOGO language popularized by Seymour Papert (Papert, 1980). Developed by MIT Media Labs' Lifelong Kindergarten Group with the goal of making programming accessible to anyone, Scratch is the most successful of these with over 550,000 active users, with over 93,000 as young as four and more than 5,500 over the age of 80 (Lifelong Kindergarten Group, 2020). Scratch does see some use at the postsecondary level "to provide a 'syntax-light' introduction to programming; …often, (although not exclusively) used with non-majors or at the start of an introductory course" (Joint Task Force on Computing Curricula, Association for Computing Machinery & IEEE Computer Society, 2013, p. 43).

Python Tutor

Python Tutor is an online development environment that focuses on helping students develop their understanding of how memory works within a program. It allows for stepwise execution of Python code while presenting the user with a visualization of the memory state of the program at each step (Guo, 2013). A screenshot of the Python Tutor executing a Python program can be seen in Figure 2.2. The Python Tutor interface has since been expanded to include additional languages, currently Java, C, C++, JavaScript, and Ruby. A student team from Kansas State University's Department of Computer Science has also adapted it for C#.

Figure 2.2

Python Tutor Interface



These Python Tutor users are asking for help right now. Please volunteer to help!

- user_731 from Tokyo, Japan needs help with Python3 - 2 people chatting - **click to help** (active a few seconds ago, requested an hour ago)
- user_eda from Singapore, Singapore needs help with Python3 - 2 people chatting - **click to help** (active a few seconds ago, requested 27 minutes ago)
- user_6f5 from Balwyn North, Australia needs help with Python3 - **click to help** (active a minute ago, requested a minute ago)

Python 3.6

```
# from: http://www.ece.uci.edu/~chou/py02/python.html
def InsertionSort(A):
    for j in range(1, len(A)):
        key = A[j]
        i = j - 1
        while (i >= 0) and (A[i] > key):
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = key

input = [8, 3, 9, 15, 29, 7, 10]
InsertionSort(input)
print(input)
```

Note. The interactive Python Tutor can be found at http://pythontutor.com

The pedagogical use of visualizations of algorithms and memory state commonly report mixed results in the literature. Some studies suggest they help, others that they hinder. Viewed through the lens of developmental epistemology, these mixed results make perfect sense. For a sensorimotor student, visualizations are simply more seemingly unrelated information that must be managed by an overburdened mind. Likewise, a preoperational student is "able to develop neither a mental representation of the code's function, nor the role of individual statements and the relationships between them" (Teague, 2015, p. 49). Thus, visualizations representing the overall function of an algorithm are of limited use. However, preoperational students *do* utilize diagrams as a supportive tool for code tracing, and predominantly use trial-and-error as a programming strategy (Lister, 2011).

Thus, the approach adopted by Python Tutor, which enhances code tracing by providing visualization of memory state consistent with a valid notional machine is likely helpful to students at this stage and may even be helpful in developing the cognitive structures necessary to achieve the next stage. Studies of a similar desktop development environment, Jeliot 3, have shown to be effective at helping students develop a more accurate notional model for Java, though the study size was small (N=34), and the authors cited concerns about the minute details of execution involved (Ma et al., 2009).

BlueJ

BlueJ is perhaps the best-known pedagogical development environment, currently in Version 5 and in widespread use for over 20 years (Brown, 2020). BlueJ takes several steps to simplify the cognitive challenges of learning Java and object-oriented programming for a novice programmer. It presents a minimal and clean interface, instead of the file tree common in professional development environments. It organizes files into a class diagram. It allows users to instantiate and manipulate objects directly without needing to execute the program and provides debugging functionality tailored to the needs of novice programmers (Kölling et al., 2003).

One of the most interesting research-focused aspects of BlueJ is its sister project Blackbox. Launched in 2013, Blackbox is a concerted data collection effort that collects student source code at every time it is compiled and when certain other events are triggered within the

BlueJ interface (Brown et al., 2014). This collection has provided a rich data source for researchers investigating how students learn programming[3].

In version 4, BlueJ adopted a continuous compilation model similar to the one described for Visual Studio, where the program code is compiled in a background process every time a student makes a change. This allows the development environment to report errors as they occur (Karvelas et al., 2020). A study of the impacts of this change found:

> (1) Users see more compiler error messages in BlueJ 4, likely because they *choose* to – a choice afforded by a more flexible presentation mechanism. They also show that (2) BlueJ 4 users compile less frequently, but (3) have more successful manual compilations, possibly indicating that these are done as reassurance that their code is correct when there is no error highlighting present. In addition, (4) the heuristics that were used for session time calculations showed consistency across the results, and thus support these results further. (Karvelas et al., 2020)

These results suggest that a continuous compilation mechanism is effective at helping students avoid compilation errors. This more closely parallels the process of *incremental program development*, a desirable skill for programmers where the program is built in incremental steps and its functionality tested at each. In contrast, novice programmers tend to try to write the complete program before compiling and testing (Vogts et al., 2008).

However, BlueJ only provides the error messages generated by the compiler, which suffer the same issues discussed in the compiler section. The Expresso project, implemented as a

[3] Examples of research emerging from the BlackBox data set in this literature review include (De Ruvo et al., 2018) and (Karvelas et al., 2020).

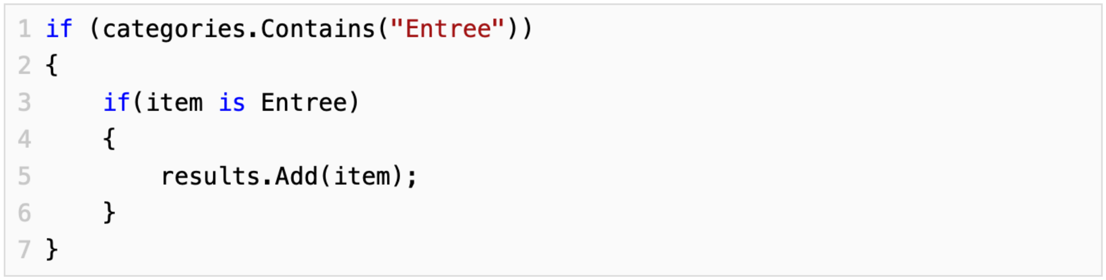
multiple-pass preprocessor for Java (essentially, a program that does the first three phases of compilation), sought to address this issue by providing more detailed error messages written from a pedagogical standpoint (Hristova et al., 2003). Unfortunately, the promised follow-on study of its effectiveness never materialized, and the program does not appear to be in use today.

Semantic Style Indicators

One of the projects supported by the Blackbox data focused on identifying *semantic style indicators*, “surface indications that suggest a programmer lacks some knowledge about one or more aspects of programming” (De Ruvo et al., 2018). This study was able to identify 16 of these semantic style indicators that could be identified from an abstract syntax tree derived from student code and represent not so much mistakes as a lack of refinement (De Ruvo et al., 2018). For example, their *unnecessary nesting if* semantic style indicator appeared in the work of one of my students and is reproduced in Figure 2.3; the preferred more concise version combining the two tests appears in Figure 2.4.

Figure 2.3

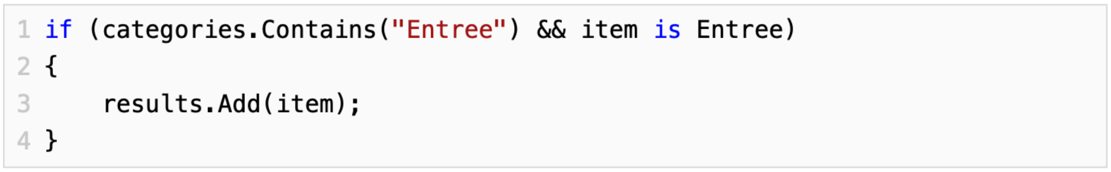
Student Code with Unnecessary Nesting IF Semantic Style Indicator



```
if (categories.Contains("Entree"))
{
    if(item is Entree)
    {
        results.Add(item);
    }
}
```

Figure 2.4

Preferred Form of the Logic from Figure 2.3



```
if (categories.Contains("Entree") && item is Entree)
{
    results.Add(item);
}
```

It is important to understand that *both* code snippets compile and accomplish the same goal. Moreover, they will likely have similar performance. Thus, the code in Figure 2.3 is not a mistake so much as it is inelegant. Accordingly, such issues are unlikely to be remarked upon in student homework, especially with large class sizes or when being marked by a teaching assistant. In the study these indicators were found in data collected from courses at every point in the student's undergraduate career, with some appearing even in fourth-year student code (De Ruvo et al., 2018).

The authors considered these inelegancies an issue as "particular semantic style indicators may be manifestations of poor knowledge of some programming concepts" (De Ruvo et al., 2018, p. 73). Considered from the framework of developmental epistemology, this suggests that some students are not reaching the formal operational stage in relation to the specific concepts associated with those semantic style indicators. Until they achieve the concrete operational stage, programmers cannot see the relationship between pieces of code (i.e., that the two tests can be combined) and in the concrete operational students are limited to working with abstractions that are familiar (Teague, 2015). It is only in the formal operational stage where the cognitive structures supporting programming are well-established that students can spare the mental resources to reason abstractly about the style of the code and recognize why the second form might be preferred.

Thus, the presence of such a semantic indicator in student code is suggestive that the student needs to develop additional cognitive structures around the concept in question. As the study authors note "such a lack of knowledge may not be only related to novice programmers, but to anyone unfamiliar with the language" (De Ruvo et al., 2018). Considering the overlapping waves model, we can extend that argument to *specific features* of the language. An otherwise expert programmer may display semantic style indicators tied to a newly added syntax feature that has not yet been incorporated into their notional machine mental model for the language. Thus, semantic style indicators could be leveraged for targeted precision feedback, as the authors note “the challenge is to provide the feedback as soon as possible” (De Ruvo et al., 2018, p. 80).

The system I proposed does just that – providing targeted feedback as soon as the student has written code containing the semantic style indicator. Although the study focused on the Java programming language, it is reasonable to assume that similar semantic style indicators can be derived for additional languages. In fact, Java and C# share enough common structure and syntax that many of those semantic style indicators identified in De Ruvo et al. (2018) exist in both languages, as was shown in the example of the *unnecessary nesting if*. Thus, my approach both expands upon and provides a necessary next step in this research.

TRACER

A much more ambitious project in a similar vein of detecting issues in student code and providing targeted feedback is TRACER (Targeted RepAir of Compilation Errors), whose design is “based on a realization that the goal of program repair in pedagogical settings is not to simply eliminate compilation errors using any means possible (such as deleting the error line altogether), but rather to reveal the underlying errors to the students, so that they may learn how to correct similar errors themselves in the future” (Ahmed et al., 2018, p. 79).

TRACER essentially carries out the same kind of activity as Visual Studio’s Code Fixes, but instead of relying on a code analysis algorithm developed by a programmer, it leverages machine learning to determine how to best fix an error. Currently the system is effective in repairs depending on the local context but does not take into consideration the global context.

And although it was developed with a pedagogical intent the supporting pedagogical structure has not yet been created (Ahmed et al., 2018).

However, the techniques employed could be leveraged to provide code analysis for the kind of intelligent tutoring system I have proposed, especially where a large data set of student work could be employed to train the machine learning algorithms. Thus, a TRACER-style approach could be incorporated into the kind of feedback system I developed. This could be done off-line (where the machine learning would be used to generate the code analysis algorithms) or possibly in real-time (where the machine learning is used as the basis for an intelligent tutoring agent). While the latter is certainly exciting, it will be difficult to and time-consuming to develop a system that would do this well. In the meantime, it would behoove us to build evidence to support that timely supplemental instruction provided from a more straightforward approach has a positive impact on student learning.

Assessment Techniques

In terms of pedagogy, the detection of semantic style indicators in student code is a form of assessment. There are two major forms assessment takes in education: *formative assessment* is used "as sources of feedback to improve teaching and learning" (National Research Council, 2000, p. 140) and *summative assessment* "measures what students have learned at the end of learning activities" (National Research Council, 2000, p. 140). Thus, the approach used in this study of using code analysis to drive targeted and timely instruction is formative in nature, as the assessment results are used to supply supplemental instruction. However, the same techniques could also be used to drive summative assessments.

Understanding this study's approach as constituting (in part) an assessment suggests we should also examine the current state of assessment practice within computer science education.

Thus, this final section of the literature review will focus on other assessment techniques that have been employed within the field.

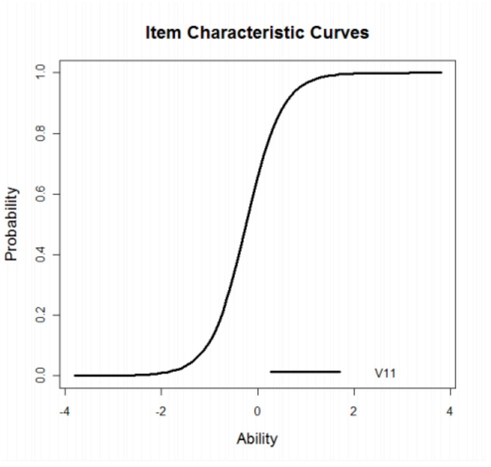
Item Response Theory

Item Response Theory (IRT) is the basis of most current standardized assessment tests. In many ways, this was a more evolved form of *classical test theory* (CTT). CTT assumes that a simple number, a *true score plus a smaller error score*, represented the depth of a student's knowledge and reasoning ability within a domain. IRT abandoned this single-score model in favor of "a mathematical model for the probability that a given person will respond correctly to a given item, a function of that person's proficiency parameter and one or more parameters for the item … such as difficulty or sensitivity to proficiency (Mislevy, 1993)." In IRT the item, rather than an overall score, is the unit of observation, and sophisticated statistical techniques have been developed to ensure the reliability and validity of IRT-based assessments.

IRT assumes each question has a "characteristic curve" plotting the probability of a student getting a question right given the level of proficiency in the subject the question addresses. This curve is defined by two parameters, α and β. β corresponds to the difficulty of the question, and α represents the likelihood that a student with proficiency greater or less than β would answer the question correctly (i.e., how discriminating the question is at determining a student's proficiency). Winters and Payne (2006) point out that calculating α and β for computer science assessments can allow educators to evaluate their instruments and develop better ones, helping ensure accurate and meaningful scores. Sudol and Studer (2010) offer a detailed introduction to IRT for computer science practitioners, including instructions for carrying out such an evaluation, as well as examples of characteristic curves from a real-world data set. One of these is reprinted in Figure 2.5 below.

Figure 2.5

Item Characteristic Curve Example



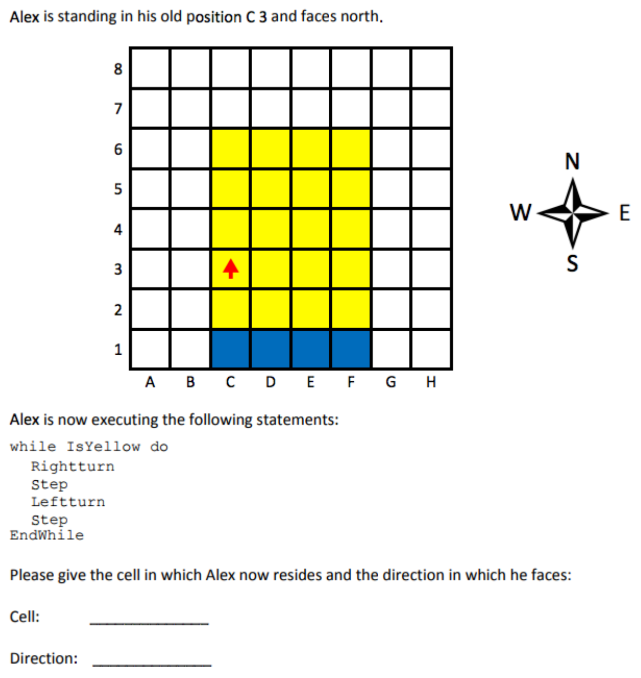
Note: Characteristic curve for an exam question. Reprinted from “Analyzing test items: using item response theory to validate assessments,” by L. A. Sudol and C. Studer, 2010, *SIGCSE ’10: Proceeding sof the 41st ACM Technical Symposium on Computer Science* (p. 438). Copyright 2010 by the ACM. Reprinted with permission.

Instruments designed for IRT typically take the form of multiple-choice tests, which are criticized as limited in their usefulness for measuring complex abilities such as writing or computer programming. To put it succinctly, to assess programming skills we need to assess student *competence* rather than *depth of knowledge*, which is difficult to do with the multiple-choice assessments favored by the IRT approach. A handful of efforts have sought to move beyond multiple-choice assessments with approaches more appropriate to programming, focused on either reading and interpreting code examples, or on analyzing student code directly.

One of these approaches adopted by Mühling et al. (2015) was to present example code that manipulated the position of a ‘robot’ in a grid-based world and ask the student to identify the outcome. This essentially assesses the students' ability to trace code, an important marker for a student's development in the framework of a developmental epistemology of programming (discussed in chapter 1). An example question of this type appears in Figure 2.6.

Figure 2.6

Question from Mühling et al.'s IRT-Based Assessment


Alex is standing in his old position C 3 and faces north.

Alex is now executing the following statements:

```
while IsYellow do
  Rightturn
  Step
  Leftturn
  Step
EndWhile
```

Please give the cell in which Alex now resides and the direction in which he faces:

Cell: ______________

Direction: ______________

Note: Reprinted from "Design and First Results of a Psychometric Test for Measuring Basic Programming Abilities," by A. Mühling et al., 2015, *WiPSCE '15: Proceedings of the Workshop in Primary and Secondary Computing Education* (p. 4). Copyright 2015 by the ACM. Reprinted with permission.

A contrasting approach adopted by Berges and Hubwieser (2015) allows the direct analysis of student-written source code for assessing understanding of object orientation. Items took the form of identifying code constructs expected to be present in student code, providing a dichotomous score. Students were challenged to complete a program that would require all the expected items to work correctly. Then the code was analyzed for the corresponding patterns; if found the student earned a 1 on the item, if not, a 0.

Computational Thinking Patterns

In developing assessments for their Scalable Game Design curriculum, the University of Colorado-Boulder developed the concept of *conceptual thinking patterns,* "which are abstracted programming patterns that are learned by students when they create games and can readily be used by students to model scientific phenomena (Basawapatna et al., 2011, p. 246)." In other words, these are design patterns specific to game programming and simulation. Further, the patterns identified in this study are derived from (and apply specifically to) to the Scalable Game Design group's flagship programming language and integrated development environment, AgentSheets, which can be used to create simulations and video games (Basawapatna et al., 2015).

The Scalable Game Design research group has studied how computational thinking skills taught through game design might transfer to other domains (Basawapatna et al., 2010). The conceptual model of computational thinking patterns provides a bridge for both teaching and assessing transfer across these domains. The eight computational thinking patterns identified by the Scalable Game Design group are listed in Table 2.2.

Table 2.2

Computational Thinking Patterns with Examples

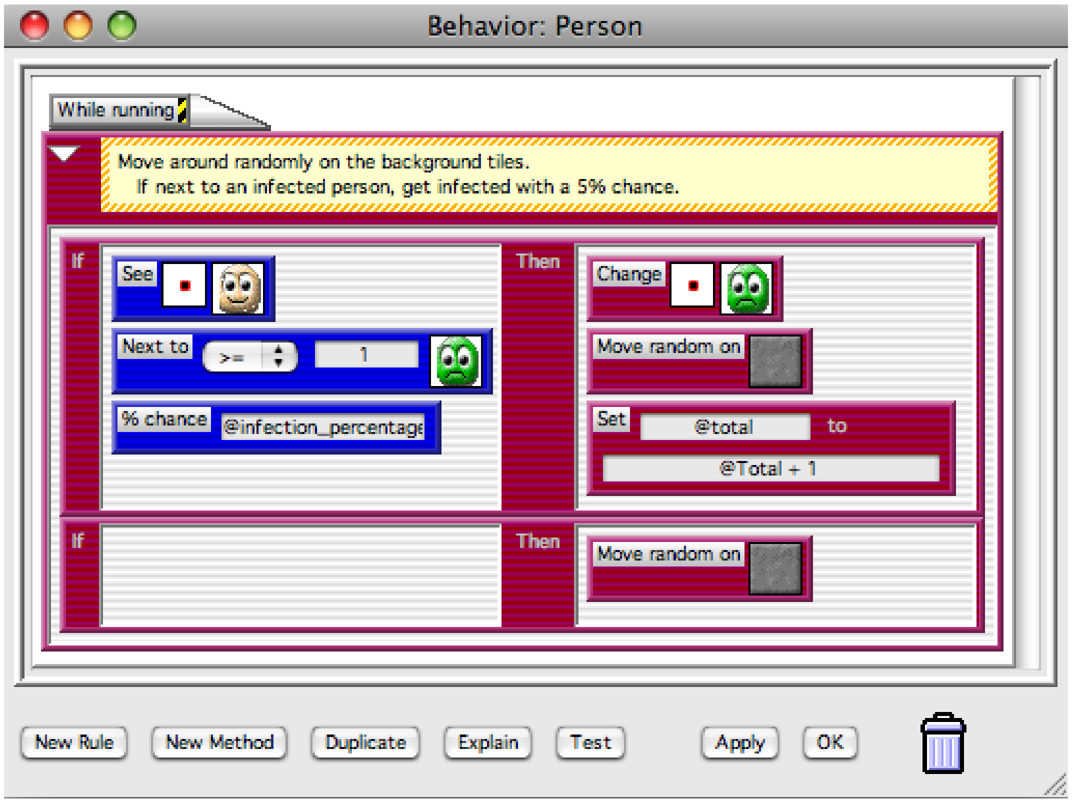
Computational Thinking Pattern	Examples
Generation	An agent creates other agents, such as spawning new characters on-screen or an organism creating offspring
Absorption	An agent consuming another, i.e., Pac-Man eating a dot
Collision	Two agents bumping into one another
Transportation	An agent carrying another agent, such as a game character riding a moving platform
Push	An agent pushing another agent, i.e., pushing blocks to form an impromptu staircase
Pull	An agent pulling one or more agents, i.e., a locomotive pulling train cars
Diffusion	Value diffusing from an agent into neighboring agents, i.e., wood placed near a fire catching fire
Hill Climbing	A search algorithm where the search environment is divided spatially, and the agent moves to the space nearest itself with the highest value – the "hill." For example, a simulated ant might view the world as grid cells, and move into the neighboring cell with the highest "food" value, then from that cell move to a new neighboring cell with an even higher one.
Cursor Control	User input in the form of a cursor directly controlling an agent.

Note. The first eight patterns are detailed in (Basawapatna et al., 2010), and the additional ninth, cursor control, is identified in (Koh et al., 2010).

AgentSheets programs consist of patterns of If/Then conditional statements that are repeatedly evaluated; when the test condition specified in the *if* section evaluates to true, the actions specified in the *then* section are executed. A screenshot of the AgentSheets programming environment for creating these rules appears in Figure 2.7.

Figure 2.7

The AgentSheets Programming Environment



Move around randomly on the background tiles.
If next to an infected person, get infected with a 5% chance.

```
If                                   Then
See                                  Change
Next to  >=  1                       Move random on
% chance  @infection_percentage      Set  @total  to
                                          @Total + 1
If                                   Then
                                     Move random on
```

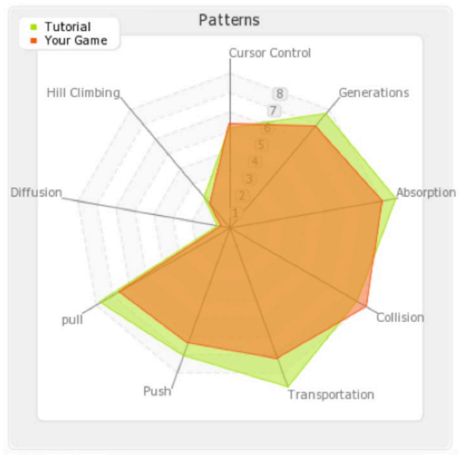
These rules echo the structure of propositional logic; the AgentSheets language can be considered to fall in the *logical paradigm* of programming languages. The structure of the AgentSheets language is limited in scope – there are 23 possible conditions that can be selected for the *if* section of a rule, and 16 possible actions that can be selected for the *then* section. This limited grammar allows for student work be compared to the expected solution through a *program behavior similarity* calculation:

> With the 23 conditions and 16 actions, it is possible to represent each game as a vector of length 39, wherein each element of the vector represents how many of each individual conditions and actions are used to implement a given game. Using these vectors, any game created in AgentSheets can be compared to any other game through a high dimensional cosine calculation for similarity. (Koh et al., 2010)

The Scalable Game Design research group has utilized this calculation in creating an assessment tool that produces a computational thinking pattern graph, and example of which appears in Figure 2.8.

Figure 2.8

An example of a computational thinking pattern graph



Note: Reprinted from “Recognizing computational thinking patterns,” by A. Basawapatna et al., 2011, *SIGCSE '11: Proceedings of the 42nd ACM technical symposium on Computer science education*, p. 246. Copyright 2011 by the ACM. Reprinted with permission.

This radial graph arranges the nine computational patterns as axes radiating out from a central point; a student's score with each pattern is marked along each axis, along with the score of a solution program, allowing easy visual comparison of the two programs (Basawapatna et al. 2011). This program behavior similarity calculation can be carried out algorithmically. The first use of computational thinking graphs was to automatically generate feedback for AgentSheets users.

Summary

In this chapter, I briefly introduced how higher order programming languages like those this study sought to teach are compiled into working programs. Then I discussed how the lexical analysis, syntax analysis, and semantic analysis stages of these processes are currently leveraged by both professional and pedagogical tools to support programmers' efforts.

This study likewise seeks to utilize lexical, syntax, and semantic analysis approaches to support a new pedagogical tool to provide students with targeted supplemental instruction. The approach I am proposing leverages Microsoft's existing Roslyn compiler and Visual Studio infrastructure, retargeting some of its existing tools to support pedagogical instead of professional development practices. Other development environments like BlueJ could likewise implement a similar approach, potentially integrating the work De Ruvio et al. (2018) have done identifying semantic style indicators in the Java language.

Finally, the approach used in this study amounts to an assessment tool that is used formatively and could also be utilized for summative assessments. With this in mind, we also discussed current algorithmic techniques used to assess learning in computer science. In doing so, we established that the proposed approach represents a new direction for computer science education, and one that has not been employed or investigated in the discipline's body of knowledge yet.

Chapter 3 - Methods

This chapter provides an overview of the research approach used in this study. As my goal was to create a software system capable of identifying issues in student code and providing supplemental instruction, I adopted the *design science* research methodology. Design science "is the scientific study and creation of artefacts as they are developed and used by people with the goal of solving practical problems of general interest" (Johannesson & Perjons, 2014, p. 7). Thus, the software developed as part of this study (referred to in Design Science as an *artifact*) is both an outcome of the research and a focus of research activities.

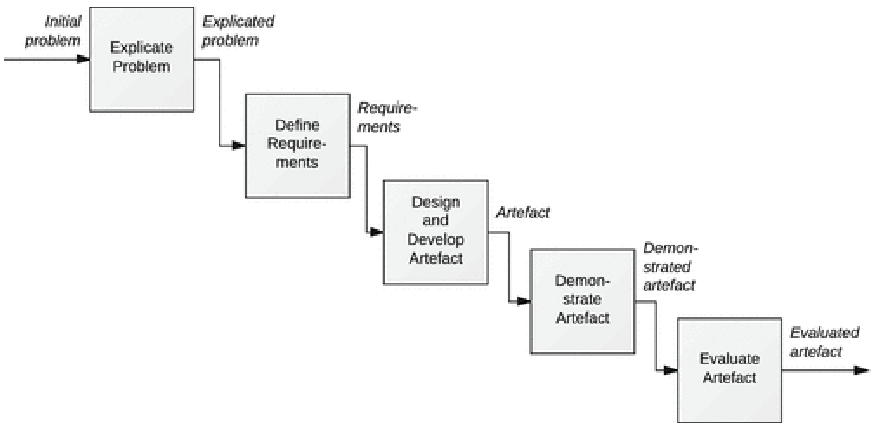
The next few sections describe the research questions, methodology, research and development activities, and study participants.

Methodology and Research Design

This study leveraged code analysis algorithms to automatically provide targeted contextual instruction. As the focus of this study is creating a software artifact, it was a good fit for the *design science* research methodology, which focuses on the creation of artifacts to solve problems and is further characterized by 1) the use of a rigorous research strategy, 2) the aim of producing new knowledge related to an existing field of study, 3) a commitment to communicate results to both practitioners and researchers (Johannesson & Perjons, 2014). The methodological framework of design science consists of a series of activities conducted in order, as laid out in Figure 3.1.

Figure 3.1

The Activities of Design Science Research

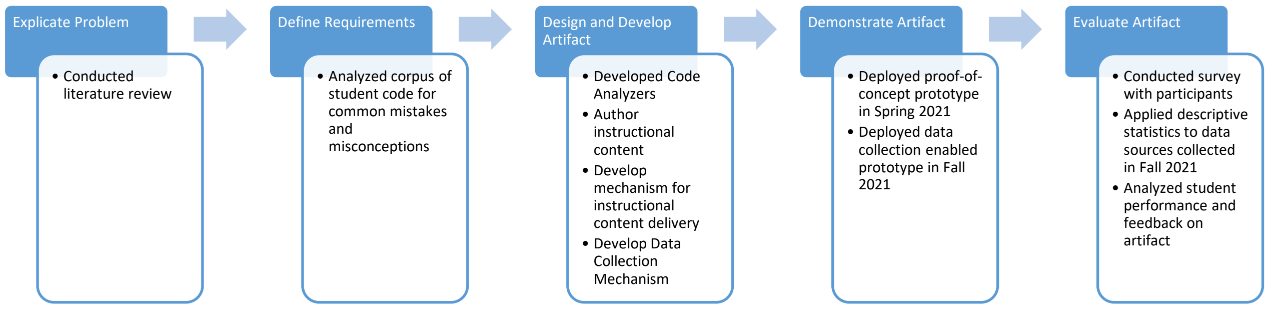


Note: Reprinted from *An Introduction to Design Science* (p. 77), by P. Johannesson and E. Perjons, 2014, Springer, Cham. Copyright 2014 by Springer International Publishing. Reprinted with permission.

Each of these activities may necessitate its own research strategy, which may be qualitative or quantitative in nature. This emphasis on research strategies reflects the understanding that “the purpose of design science research is not only to create artifacts but also to answer questions about them and their environments” (Johannesson & Perjons, 2014, p. 77). These activities also must strike a balance between the practical problems to be solved and the knowledge questions to be answered. These are both present in design science (which is what differentiates it from pure design) but in design science the practical problem is the priority (Wieringa, 2010). Working from this understanding, the research activities involved in each step of this study appear in Figure 3.2 below.

Figure 3.2

The Design Science Steps and the Corresponding Activities from this Study


Explicate Problem
• Conducted literature review
Define Requirements
• Analyzed corpus of student code for common mistakes and misconceptions
Design and Develop Artifact
• Developed Code Analyzers
• Author instructional content
• Develop mechanism for instructional content delivery
• Develop Data Collection Mechanism
Demonstrate Artifact
• Deployed proof-of-concept prototype in Spring 2021
• Deployed data collection enabled prototype in Fall 2021
Evaluate Artifact
• Conducted survey with participants
• Applied descriptive statistics to data sources collected in Fall 2021
• Analyzed student performance and feedback on artifact

The next few subsections expand upon the activities undertaken in this study corresponding to the stages of Design Science. This discussion omits first step, *Explicate Problem*, as the results of this stage are reported as Chapters 1 and 2 of this study. Steps 2-4 are presented in Chapter 3. Step 5, Evaluate the Artifact, is presented in Chapter 4.

Data Sources and Participants

The study participants were drawn from the Fall 2019 - 2021 CIS 400 (Object-Oriented Design, Implementation and Testing) course taught at Kansas State University. This is a sophomore-level programming course that focuses on teaching Object-Oriented programming theory and practice (the syllabus of which is Appendix A). Student activities in the course were separated into three categories:

1. The *daily work* involved the students reading the textbook and working through video tutorials that provided hands-on experience with the topics covered. The purpose of the daily work is to introduce the knowledge students are expected to learn and provide scaffolded experiences in how to put that knowledge to use.

2. The *milestones* were weekly assignments that iteratively built a series of computer programs for managing the day-to-day operations of a fast-food franchise. Each milestone drew upon the subject matter of the daily work, with increased nuance and variation. The primary goal of the milestones was to practice programming, reinforce good practice (i.e., code styling, testing, and documentation), and develop facility with iterative program development. A sample milestone assignment is included in Appendix C.
3. The *exams* are traditional paper exams consisting of long-text vocabulary and programming problems. The course had one preliminary exam, two midterms, and a comprehensive final that covered all object-oriented theory and practice covered in the course. A copy of the final exam from Spring 2021 is included as Appendix D.

Student grades in this course were weighted by these three categories. Daily work was 30% of the final course grade, milestones were 44%, and exams were 26%. Students entering the course had at least four prior post-secondary programming courses.

Student Work from Prior Semesters

The original design of this study included a quasi-experimental evaluation component comparing the students from two subsequent semesters, with the latter group using the developed artifact. Unfortunately, it was during this time that the COVID-19 pandemic reached our university. The changes to the course throughout this time are detailed in Table 3.1 below.

Table 3.1

CIS 400 COVID-19 Impact and Instructional Approaches by Semester

Semester	Description
Fall 2019	The course is taught using traditional lecture/lab instructional approach.
Spring 2020	The COVID-19 pandemic reaches the state in February. Classes transition to online delivery after spring break.
Fall 2020	Courses are offered through online delivery only; we transition to using recorded programming tutorials and Zoom lectures.
Spring 2021	Courses are resumed on-campus with masks, social distancing, and frequent disinfection; we adopt a ‘flipped’ classroom allowing students to quarantine with minimal impact.
Summer 2021	Course was taught online in a compressed, 8-week format.
Fall 2021	Social distancing requirements are lifted, but masks remain. Course remains in ‘flipped’ format.

The milestone assignments from Fall 2019 – Spring 2021 proved useful in the development the artifact. This body of source code and the notes produced by the teaching team when grading them was utilized primarily for the purpose of identifying common student errors. While the exact number of milestone assignments varied somewhat between semesters (with the summer semester assignments significantly condensed), this body contained more than 3,750 student milestone submissions and represented the work of 388 unique students.

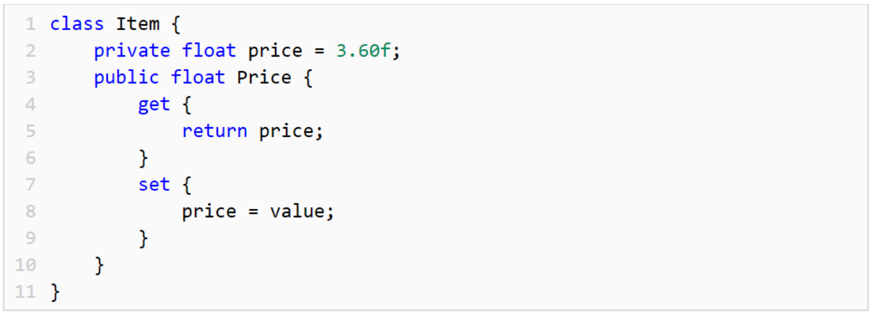
Define Requirements – Analyze Corpus of Student Code

From the first offering of the CIS 400 course in Fall 2019, my teaching team (a group of four to six undergraduate teaching assistants) and I kept notes on what kinds of mistakes our students routinely made in their C# code. Essentially, we employed the *content analsyis* research strategy, examining the students' code for evidence of misconceptions and mistakes and categorizing them. This bears a striking resemblance to the qualitative coding strategies employed in grounded theory. However, instead of looking for statements in prose that seem to convey the same idea, we are looking for patterns in software source code such as those of the semantic style indicators discussed in Chapter 2. These patterns were then categorized into groups that we believed represented permutations of the same misconception, much as qualitative codes form the basis for theory building in grounded theory.

For example, the C# language used in the course includes in its grammar the concept of a *property*, a class member that provides an abstraction for getter and setter methods, allowing access to the underlying value using a notation more like that of a field. An example of a well-constructed property using a private backing field is presented in Figure 3.3.

Figure 3.3

Well-constructed Property

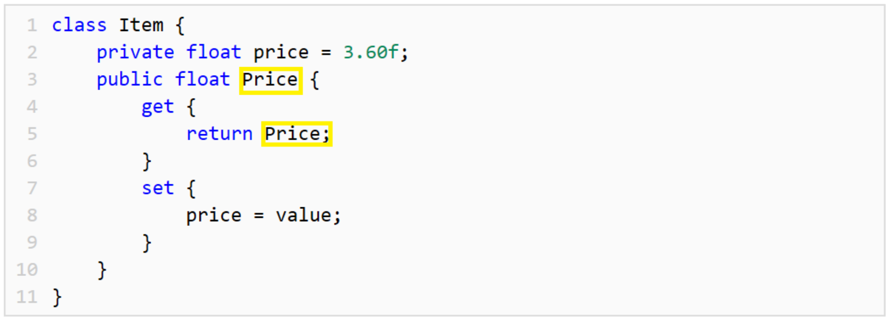


```
class Item {
    private float price = 3.60f;
    public float Price {
        get {
            return price;
        }
        set {
            price = value;
        }
    }
}
```

Notice the symbols used to define both the property, **Price**, and the private backing field, **price**. A common mistake made by students in the course was to use the property symbol within its own getter or setter, as demonstrated in Figure 3.4. Notice the symbol **Price** is repeated in its own getter method. This creates a circular reference – if at any point the code attempts to reference **Price**, it will invoke its getter, which will reference **Price,** causing it to invoke its getter, which will reference **Price**, causing it to invoke its getter. This process of recursion continues infinitely, or more accurately, until the program runs out of stack memory, causing it to crash.

Figure 3.4

Erroneously Constructed Property



```
class Item {
    private float price = 3.60f;
    public float Price {
        get {
            return Price;
        }
        set {
            price = value;
        }
    }
}
```

We identified multiple code patterns in student submissions that shared this common element of referencing a property from within itself. The self-reference could appear in the getter, setter, or both. Each of these related patterns were grouped under a single code issue which we identified as a *Property Self-Reference*.

These notes were generated as part of our milestone assignment grading process and discussed and collated at our weekly teaching team meetings. This effort was carried out over six semesters (three Fall, two Spring, and one Summer), covering several modalities (in-person, online, and hybrid/flipped). While the exact number of milestone assignments varied somewhat between semesters (with the summer semester assignments significantly condensed), the total number of student programs so evaluated was more than 3,750 and represented the work of 388 unique students. The code issues identified during this stage from student work then served as the starting point for the next steps in designing and developing the software artifact.

Design and Develop Artifact - Develop Code Analyzers

I, along with a student software engineering team consisting of Austin Hess, Sam Brunner, and Zachary Nelson, developed Roslyn code analyzers to detect the commonly encountered issues in student code identified in the previous step. In addition, I modified an existing open-source library for detecting common issues with the **INotifyPropertyChanged** interface (Larson & Musser, 2021). These analyzers embody an algorithm that evaluates the student code for the pattern(s) corresponding to the issue using the Roslyn compiler and code analysis tools and produces a *diagnostic* – a data structure describing the issue and where it was found. These are the same open-source tools that Microsoft uses to compile C# code and identify syntax errors in that code.

The code analyzer used to find the property self-reference errors described in the previous section can be found in Figure 3.5. The **AnalyzeNode()** method is called by the Roslyn compiler as it processes the definition of the property in source code (examples of which were given in Figure 3.3 and Figure 3.4). The **context** parameter represents both the syntax element representing the property definition *and* metadata created by the compilation – like the symbol table, a list of all user-defined symbols in the program (basically, everything the programmer gave a name to). The algorithm defined in this method looks for the symbol corresponding to the property *within* the definition of the property's **get** and **set** bodies, while excluding several edge cases.

As each code analyzer was developed, I also developed an accompanying test framework to verify the analyzer works as expected. These tested the analyzer against multiple instances of the mistake identified in the define requirements activity, ensuring the analyzer identified the issue. In addition, I tested against examples of correct code to ensure that no false positives were identified. This testing helped identify several errors *before* the code analyzers were put into use.

Figure 3.5

Code Analyzer to Detect Property Self-Reference Errors

```
/// <summary>
/// Analyzes a PropertyDeclarationSyntax Node for self-reference errors
/// </summary>
/// <param name="context">The context of the PropertyDeclarationSyntax Node</param>
private static void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
    // The context Node should be a PropertyDeclarationSyntax
    var propertyDeclaration = (PropertyDeclarationSyntax)context.Node;

    // The symbol corresponding to the property
    var propertySymbol = context.SemanticModel.GetDeclaredSymbol(propertyDeclaration);

    // Find all identifiers that exist within the property declaration
    var identifiers = propertyDeclaration.DescendantNodes()
        .OfType<IdentifierNameSyntax>();
    foreach (var identifier in identifiers)
    {
        // Determine if the identifier is wrapped in a `nameof()` expression
        // We can skip processing these
        var isNameOf = identifier.Ancestors()
            .OfType<InvocationExpressionSyntax>()
            .Where(ex => ex.Expression is IdentifierNameSyntax idSyntax
                 && idSyntax.Identifier.ValueText == "nameof")
            .Any();
        if (isNameOf) continue;
        // Get the symbol that corresponds to the identifier
        var symbolInfo = context.SemanticModel.GetSymbolInfo(identifier);
        // Check that the identifier symbol is not the property symbol
        if (SymbolEqualityComparer.Default.Equals(propertySymbol, symbolInfo.Symbol))
        {
            // Determine if the self-reference occured in a getter or setter
            AccessorDeclarationSyntax accessor = identifier.Ancestors()
                .OfType<AccessorDeclarationSyntax>()
                .FirstOrDefault();
            if (accessor != null && accessor.IsKind(SyntaxKind.SetAccessorDeclaration))
            {
                // Create the diagnostic
                var primaryLocation = propertyDeclaration.Identifier.GetLocation();
                var additionalLocations = new Location[] { identifier.GetLocation() };
                var diagnostic = Diagnostic.Create(
                    PropertySelfReferenceRule,
                    primaryLocation,
                    additionalLocations,
                    new string[] { propertyDeclaration.Identifier.ValueText, "set" }
                    );
                context.ReportDiagnostic(diagnostic);
            }
            else
            {
                // Create the diagnostic
                var primaryLocation = propertyDeclaration.Identifier.GetLocation();
                var additionalLocations = new Location[] { identifier.GetLocation() };
                var diagnostic = Diagnostic.Create(
                    PropertySelfReferenceRule,
                    primaryLocation,
                    additionalLocations,
                    new string[] { propertyDeclaration.Identifier.ValueText, "get" }
                    );
                context.ReportDiagnostic(diagnostic);
            }
        }
    }
}
```

The required structure of the diagnostic includes a unique *diagnostic code* to identify it, and the location(s) in the code where the issue was found. The diagnostic can also be supplied a help URL directed at a web page describing the detected issue in more detail. The diagnostic code is a sequence of letters corresponding to the category, followed by a number. The final prototype contained 37 code analyzers, and the corresponding diagnostic codes and their meaning can be found in Table 3.2 below.

Table 3.2

Design Issue Diagnostic Codes

Diagnostic	Description
FAL0001	Property contains self-reference
FAL0002	Using the ‘‘new’’ keyword hides inherited property.

Diagnostic	Description
FAL0003	Using ‘‘new’’ keyword hides inherited property.
INCC0001	NotifyCollectionChangedEventArgs needs arguments
INCC0002	NotifyCollectionChangedEventArgs needs two arguments when notifying of an ‘‘Add’’ event
INCC0003	NotifyCollectionChangedEventArgs needs three arguments when notifying of a ‘‘Remove’’ event
INPC001	The class has mutable properties and should implement InotifyPropertyChanged.
INPC002	Mutable public property should notify.
INPC003	Notify when property changes
INPC016	Notify after updating the backing field.
INPC017	Backing field name must match
NAM0001	All segments of a Namepace should be in Pascal Case
NAM0002	Classes should be named using Pascal Case
NAM0003	Class names should not begin with ‘‘I’’ followed by a capital letter
NAM0004	Structs should be named using Pascal Case
NAM0005	Struct names should not begin with an ‘‘I’’ followed by a capital letter
NAM0006	Enums should be named using Pascal Case
NAM0007	Enum names should not begin with ‘an’ ‘I’ followed by a capital letter
NAM0008	Interface names should begin with ‘an’ ‘I’
NAM0009	Interface names should be in Pascal Case after the prefix letter
NAM0010	Methods should be named using Pascal Case
NAM0011	Properties should be named using Pascal Case
NAM0012	Parameters should be named using Camel Case

Diagnostic	Description
NAM0013	Fields declared ‘public’ and ‘const’ should be named using Pascal Case
NAM0014	Fields declared ‘private’ or ‘protected’ should be prefixed with an underscore (‘_’)
NAM0015	Fields declared ‘private’ or ‘protected’ should be named using Camel Case after the prefix
NAM0016	Local variables should be named using Camel Case
XMLC0001	Class is missing summary XML comment tag
XMLC0002	Struct is missing summary XML comment tag
XMLC0003	Interface is missing summary XML comment tag
XMLC0004	Enum is missing summary XML comment tag
XMLC0005	Field is missing summary XML comment tag
XMLC0006	Property is missing summary XML comment tag
XMLC0007	Method is missing summary XML comment tag
XMLC0008	Method is missing param XML comment tag
XMLC0009	Method is missing returns XML comment tag
XMLC0010	Malformed XML comment

Code analyzers built following these principles are compiled into a dynamically linked library (a file of compiled code that can be utilized by other programs). This library can then be added to a project like the artifact developed for this study directly, or it can be packaged as a NuGet package which allows Visual Studio to integrate it directly into its normal error-finding process. The current version of the code analyzer project consists of 1,972 lines of code, and the test project consists of an additional 2,784 lines, and is available as a NuGet package at https://www.nuget.org/packages/KSU.CS.Pendant.CodeAnalysis/.

Design and Develop Artifact - Author Instructional Content

In providing the instructional content to correspond to the issues found in student code, I wanted to make the materials easy to update, portable, and familiar to the students. Accordingly, I wrote the content in markdown, a text-based format which allows for the easy inclusion of code examples that can be easily converted into HTML. An example markdown file can be seen in Figure 3.6.

Figure 3.6

Tutoring Content in Markdown for Property Self-Reference Error

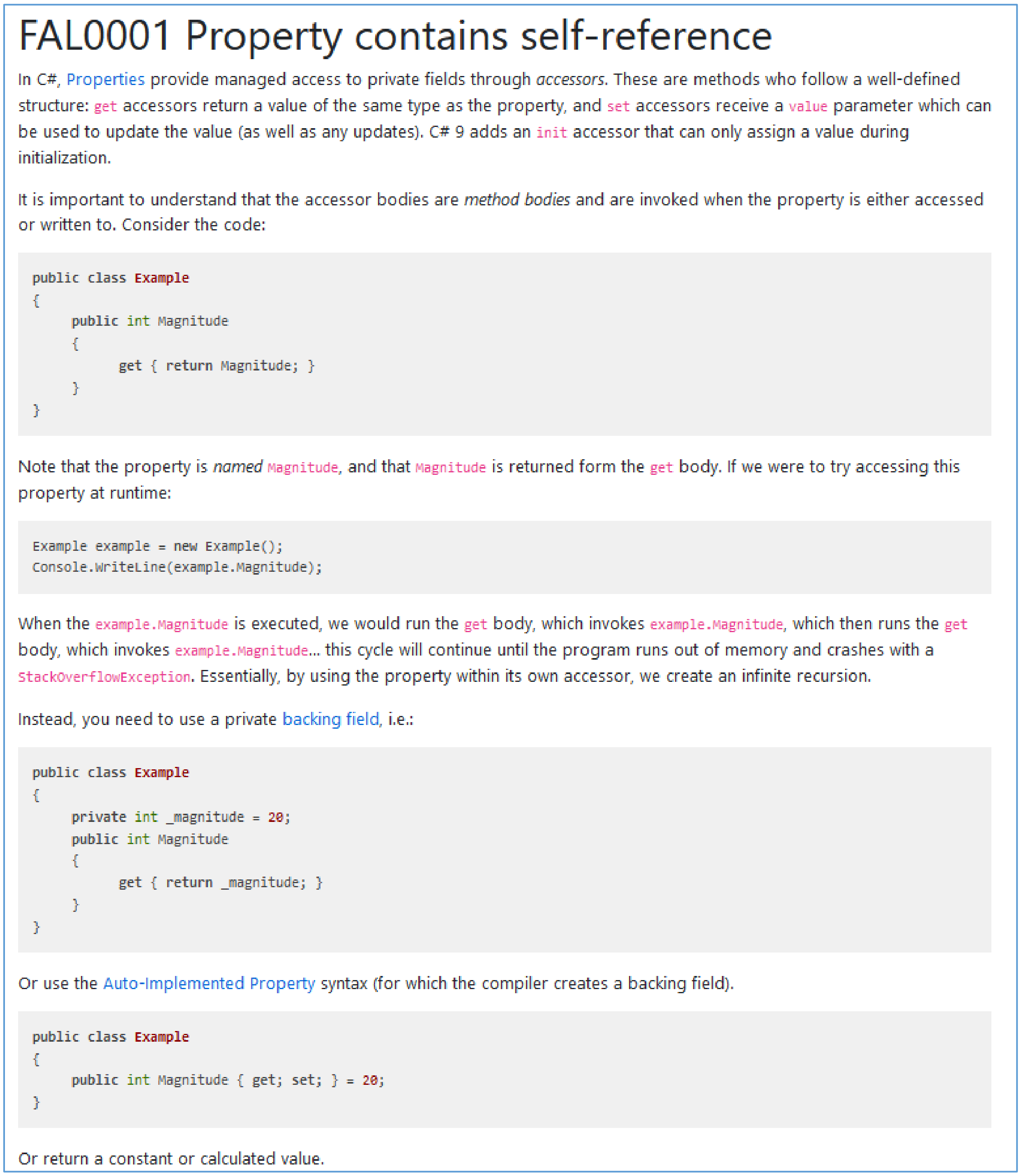
~~~markdown
In C#, [Properties](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-
structs/properties) provide managed access to private fields through _accessors_.  These are methods
who follow a well-defined structure: `get` accessors return a value of the same type as the property,
and `set` accessors receive a `value` parameter which can be used to update the value (as well as any
updates).  C# 9 adds an `init` accessor that can only assign a value during initialization.

It is important to understand that the accessor bodies are _method bodies_ and are invoked when the
property is either accessed or written to.  Consider the code:

```csharp
public class Example
{
     public int Magnitude
     {
           get { return Magnitude; }
     }
}
```

Note that the property is _named_ `Magnitude`, and that `Magnitude` is returned form the `get` body.
If we were to try accessing this property at runtime:

```csharp
Example example = new Example();
Console.WriteLine(example.Magnitude);
```

When the `example.Magnitude` is executed, we would run the `get` body, which invokes
`example.Magnitude`, which then runs the `get` body, which invokes `example.Magnitude`... this cycle
will continue until the program runs out of memory and crashes with a `StackOverflowException`.
Essentially, by using the property within its own accessor, we create an infinite recursion.
~~~

~~~
Instead, you need to use a private [backing field](https://docs.microsoft.com/en-
us/dotnet/csharp/programming-guide/classes-and-structs/properties#properties-with-backing-fields),
i.e.:

```csharp
public class Example
{
     private int _magnitude = 20;
     public int Magnitude
     {
           get { return _magnitude; }
     }
}
```

Or use the [Auto-Implemented Property](https://docs.microsoft.com/en-us/dotnet/csharp/programming-
guide/classes-and-structs/properties#auto-implemented-properties) syntax (for which the compiler
creates a backing field).

```csharp
public class Example
{
     public int Magnitude { get; set; } = 20;
}
```
Or return a constant or calculated value.
~~~

The sections separated by triple backtick marks (```) in Figure 3.2 are used to indicate a code example. An optional language signifier (the "csharp") can be used to indicate the programming language used in the example. When the markdown is rendered into HTML or another form, these code sections are styled to appear similar to how they would look in an integrated development environment – using strict formatting for whitespace and highlighting keywords. This makes it easier for students to read, while minimizing the work for the author. The example markdown from Figure 3.6 rendered as HTML by one of these libraries appears in Figure 3.7, below.

Figure 3.7

Tutoring Content for the Property Self-Reference Rendered as HTML



FAL0001 Property contains self-reference

In C#, Properties provide managed access to private fields through *accessors*. These are methods who follow a well-defined structure: `get` accessors return a value of the same type as the property, and `set` accessors receive a `value` parameter which can be used to update the value (as well as any updates). C# 9 adds an `init` accessor that can only assign a value during initialization.

It is important to understand that the accessor bodies are *method bodies* and are invoked when the property is either accessed or written to. Consider the code:

```
public class Example
{
    public int Magnitude
    {
        get { return Magnitude; }
    }
}
```

Note that the property is *named* `Magnitude`, and that `Magnitude` is returned form the `get` body. If we were to try accessing this property at runtime:

```
Example example = new Example();
Console.WriteLine(example.Magnitude);
```

When the `example.Magnitude` is executed, we would run the `get` body, which invokes `example.Magnitude`, which then runs the `get` body, which invokes `example.Magnitude`... this cycle will continue until the program runs out of memory and crashes with a `StackOverflowException`. Essentially, by using the property within its own accessor, we create an infinite recursion.

Instead, you need to use a private backing field, i.e.:

```
public class Example
{
    private int _magnitude = 20;
    public int Magnitude
    {
        get { return _magnitude; }
    }
}
```

Or use the Auto-Implemented Property syntax (for which the compiler creates a backing field).

```
public class Example
{
    public int Magnitude { get; set; } = 20;
}
```

Or return a constant or calculated value.

Design and Develop Artifact - Develop Instructional Content Delivery Mechanism

While HTML files generated from the markdown as described above could have been delivered by a webserver configured to serve static files, such as Apache or Nginx, or a content management system such as the one our university uses for its website, doing so would have missed an opportunity for collecting data on how often students access that content. Accordingly, during this stage I built a custom web server in ASP.NET to deliver the HTML content corresponding to the error codes. This server then collected data every time the instructional content was accessed – specifically, when the access occurred, and which student requested it. The server was hosted on a departmental server and utilized our university's single sign on system to determine the students' identity.

This server was further augmented to collect additional data on how students were engaging with the tutoring process, which will be discussed in the next section.

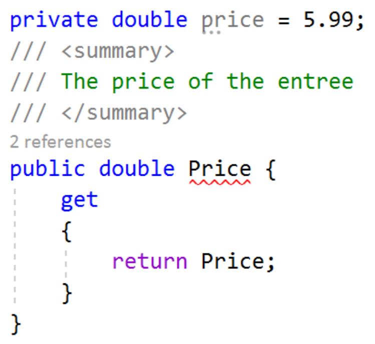
Design and Develop Artifact - Develop Data Collection Mechanism

While the code analyzers used in conjunction with the supplemental instruction materials were enough to address the goals of the artifact, I also wanted to be able to collect additional data directly from the process of students engaging with it. This additional data was used during the evaluation process alongside a more traditional survey. Having both data sources allowed for a stronger confirmation of how students engaged with the platform.

Ideally the data collection mechanism would have been integrated directly into Visual Studio, the program the students use to write their programs. As discussed earlier, Visual Studio allows code analyzers to be installed using a NuGet package and doing so automatically applies the analyzer as code is written in the editor. This includes highlighting any areas a diagnostic identifies as containing an issue and adds the corresponding message to the list of errors and warning about the program being written. This means that to the student, the errors reported by the custom code analyzers are indistinguishable from the errors reported from the IDE. An example of the Property Self-Reference error being reported in Visual Studio can be seen in Figure 3.8 below.

Figure 3.8

The Property Self-Reference Error from my Custom Code Analyzer Flagged by Visual Studio

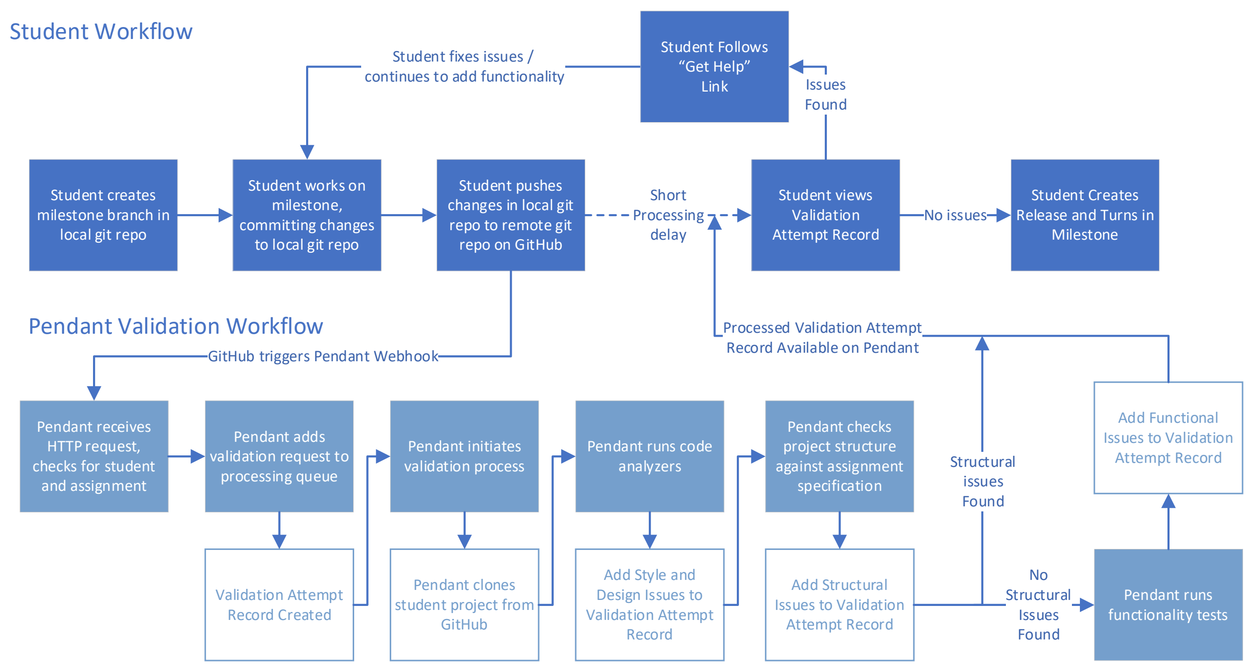

```
private double price = 5.99;
/// <summary>
/// The price of the entree
/// </summary>
2 references
public double Price {
    get
    {
        return Price;
    }
}
```

It is also a very easy mechanism for an instructor to utilize in assignments. For these reasons, this was the approach I used for the first prototype deployment in the Spring 2021 semester. However, there proved to be a couple of downsides: 1) students could uninstall the NuGet package, thereby stopping the reporting of diagnostics from the custom code analyzers, and 2) there was no viable mechanism for determining how often the issues were identified in student code or how students responded to them. While this approach is sufficient for regular use, it did not provide the level of data collection I desired to evaluate the code analyzers and their impact.

Accordingly, for Fall 2021, I modified the ASP.NET server delivering the tutoring content to also evaluate the students' code and collect data on how students engaged with it. I named this enhanced server tool *Pendant*. This approach created an additional challenge. With a NuGet code analyzer package, running the code analysis was directly triggered by Visual Studio when a student made a change. Thus, Visual Studio exposed the student's code to the code analyzer. The Pendant tool did not have this access, and therefore required a different strategy. The one I adopted utilized another best practice introduced in CIS 400, working with Git and branches. The process by which students interact with Pendant is laid out visually in Figure 3.9 and described in detail next.

Figure 3.9

The Process of Working with Pendant



Student Workflow
Student fixes issues / continues to add functionality
Student Follows "Get Help" Link
Issues Found
Student creates milestone branch in local git repo
Student works on milestone, committing changes to local git repo
Student pushes changes in local git repo to remote git repo on GitHub
Short Processing delay
Student views Validation Attempt Record
No issues
Student Creates Release and Turns in Milestone
Pendant Validation Workflow
GitHub triggers Pendant Webhook
Processed Validation Attempt Record Available on Pendant
Pendant receives HTTP request, checks for student and assignment
Pendant adds validation request to processing queue
Pendant initiates validation process
Pendant runs code analyzers
Pendant checks project structure against assignment specification
Structural issues Found
Add Functional Issues to Validation Attempt Record
Validation Attempt Record Created
Pendant clones student project from GitHub
Add Style and Design Issues to Validation Attempt Record
Add Structural Issues to Validation Attempt Record
No Structural Issues Found
Pendant runs functionality tests

Git is a tool for software developers to manage their development efforts by effectively 'saving' the current state of a program they have been writing. Each of these 'saves' is known as a *commit*, and the user can roll back any changes made to a program to one of these commits.

Thus, git can act as a kind of global ‘undo’ tool. Git is often used in conjunction with a remote repository like the GitHub website. Commits made to a local programming project can be *pushed* to GitHub, and then *pulled* to another computer. Thus, using git can allow a programmer to work on their program from any machine. These two abilities – the ability to roll back changes and easily move code between a lab computer and the students’ home computer, are extremely useful to students. Using Git this way is taught in CIS 300 – the course immediately before CIS 400, the subject body for this study.

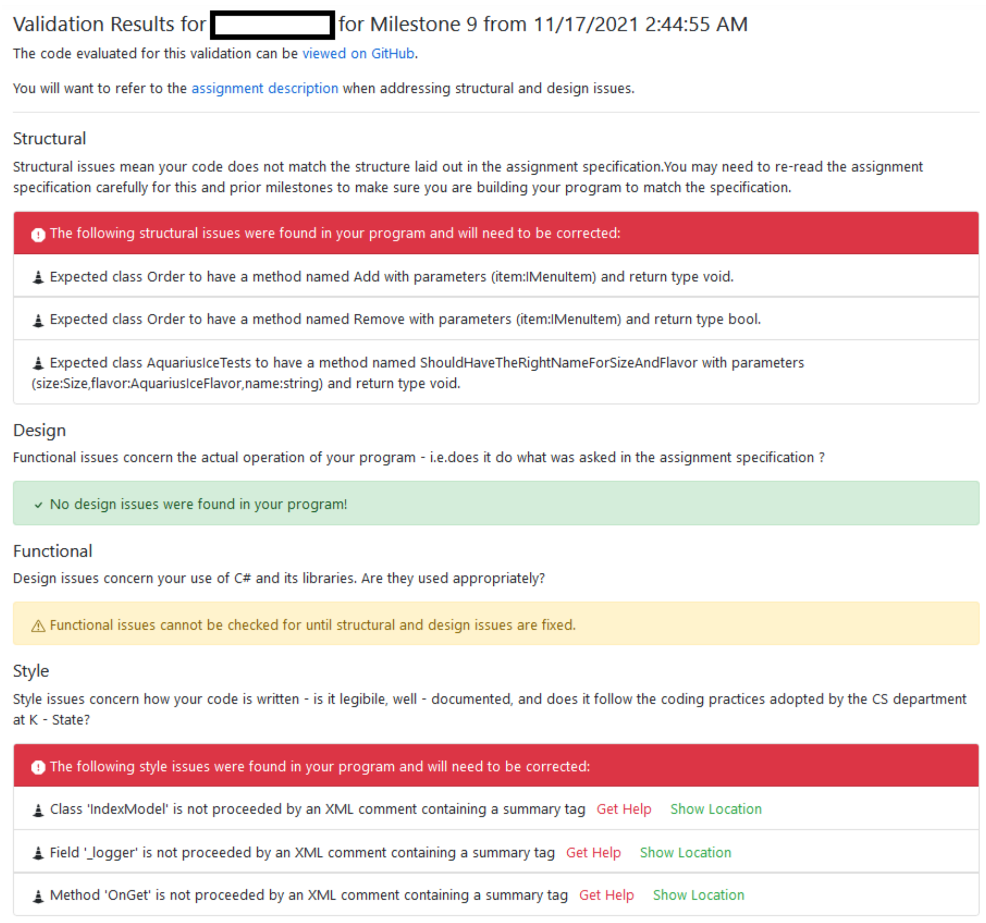
A third useful feature of Git is the ability to create *branches* of a project. Each branch represents a different possible direction to take the program. For example, if you had two possible approaches to solve a particular programming problem in mind, you could create a separate branch for each. You could write the specific code for each approach on the corresponding branch, and when you had finished you could pick whichever branch worked the best and *merge* it back into the main branch. Branches also serve an important role when working with other programmers collaboratively: each programmer typically makes changes to the program on their own branch, merges changes from the main branch into it, fixes any issues that causes, and then merge it back into the main branch. This approach becomes very important in later Computer Science courses and professional programming practice.

In CIS 400 we use *feature branches*. A feature branch is a kind of experiment, where the programmer creates a branch to develop a specific feature. Once that is working well, it is merged into the main branch. If, for some reason it does not go well, the branch can be abandoned and a new one started. The CIS 400 course structure involves creating a large software project iteratively over the entire semester. Each week the students add functionality to the program to meet a specific set of milestone assignment requirements. For each milestone, the students were encouraged to create a milestone feature branch and use it to save their work until they were ready to create a release tag (which is how they turn in the weekly assignment).

To allow Pendant to evaluate their code, the students added a *webhook* to their GitHub repository – this triggers a HTTP request against the Pendant web application. Once the webhook was set up, any time the student pushed their changes to GitHub, GitHub would send a HTTP request to Pendant describing the push, triggering the validation process on Pendant. From the information GitHub supplied, the Pendant tool determined the student, milestone, and exact code to analyze. In addition to running the code analyzers, the Pendant tool also verified that the structure of the students' code matched the guidelines laid out in the assignment specification. If it did so, the Pendant tool could also run a series of tests that would verify the students' program also functioned as expected (which unfortunately was not complete for the Fall 2021 semester). When the process finished, Pendant was able to serve the student a validation result report describing any issues found in the students' source code. An example of this report can be seen in Figure 3.10.

Figure 3.10

An Example Validation Result Report from the Pendant Tool



Validation Results for [redacted] for Milestone 9 from 11/17/2021 2:44:55 AM

The code evaluated for this validation can be viewed on GitHub.

You will want to refer to the assignment description when addressing structural and design issues.

Structural

Structural issues mean your code does not match the structure laid out in the assignment specification.You may need to re-read the assignment specification carefully for this and prior milestones to make sure you are building your program to match the specification.

The following structural issues were found in your program and will need to be corrected:

Expected class Order to have a method named Add with parameters (item:IMenuItem) and return type void.

Expected class Order to have a method named Remove with parameters (item:IMenuItem) and return type bool.

Expected class AquariusIceTests to have a method named ShouldHaveTheRightNameForSizeAndFlavor with parameters (size:Size,flavor:AquariusIceFlavor,name:string) and return type void.

Design

Functional issues concern the actual operation of your program - i.e.does it do what was asked in the assignment specification ?

✓ No design issues were found in your program!

Functional

Design issues concern your use of C# and its libraries. Are they used appropriately?

⚠ Functional issues cannot be checked for until structural and design issues are fixed.

Style

Style issues concern how your code is written - is it legibile, well - documented, and does it follow the coding practices adopted by the CS department at K - State?

The following style issues were found in your program and will need to be corrected:

Class 'IndexModel' is not proceeded by an XML comment containing a summary tag Get Help Show Location

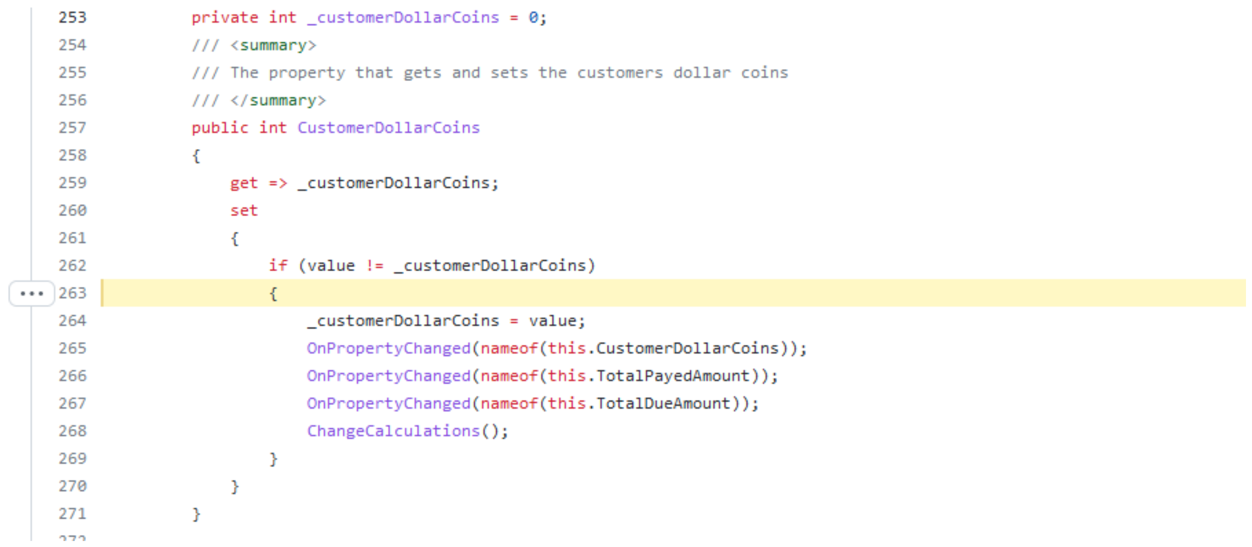
Field '_logger' is not proceeded by an XML comment containing a summary tag Get Help Show Location

Method 'OnGet' is not proceeded by an XML comment containing a summary tag Get Help Show Location

Clicking the 'Get Help' link in a validation report would open the supplemental instruction help page (an example of which was presented in the prior chapter as Figure 3.7). Clicking the 'Show Location' link in the validation report would also open the version of the students' code the analyzer checked in GitHub, highlighting the location of the issue. An example of this appears in Figure 4.3 below.

Figure 3.11

'Show Location' link displaying issue location in student code on GitHub


```
        private int _customerDollarCoins = 0;
        /// <summary>
        /// The property that gets and sets the customers dollar coins
        /// </summary>
        public int CustomerDollarCoins
        {
            get => _customerDollarCoins;
            set
            {
                if (value != _customerDollarCoins)
                {
                    _customerDollarCoins = value;
                    OnPropertyChanged(nameof(this.CustomerDollarCoins));
                    OnPropertyChanged(nameof(this.TotalPayedAmount));
                    OnPropertyChanged(nameof(this.TotalDueAmount));
                    ChangeCalculations();
                }
            }
        }
```

This was part of the normal workflow students are being trained to use – to commit changes to an online repository hosted on GitHub which can be used to both 'save' their progress on an assignment and to resume working on it from multiple locations (i.e., their home computer and a lab computer). This also demonstrated that the Code Analyzers could be utilized separately from Visual Studio. This is important, as it means programs built to automatically grade student assignments can make use of them.

Demonstrate Artifact – Deploy Prototypes

As mentioned above, the Code Analyzers built as part of this project were first deployed in the Spring 2021 semester as a NuGet Package that students were encouraged to install into their milestone project. Due to the chaotic nature of this semester (it was held completely online due to the COVID-19 pandemic) I was unable to do more than informal information gathering. It was unclear how many students were actively using the code analyzers, or what impact it might have been having. What it did help establish, however, was that a NuGet package deployment could work for regular instructional use.

To develop a richer picture of how students were engaging with the code analyzers and tutoring content, the ASP.NET platform mentioned previously was deployed on a departmental server and made accessible to the Fall 2021 students as the Pendant tool. The students were encouraged to use this server to evaluate their milestone projects as they worked with it. Additionally, I developed a survey tool that was administered to the students in the last week of the course (this survey is reprinted in Appendix B).

Evaluate Artifact

The final step in the study was to evaluate the artifact. The original plan for doing so was a quasi-experimental design comparing two cohorts drawn from subsequent semesters of the CIS 400 course, with the latter group using the artifact. However, the ongoing and evolving impacts of the COVID-19 pandemic introduced too many threats to the validity of such a comparison to be reasonably managed. Therefore, I pivoted the evaluation to consider just the students engaging with the tool. To provide a richer context for this evaluation, I created the survey and added the additional data collection integrated into the Pendant tool described above. These changes provided an additional data for evaluating the students' use of the code analyzers and engagement with the associated instructional content.

Ethical Considerations

As with any research conducted with human subjects, this study needed to carefully consider the ethics involved. The study was carried out under the auspices of the Kansas State University Institutional Review Board, which determined it meets the criteria for Exemption 1.

All data collected in this project was maintained in accordance with university and departmental policies.

Participation in the study was voluntary; students were not required to use either the NuGet package or the Pendant tool. An informed consent disclosure describing the research was provided in the course materials and covered in the classroom for the Fall 2021 semester of the CIS 400 course.

Summary

In this chapter we examined the research design for the study. The steps involved in implementing a design science methodology were identified. The semesters over which the artifact was developed/tested as well as the different instructional approaches driven by COVID-19 were described. The remainder of the chapter described the artifact requirements, design and development of the artifact, and the deployment of the proof-of-concept prototype in Spring 2021.

Chapter 4 - Results

This chapter details the results obtained from this study that were used in the final step of the Design Science methodology, *evaluate artifact,* conducted in Fall 2021. The artifact in question is the Pendant tool described in Chapter 3. Reported results include the data collected by the Pendant Tool as students utilized it, a student survey given to the class at the end of the semester, and an analysis of students' grades in relation to their use of the Pendant tool. The next few sections present the results from these efforts.

The Fall 2021 semester had three sections, A, B, and S, with a total of 61 students. Of these, 52 consented to participate in the study, but only 51 completed the survey on use of the artifact (printed as Appendix B). The data sources this study draws upon included:

- full student source code from daily lab assignments
- full student source code from weekly milestone assignments, each of which addressed a set of specific milestone requirements. An example of one such milestone assignment is included as Appendix C
- digital scans of all paper exams completed by the students over the semester
- the instructor and TA notes collected throughout the semester
- assignment, exam, and final course grades
- a survey of questions concerning the use of the artifact, included as Appendix B

These data sources and the research questions they address are summarized in Table 5.1.

Pendant Data

During the Fall 2021 semester, students participating in the study (N=32) created a total of 1,898 validations. The frequency of validations per student is presented in Figure 4.1, with two outlier values (of 81 and 168 validations) omitted for clarity. The distributions of these validations by milestone are displayed graphically in Figure 4.2 below. Every student created at least one milestone during this time, though many had milestones with no validation attempts.

Figure 4.1

Overall Validation Frequency per Student, with Outliers Removed

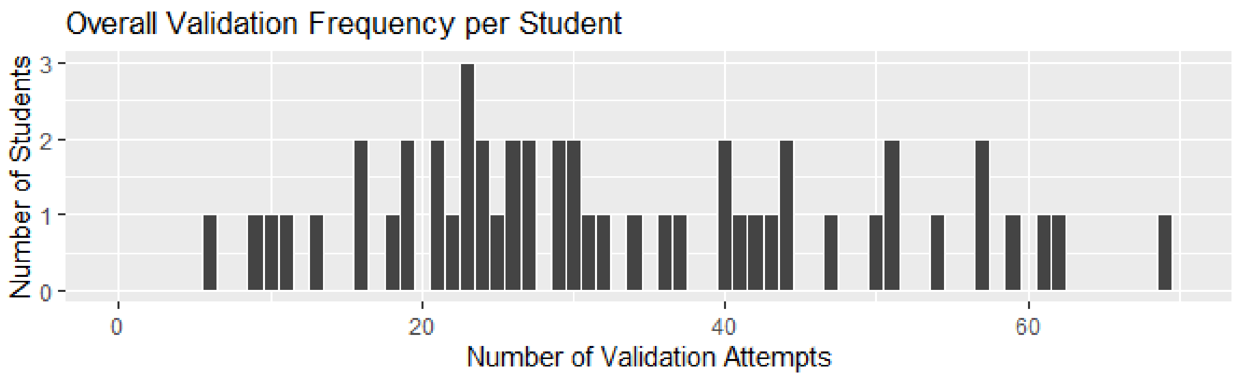
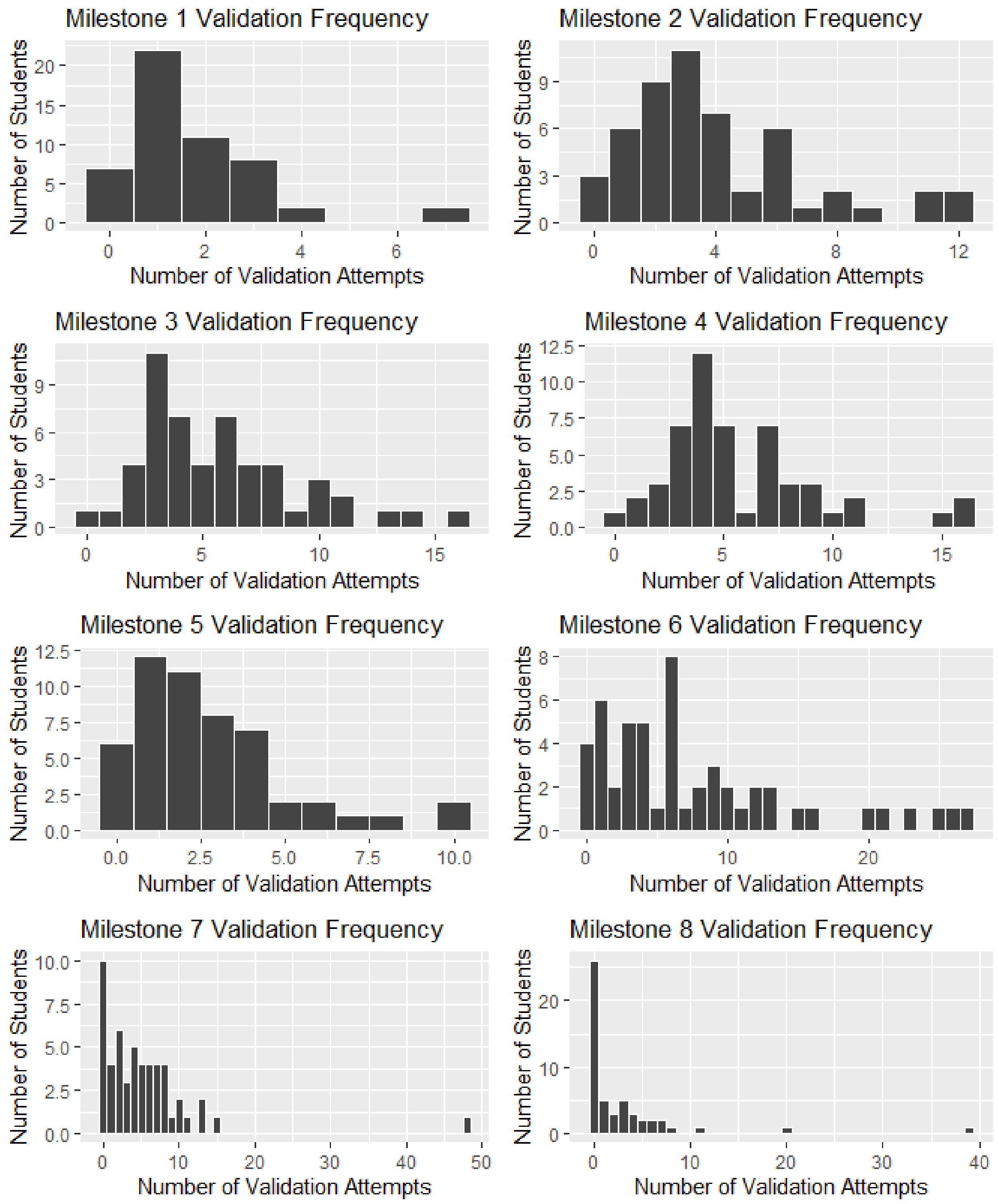


Figure 4.2

Number of Student Validation Attempts by Milestone Programming Assignment



During the Fall of 2021 semester, Pendant was able to detect a total of 40,714 individual issues in student code. Most of the identified issues could be categorized as problems related to coding style, primarily with XML documentation (27,427) and less so naming conventions (3,085), with design issues making up the remainder (10,202). Design issues are considered more serious, as they affect the program's ability to run (while coding style issues only affect ease of which a human can read the source code). The most detected design issues (and the number of occurrences for each) are listed in Table 4.2. As can be seen in the table, issues around implementing the **IPropertyChanged** interface were the most frequent, followed by the property self-reference syntax error, and then issues arising from an understanding of inheritance and method overriding and the 'new' keywords use.

Table 4.1

Design Issue Occurrences with Associated Tutoring Content Lookups from Pendant

Diagnostic	Occurrences	Lookups	Meaning
INPC002	3,373	14	Mutable public property should notify.
INPC003	3,198	4	Notify when property changes.
FAL0001	1,350	18	Property contains self-reference.
INPC017	613	5	Backing field name must match.
FAL0003	252	0	Using ''new' keyword hides inherited property.
INCC003	137	4	NotifyCollectionChangedEventArgs needs three arguments when notifying of a 'Remove' event.
FAL0002	32	1	Using the 'new' keyword hides inherited method.

Survey Results

To get a more complete picture of how students were engaging with and perceiving the Pendant tool, I developed a survey consisting of 33 questions organized into six thematic groups:

1. Perceived usefulness of the Pendant tool for students' learning goals
2. Perceived difficulty of using Git in the ways necessary to make use of Pendant
3. The student's time management and class attendance
4. The student's normal practices for working with Visual Studio, which can affect the impact of the NuGet Package approach to deploying the code analyzers
5. The student's use of particular features of the Pendant tool. This was prompted by the low numbers of help page lookups recorded by Pendant.
6. The student's willingness to use Pendant features if they were instead supplied by their integrated development environment (the program in which they write code)

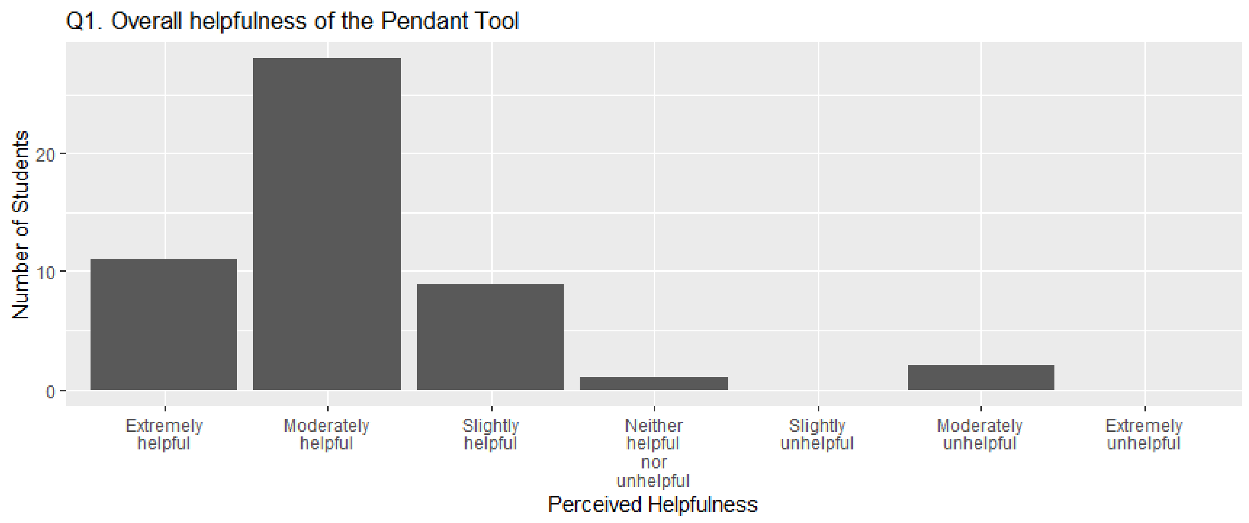
Most of the questions were ordinal, Likert-like scales with a single free-text answer question in each grouping to allow students to offer clarification for their responses to the earlier questions. The questions employed in this survey were intentionally broad ranging, as in addition to evaluating the artifact tool in its current form it was intended to provide direction for future work. The complete survey is included as Appendix B.

The survey was distributed to study participants at the end of the Fall 2021 semester. One study participant did not complete the survey, so there was a total of 51 responses. The rest of this section reports the aggregate responses to the survey questions.

The first question asked the students to report how useful they found the Pendant tool on a seven-point scale. The responses are presented graphically in Figure 4.3.

Figure 4.3

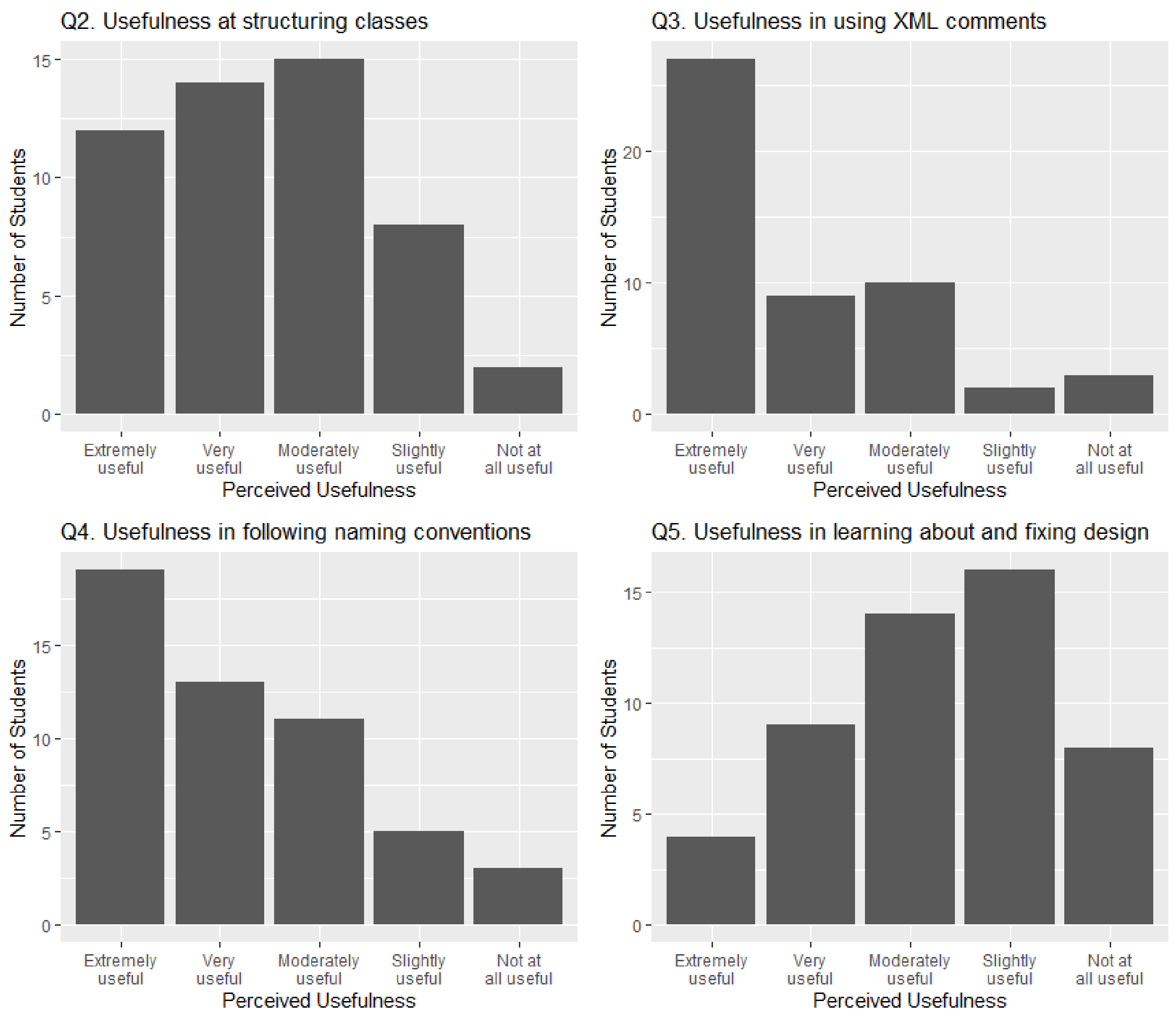
Overall Helpfulness of Pendant Tool



The next four questions asked the students to rate the usefulness of the Pendant tool on a five-point scale across specific learning goals: structuring classes, using XML comments, following naming conventions, and fixing design issues. The responses to these questions are reported in Figure 4.4.

Figure 4.4

Responses to questions from the survey addressing perceived usefulness of the Pendant Tool.



The sixth question was a free-text prompt: *Please feel free to offer any explanations of how you used Pendant during the semester.* The responses were qualitatively coded and the results are presented in Table 4.2 below.

Table 4.2

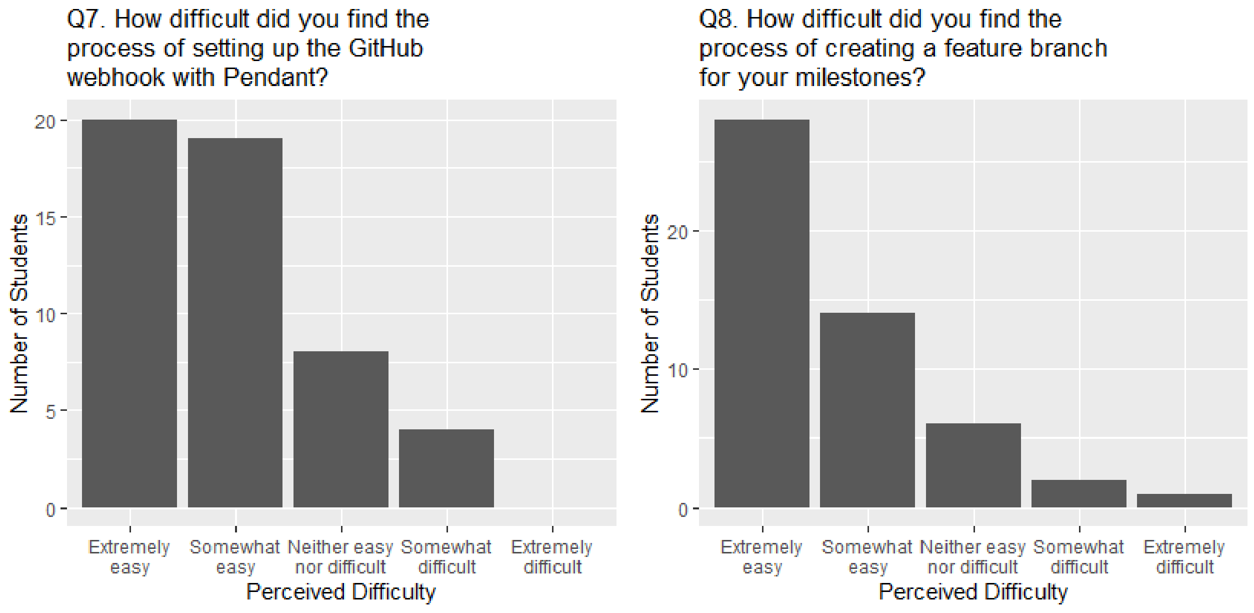
Response codes from question about how students used Pendant through the semester

Count	Code
8	Useful for the first part of the semester/not as useful later in the semester
11	Useful for checking XML comments
4	Useful for checking program structure
4	Useful for checking naming conventions
2	Useful for finding design issues
4	Useful for final quality check
2	Disliked the specificity of reported issues
3	Was not always available/encountered errors
1	Validations sometimes conflicted with assignment description

The next few questions concerned the use of Git and GitHub in conjunction with Pendant. Questions seven and eight asked about perceived difficulty setting up webhooks and using feature branches, both necessary steps to obtaining a validation report. These used a five-point scale, and the results are reported in Figure 4.5 below.

Figure 4.5

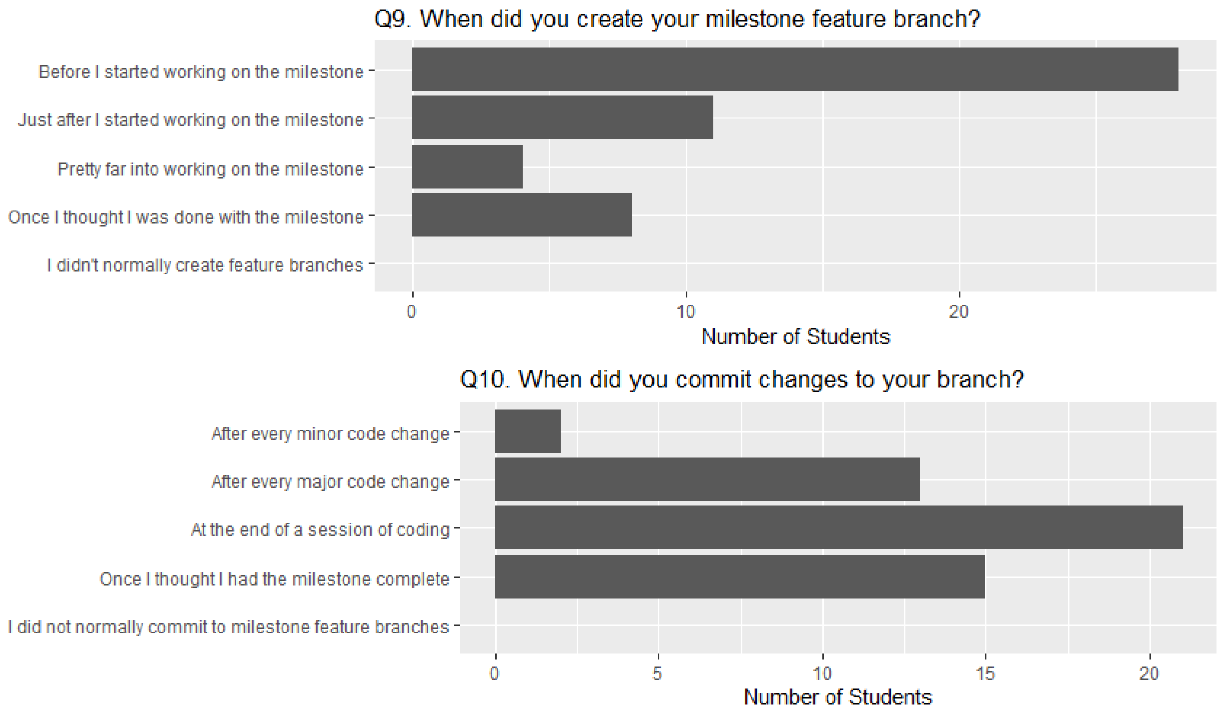
Perceived Difficulty Working with Git and GitHub Features



Questions nine and ten inquired about when feature branches were created and how often they were committed to, as both contributed to when and how often validation results would be generated for students. These results are reported in Figure 4.6 below.

Figure 4.6

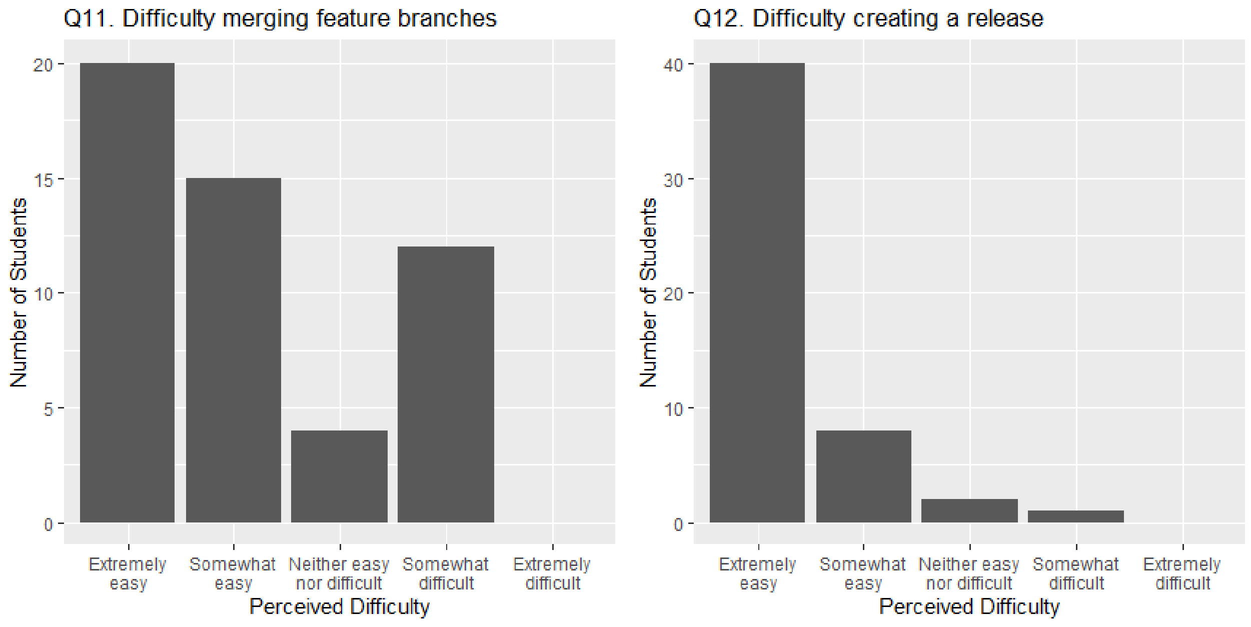
Self-Reported Branch Creation Time and Commit Frequency



Questions eleven and twelve concerned the perceived difficulty of merging a feature branch back into the main branch and creating a release on GitHub, both necessary steps for turning in the students' work for a milestone. These are reported in Figure 4.7 below.

Figure 4.7

Responses to Questions about Perceived Difficulty with Git and GitHub Processes



Question thirteen was another free-text prompt, *please feel free to offer any explanations of your answers to the questions concerning Git*. The responses were qualitatively coded and the results are presented in Table 4.3 below.

Table 4.3

Response Codes from Student Explanations About Using Git

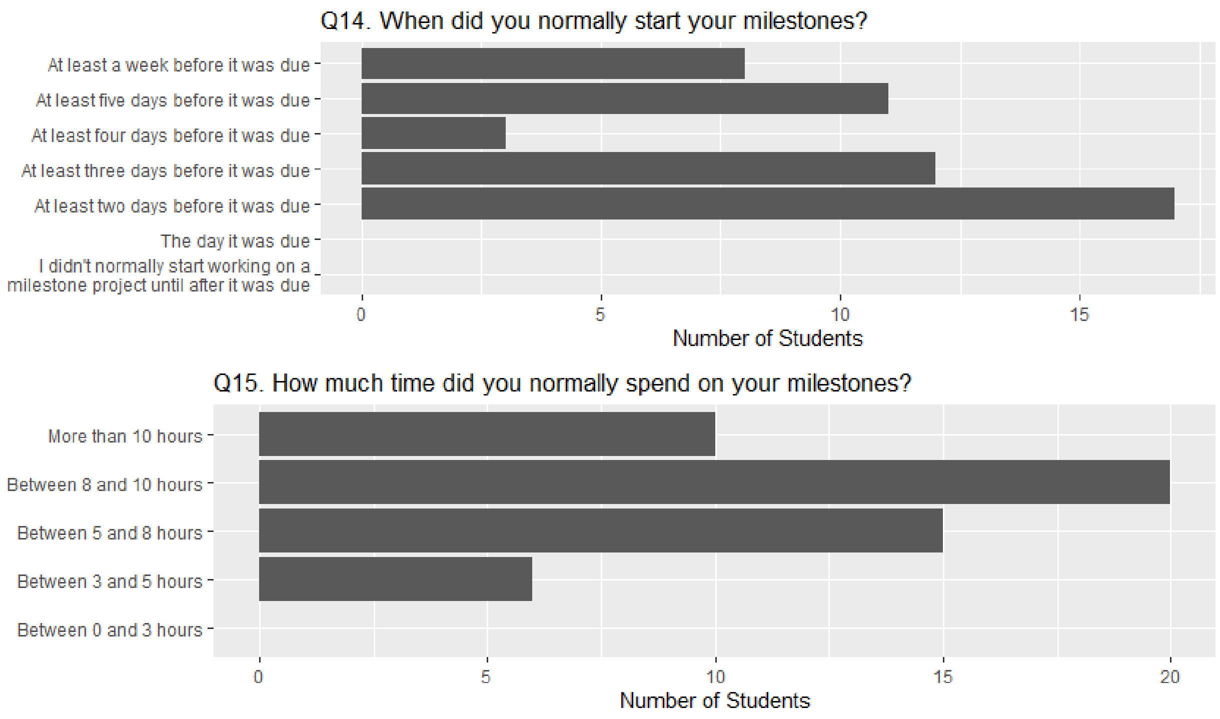
Count	Code
4	Working with Git gets easier with practice
3	Forgot to make feature branch before starting to write code
2	Find working with Git easy
2	Had difficulties working from multiple computers

2	Had some difficulties with merging
1	Had some difficulty creating releases
1	Used a “cheat sheet” listing the commands in the workflow

The next two questions concerned time management around the completion of milestones. Question fourteen asked when students normally started working on their milestones, and question fifteen about the amount of time spent on the milestone assignment. The results are reported in Figure 4.8 below.

Figure 4.8

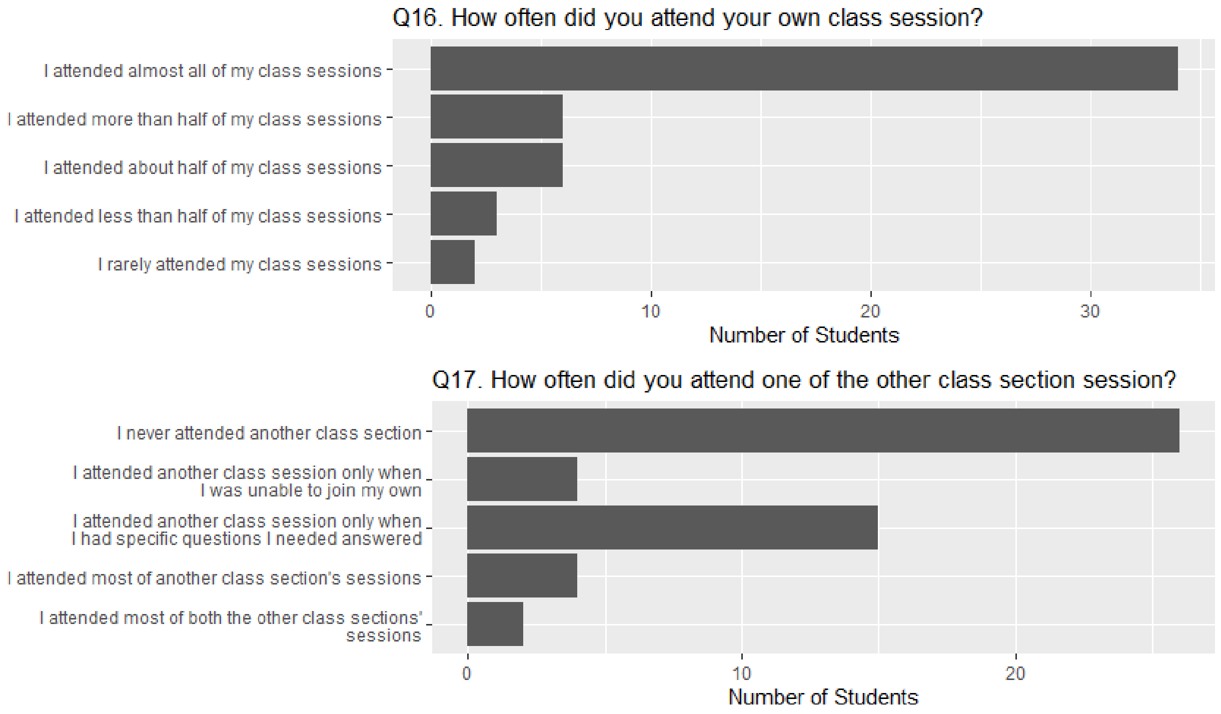
Responses to Questions about Time Management



The next two questions concerned class attendance. In the Fall 2021 semester, there were three class sections that each met once per week for two hours. As the course was run in a "flipped" format, this class session was treated as a combination of recitation and work session – a chance for students to ask questions, get help, and time to work on their assignments. As it became clear that the classroom was never full, students were invited to attend the class sessions of the other sections as they felt the need. Doing so was strongly suggested to several struggling students. The responses to these questions appear in Figure 4.9 below.

Figure 4.9

Responses to Questions about Class Attendance



Question 18 was again a free-text answer, allowing the students to clarify their answers to the questions about time management and class attendance. The responses here were primarily explanations for attendance and time spent, and other than noting that milestones varied in difficulty (2 students) were unique. One student provided a lingering illness as reason for poor attendance, another stated their only reason for attending class sessions was to pick up graded tests, and a final student explained they had poor time management skills which was why they started milestones late.

The next series of questions concerned the students' use of Visual Studio and was primarily included to understand how effectively students might make use of the NuGet package approach to deploying the Code Analyzers, as it uses Visual Studio's built-in error checking process. To access the supplemental instructional content in that approach, the student would need to view the error in the Error List and click the hyperlinked error code to navigate to the associated help page. The responses to these questions appear in Figure 4.10 and Figure 4.11.

Figure 4.10

Responses to Questions about using the Error List and its Hyperlinked Error Codes

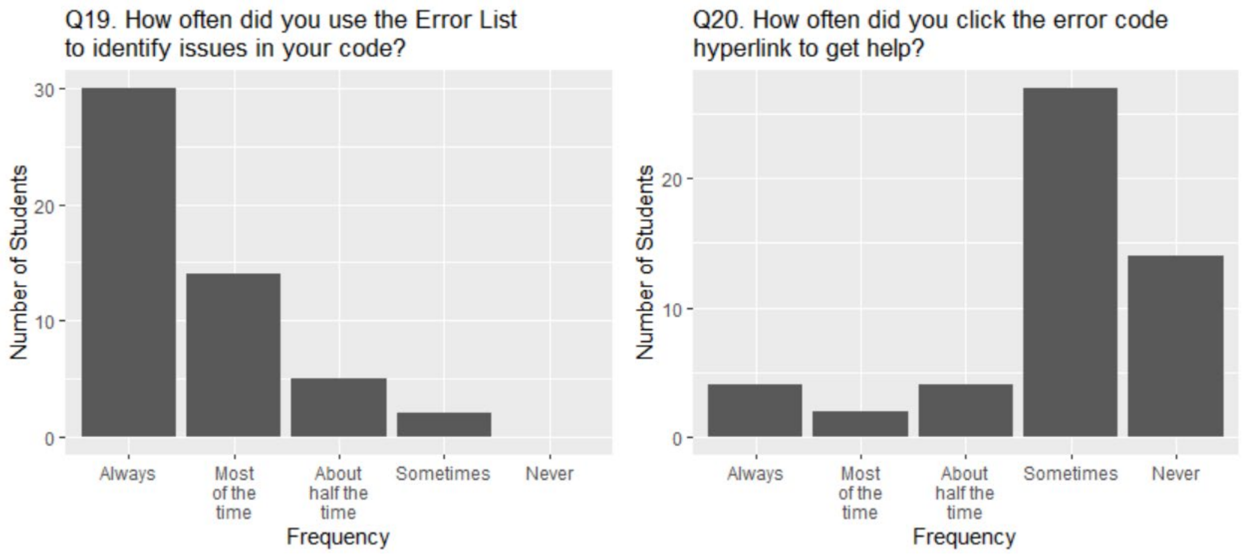
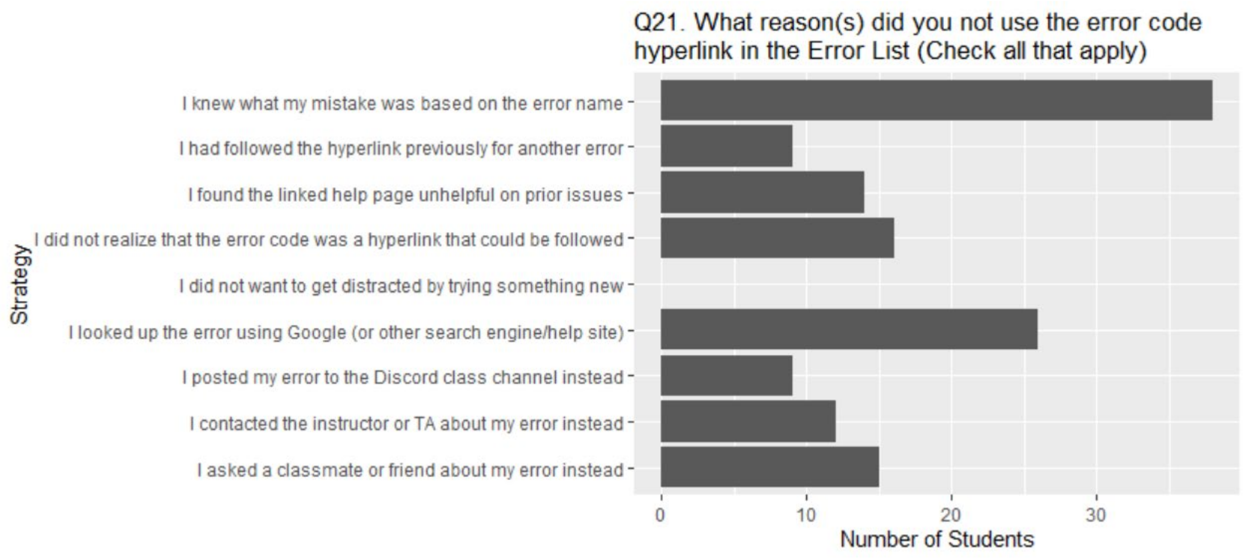


Figure 4.11

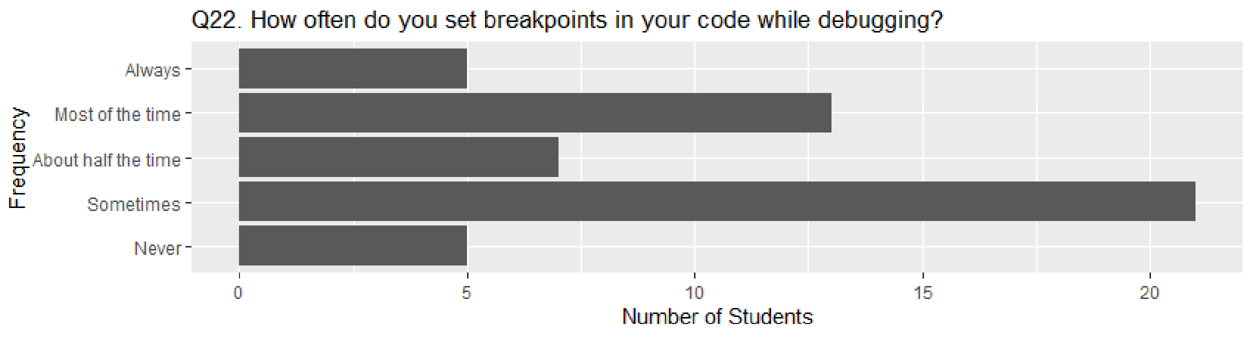
Students' Reasons for Not Following Hyperlinked Error Code in the Error List



Wrapping up the Visual Studio questions was one inquiring about use of the breakpoint feature of Visual Studio, reported in Figure 4.12. It was included as the instructor noticed many students did not make use of breakpoints, even though they were covered in prior courses.

Figure 4.12

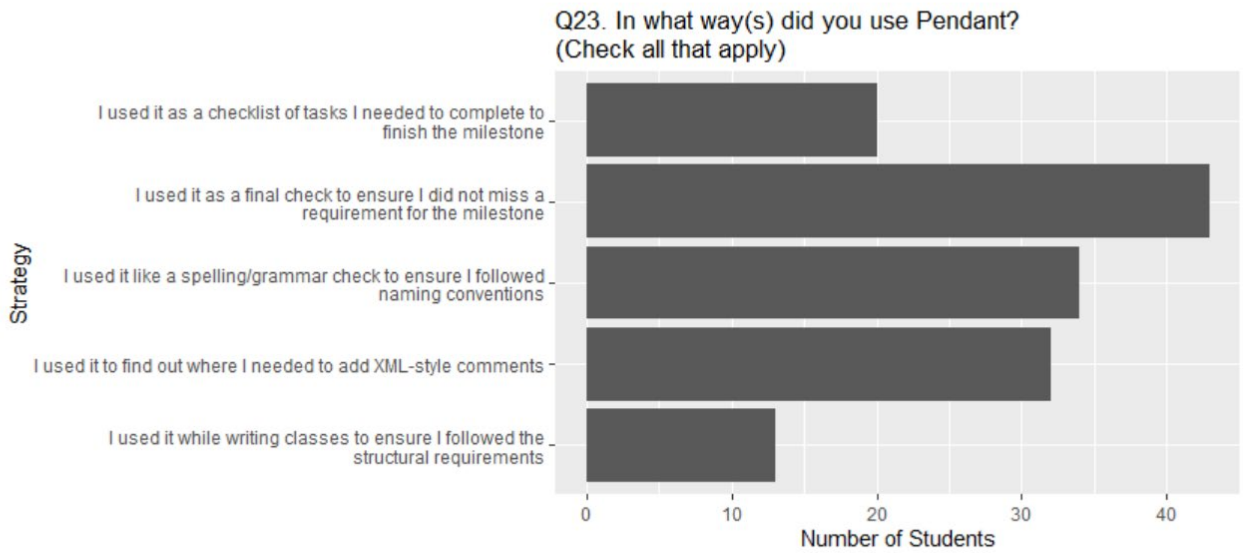
Students' Self-reported Use of Breakpoints.



The next series of questions returned to the student's use of Pendant. The first of these attempted to identify the ways that students were interacting with the Pendant tool and is reported in Figure 4.13 below.

Figure 4.13

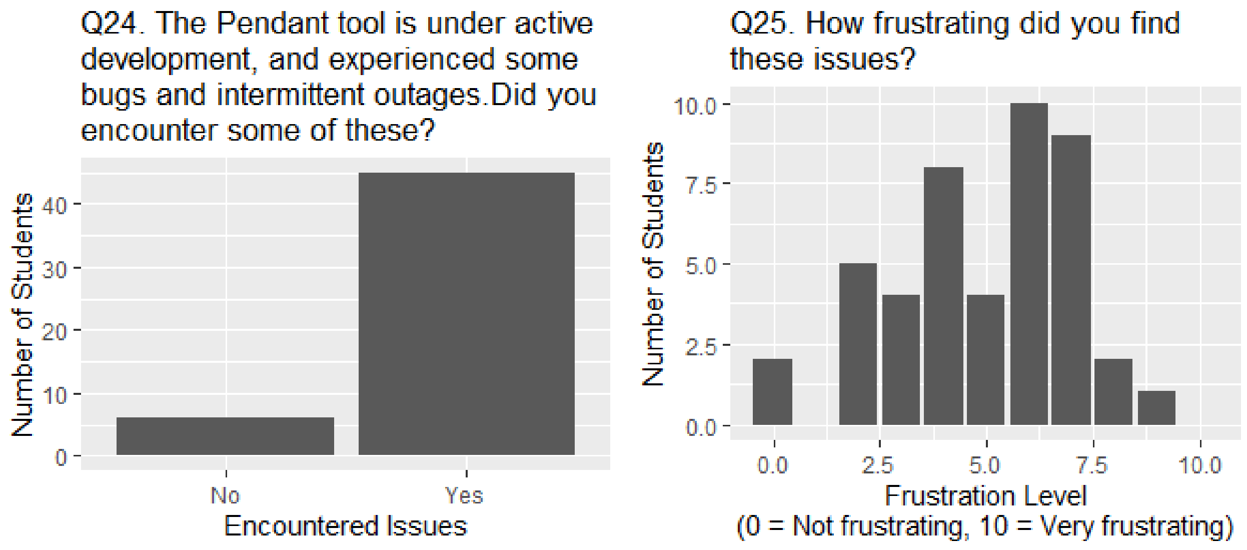
Students' self-reported Strategies for Using the Pendant Tool



The next few questions concerned outages and bugs with the Pendant and attempted to gauge the students' frustration with these problems on a ten-point scale with 10 being very frustrating (*M* = 4.9, *SD* = 2.1). These are reported in Figure 4.14 below.

Figure 4.14

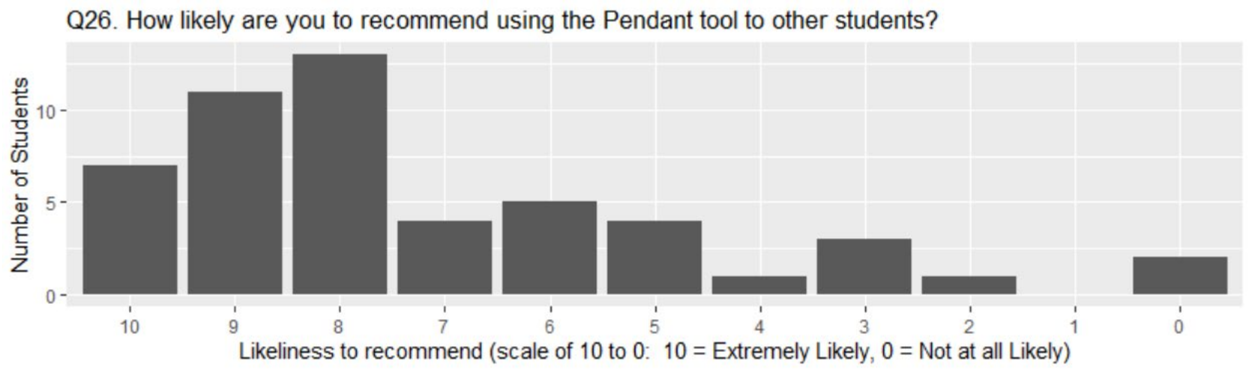
Student's Frustration with Bugs and Outages in the Pendant Tool.



The next question in this series asked students how likely they were to recommend the Pendant tool to peers indicated on a scale of 0 to 10, with 10 indicating very likely (*M* = 7.2, *SD* = 2.5). Its responses are presented in Figure 4.15 below.

Figure 4.15

Students' Self-reported Likeliness to Recommend Pendant to Others



The next series of questions concerned the students' use of the 'Get Help' link provided by Pendant for style and design issues. Question 27 asked how often students followed the link and is reported in Figure 4.16, and question 28 asked for reasons why students did not follow the link and is reported in Figure 4.17.

Figure 4.16

Students' Self-reported Frequency of using the 'Get Help' link in a Pendant Validation

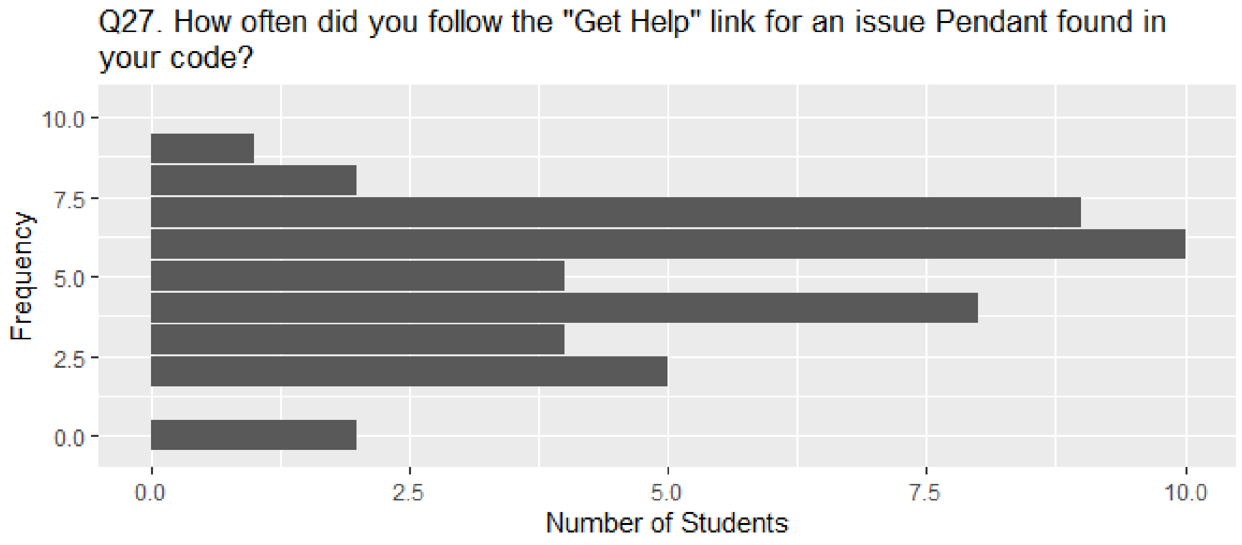
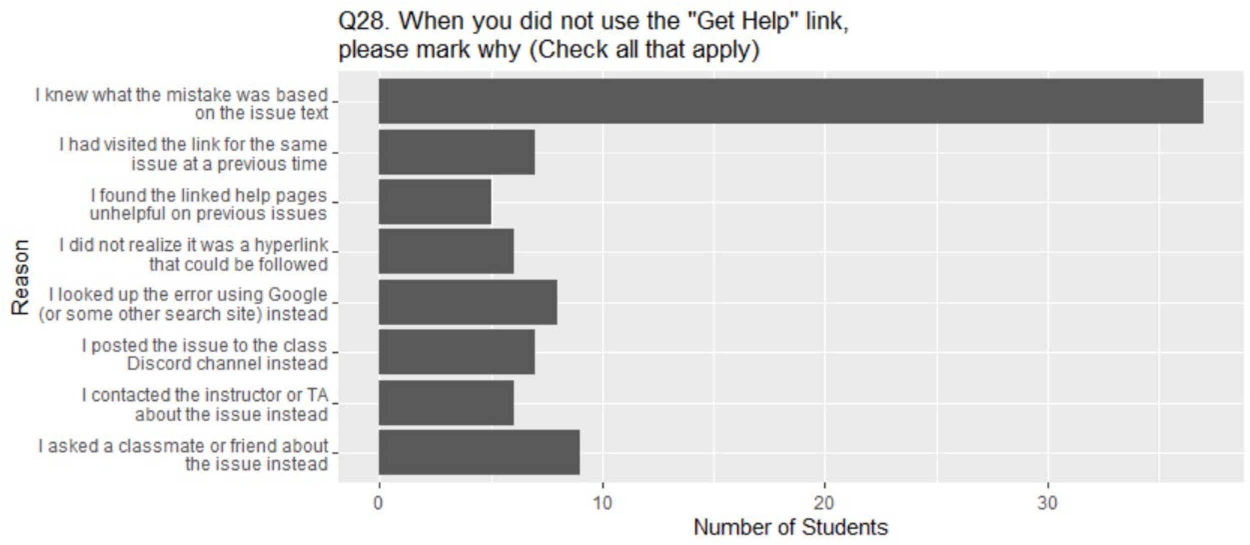


Figure 4.17

Students' Reported Reasons for not Following the 'Get Help' Link in a Pendant Validation



The next question was another free-text prompt allowing students to explain their reasons for not using the 'Get Help' link in Pendant. The responses were qualitatively coded and the results are presented in Table 4.4 below.

Table 4.4

Response Codes from Student Explanations about Using the 'Get Help' Link in Pendant

Count	Code
3	Pendant was helpful when needed
1	The link sometimes took you to the wrong code section*
1	I recognized errors I had made previously
1	The error names/messages were self-explanatory
1	The provided help was too generic

1	Pendant had some bugs
1	Pendant was a good tool

Note: The response marked with an asterisk (*) seems to be referring to the 'Show Location' link, not the 'Get Help one.

The next series of questions concerns the 'Show Location' link, which when followed displayed the location of the issue discovered by the Code Analyzer in a webpage generated by GitHub. An example of what this looked like was presented earlier in this chapter as Figure 3.11. The student responses to these questions appear as Figure 4.18 and Figure 4.19 below.

Figure 4.18

Students' Reported Frequency of Following the 'Show Location' Link

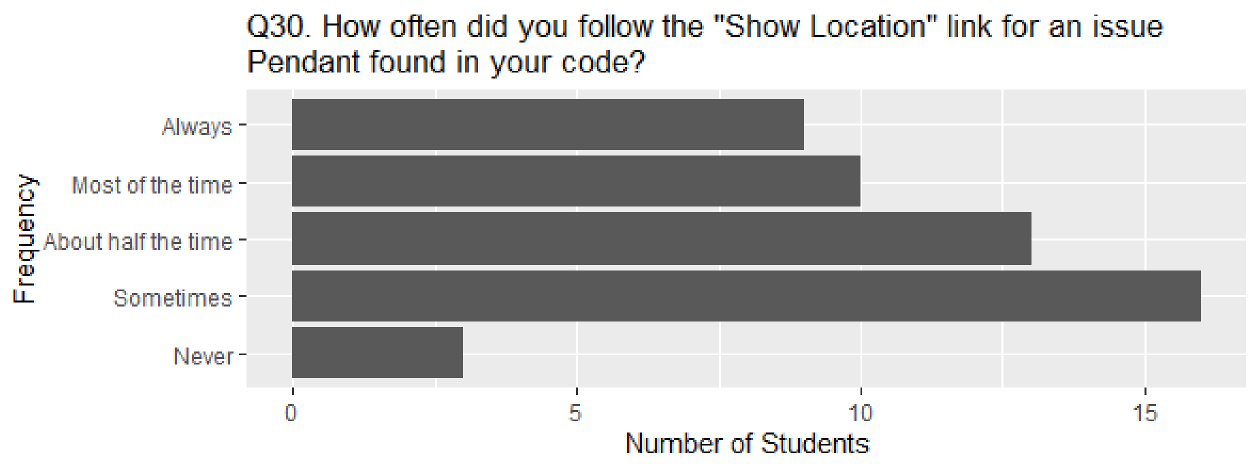
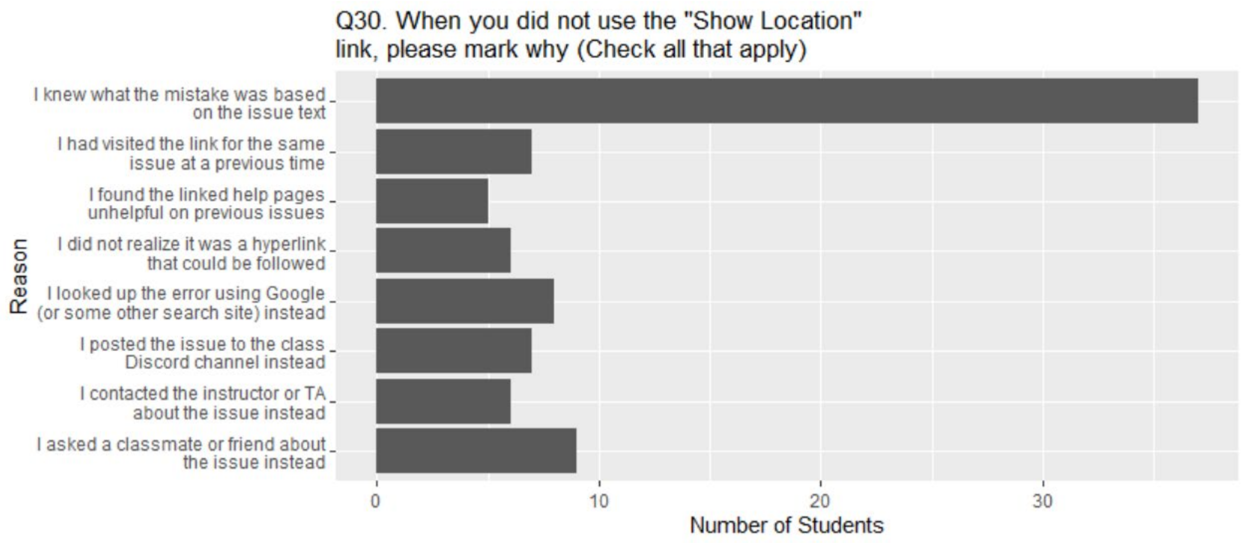


Figure 4.19

Students' Reported Reasons for Not Following the 'Show Location' Link



The final question of this series was a free-text explanation of how the students used the 'Show Location' link. The responses were qualitatively coded and the results are presented in Table 4.5 below.

Table 4.5

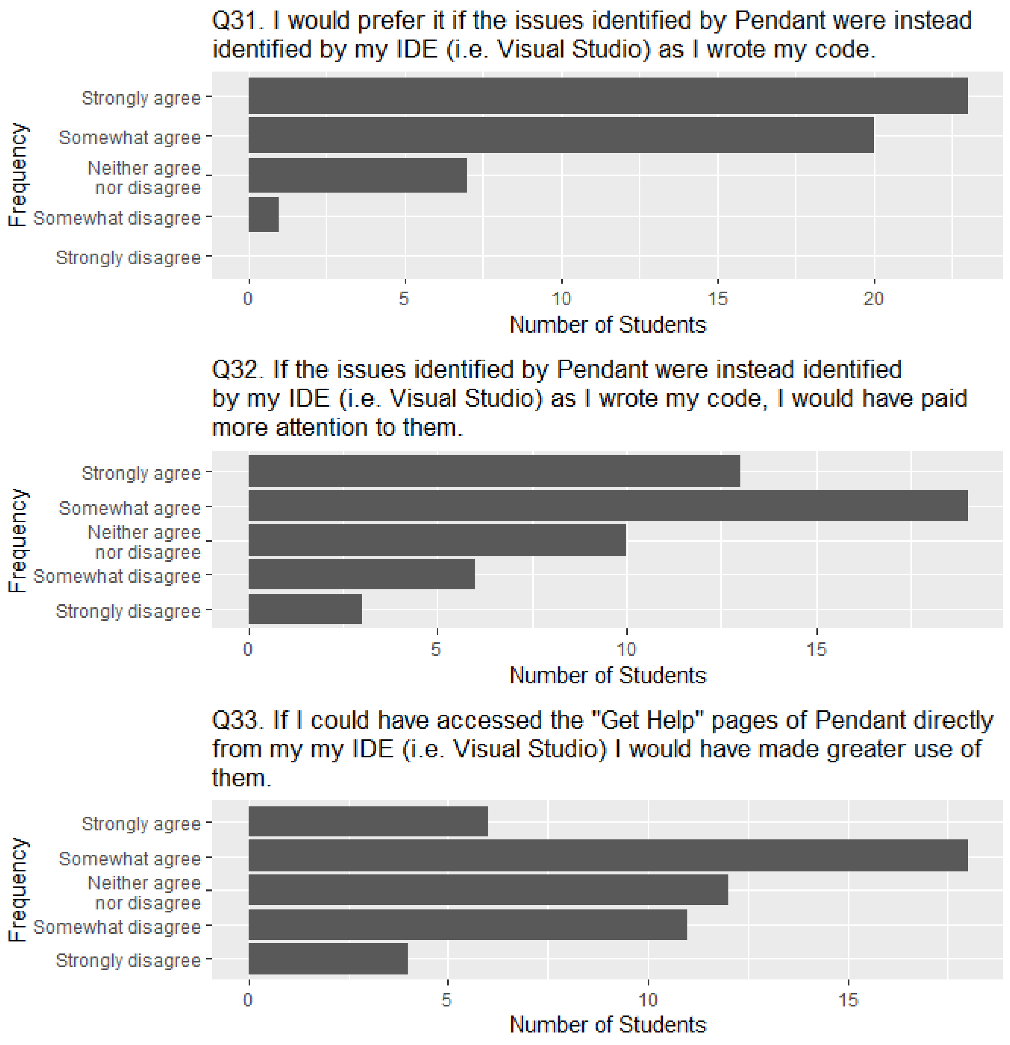
Response Codes from Student Explanations About Using 'Show Location' Link in Pendant

Count	Code
3	Helpful for locating and fixing issues
2	Really liked this feature
1	Would like the link to open a new browser tab
1	Occasionally opened the wrong location
1	Sometimes was out of sync with current changes to code

The final series of questions asked students about integrating Pendant features directly into their integrated development environment (i.e., Visual Studio), and if they would make more use of them if they were available in that environment. The responses to these questions are provided in Figure 4.20 below.

Figure 4.20

Students' Reported Willingness to use Pendant Features if Provided by their IDE

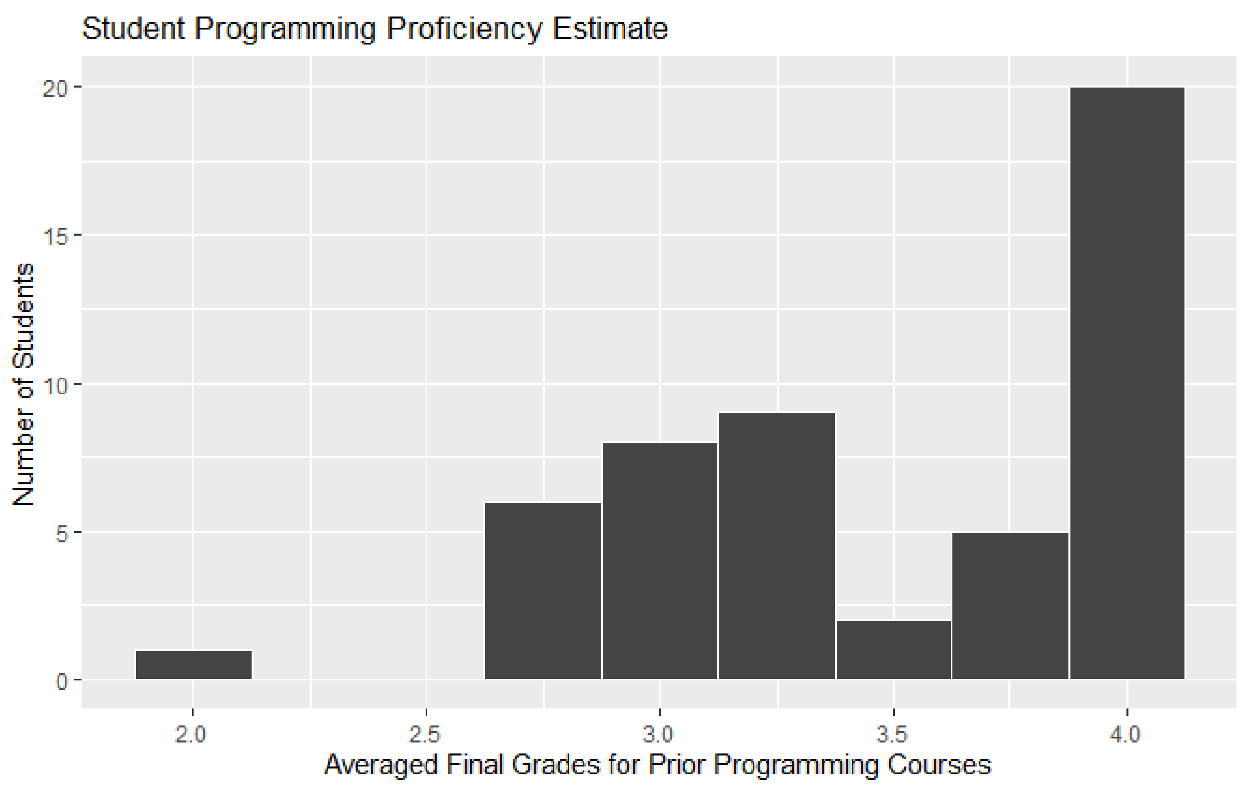


Disaggregating Survey Results

In addition to investigating how all students interacted with the Pendant tool, I was especially interested in how students of different proficiency levels engaged with it. As an estimated measure of proficiency, the students' prior programming course grades (CIS 115, CIS 200, and CIS 300; corresponding to the common CS0, CS1, CS2 course progression) were averaged (*M* = 3.5, *SD* = 0.5), the distribution of which is displayed visually in Figure 4.21.

Figure 4.21

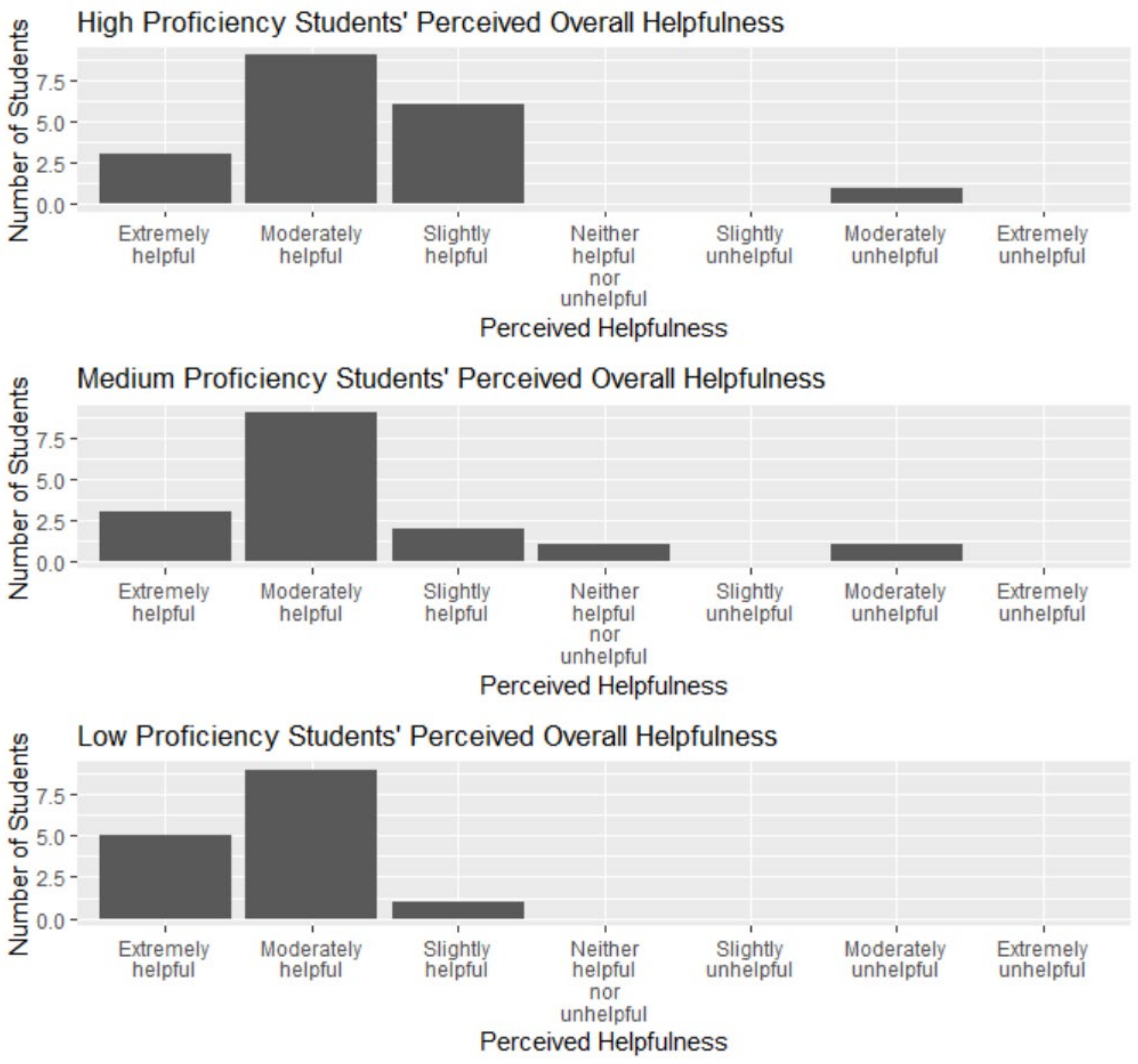
Students' Averaged Final Grades from Prior Programming Courses



Based on this proficiency estimate, students were sorted into three categories: High (those with an average grade of 4.0, *N* = 20), Medium (those between 3.0 and 4.0, *N* = 16), and Low (those with 3.0 and below, $N = 15$). Re-examining how useful students within these groups perceived the Pendant tool reveals a stronger positive skew for low-proficiency students, as can be seen in Figure 4.22 below.

Figure 4.22

Students' Perceived Helpfulness of the Pendant Tool, Disaggregated by Proficiency Level



Effects of Pendant Tool Use

Of course, beyond the question of how useful students perceived the Pendant tool to be is the question of how it impacted their learning. As discussed before, it was not possible to use a quasi-experimental comparison between the students of two different course offerings as originally planned. But we can look to the degree that students within the study used the Pendant tool, as represented by the total number of validations they generated. In doing so, it is reasonable to use the students' final exam scores as a measure of what they have learned in the course, as the final is comprehensive and focused on the course's specified learning outcomes. We would expect the students' prior learning in the subject, represented by the students' averaged prior programming course grades would correlate with this measure, which it does, $r(49) = 0.41$.

To investigate this relationship in the context of Pendant tool usage, I separated the participating students into quartiles based on the total number of validations they generated. The descriptive statistics for the quartiles can be found in Table 4.6, below.

Table 4.6

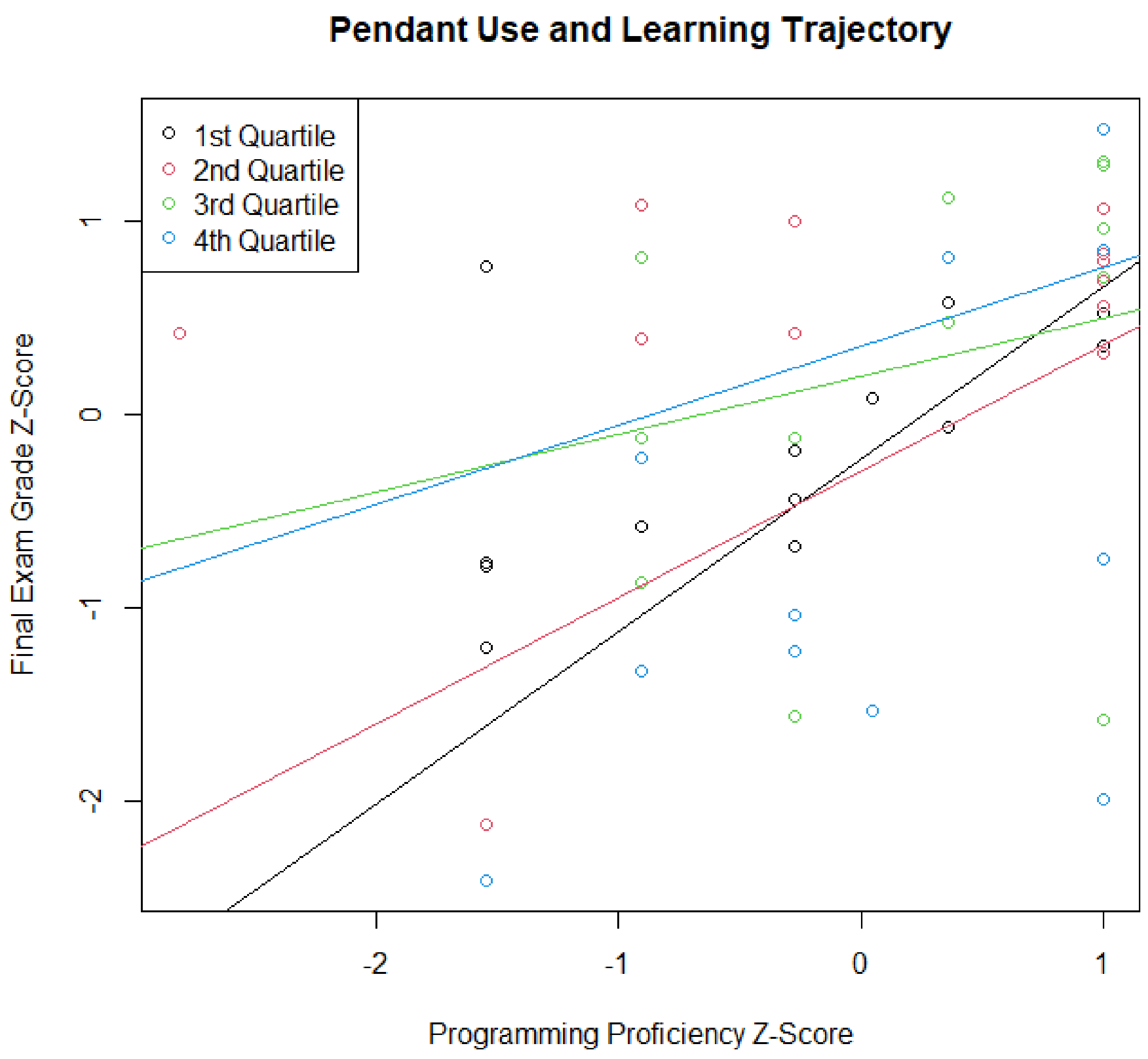
Details for Student Quartiles by Total Validations

Quartile	N	Validation Statistics
1st	13	$M = 15.5$, $SD = 5.2$
2nd	13	$M = 26.2$, $SD = 2.7$
3rd	13	$M = 39.3$, $SD = 4.9$
4th	12	$M = 68.3$, $SD = 32.6$

Plotting the data points for the quartiles and determining the line of best fit for each reveals some interesting results, as seen in Figure 4.23.

Figure 4.23

The Programming Proficiency vs. Final Exam Score Influenced by Pendant Use (In Quartiles).



The quartiles that used the tool least (1st and 2nd) show a nearly 1:1 slope with a y-intercept near (0, 0). This indicates that for these students, the z-score for their final exam score is a near-match for their z-score for prior programming proficiency. In other words, students who were more proficient in programming at the start of the course scored higher on the final, and those who were weaker on programming entering the course scored lower on the final.

But the slope for the quartiles that used the tool more (3rd and 4th) is flatter and the y-intercept is higher. This means that for students entering the course with a lower programming proficiency, use the Pendant tool corresponded to an improved final exam score. This finding suggests use of the Pendant tool helped improve their learning of the subject. However, as all four lines draw near point on the left side of the graph, students who were already more proficient did not see this learning benefit.

To evaluate the significance of these linear models, a one-way ANCOVA was conducted comparing the final exam scores for the four quartiles while controlling for prior programming proficiency. Levene's test and normality checks were carried out and assumptions met. There was a significant effect for the quartile after controlling for the effect of the prior programming proficiency $F(1,3)=3.52$, $p<0.05$. Post hoc tests showed the only significant difference between groups was between quartiles 4 and 2. Quartile 4 represents the most frequent Pendant users, creating between 35 and 101 validations, while quartile 2 created 23 to 26 validations – roughly 3 per milestone.

Summary

In this chapter, we reviewed the results of the artifact evaluation effort. This included reporting the usage data automatically collected by the Pendant tool, reporting the results of a survey given to the participants on their usage of the Pendant tool, and an analysis of the students’ final grades in relation to their use of the Pendant tool and programming proficiency entering the course.

Chapter 5 - Conclusions

This chapter details my conclusions based on the outcomes from the study. The four research questions and the data sources are shown in Figure 5.1.

Table 5.1

Research Questions and Data Sources

Research question	Data source
1. Can the proposed system for providing targeted contextual instruction be built with the software development tools currently available?	Student daily work and milestone programming projects, exams, instructor, and TA notes from Fall 2019 to Spring 2021
2. Can such a system effectively provide targeted contextual instruction corresponding to the difficulties the student is experiencing?	Student survey with Fall 2021 cohort, automated data collection by the artifact, notes from Instructor and TA
3. Can such a system be easily integrated into existing postsecondary programming instructional practice?	Instructor notes from the Fall 2021 semester
4. Will students perceive the system as useful for aiding in their learning process and achieving their learning goals?	Student survey with the Fall 2021 CIS 400 cohort

Research Question 1

Essentially, research question 1 asks if it would be possible to build the proposed artifact. Considering that *two* prototype systems (NuGet and Pendant) were developed over the process of this study and were utilized by two sizeable (56 in Spring 2021 and 60 in Fall 2021) student bodies, I would say the answer is a definite ‘yes.’ The two prototypes also help suggest different potential approaches to utilizing code analysis in an educational context.

Deploying code analyzers using a NuGet package allows for integration into Visual Studio at a fundamental level. From a student perspective, the errors and warnings reported by a custom code analyzer are largely indistinguishable from those reported by Visual Studio itself. Thus, it requires no additional learning or techniques on the part of the student and can be used as soon as the student starts working with the development environment.

In contrast, the Pendant application represents a more involved approach, as the student must incorporate additional steps into their design process – the use of git feature branches and webhooks as described in Chapter 3 (and visually expressed in Figure 3.9). While these are good practices to learn, it also represents additional complexity and cognitive load for the student. We can see this in the twelve students who reported finding merging feature branches “somewhat difficult” in response to survey question 11. As such it is probably not appropriate for beginning students in a first or second programming course where developing their basic programming skills must take priority. However, it *does* add the potential for verifying the students’ ability to program against a specification, as it was capable of verifying the student followed a structural and functional specification. Thus, it is especially suitable for mid-level programming courses, which often seek to prepare students to develop code against a specification provided by a software architect. However, the approach becomes less useful as students transition to developing their own software architectures and specifications.

A third variant suggested by the Pendant tool and how many students used it is to utilize the same techniques in an automated grading and feedback system. Rather than using a webhook triggered by milestone branches, a LMS (Learning Management System) plugin could be authored to process the student's submission and provide a near-immediate grade and feedback for the student. Moreover, it could be set to allow the student to resubmit, encouraging them to both fix their errors (potentially learning from their mistakes) *and* start the assignment in a timely manner, allowing for the additional time to address issues and resubmit. This approach encounters the same limitations described in the previous example, that it ceases to make sense once students are no longer programming against a specification. But until that point, it could prove an effective approach to getting students timely grades and feedback, especially as a class size grows.

Research Question 2

The second research question is essentially asking did the artifact do what it was intended to do – provide instructional feedback based on the issues a student is struggling with? The answer here is more ambiguous. It is clear issues were being detected in the student code – the 40,714 issues detected in the semester are a testament to this fact. However, the instructional content associated with these issues was only referenced a total of 89 times by the 52 participants in the study – that's an average of 1.7 lookups per participant.

This low number is one of the primary reasons for the survey question asking for reasons students did not follow the 'Get Help' link provided by Pendant. A large percentage of students indicated the reason was often that the name of the reported error gave them a strong enough clue to recognize their mistake (72.5%).

A not inconsiderable number of students turned to other resources instead: a classmate (37.2%), the instructor (21.5%), Google (19.6%), or the class Discord channel (11.7%). A possible explanation arises when comparing the percentage of students who reported Pendant's help pages (15.6%) and Visual Studio's help pages (27.4%) unhelpful in past experiences. It is not surprising that Visual Studio's help pages, written for an audience of professional programmers, would be less helpful to novice programmers. But these early experiences (as these students had been using Visual Studio for at least a semester before taking CIS 400) may have taught them to seek out help from peers and instructors, rather than reading documentation. If this is indeed the case, replacing the Visual Studio help links with ones more pedagogically grounded could improve these numbers across the board.

Research Question 3

The third research question addresses the need for tools to be easily integrated into teaching practice. It is also answered by our experiences with the two prototypes that were the artifacts developed in this study.

In the Fall of 2020, the code diagnostics were made available through a NuGet package the students could install into their project in Visual Studio. These provided (through Visual Studio's built-in Error List), a hyperlink to a static website that presented the tutorial material. This approach is straightforward and meshes well with common instructional practices, as students are already taught to utilize the error list in Visual Studio when determining what issues exist in their code. For assignments that begin with a starter project (a partially complete program provided as a form of instructional scaffolding), the process is even less invasive – the instructor can add the NuGet package directly to the starter project, and the students do not need to do anything additional to receive the benefits.

In the Spring 2021 semester we used the Pendant web tool. Adding Pendant to a course was more work. Additional instruction in the use of both Git branches and webhooks had to be added to the course content, and students had to make use of these tools to gain the benefits of the validations. However, both of these tasks aligned with curricular goals for the course. As a sophomore-level course, CIS 400 is well-positioned for introducing and making effective use of feature branches. Setting up webhooks is a one-time activity, so it was easy to bundle into an assignment. It is something the students should have at least a passing familiarity with.

The additional tasks required to implement Pendant effectively – writing the structural specification and testing suite – did add additional overhead but were ancillary to the code analysis checking for design issues. Additionally, these were steps towards an auto-grading process that ultimately reduces the overall workload for the instructor and teaching assistants, freeing them for more valuable face-to-face interactions with students.

Thus, assuming the code analysis tools and supporting tutoring documentation are created and shared as a NuGet package and static website, there is little additional work required to integrate it into an existing course. If the additional features offered by the Pendant tool are desired (the ability to analyze program structure against an assignment specification, implanting an auto-grader, etc.), then significantly more work is required to integrate the tool, with potentially gaining back some time that would otherwise have been spent grading.

Research Question 4

This brings us to the final, and perhaps most important question of all, *Will students perceive the system as useful for aiding in their learning process and achieving their learning goals?* This was broadly addressed in the first question of the post-course survey, *Overall, how helpful or unhelpful did you find the Pendant Tool?* the results of which appeared in Figure 4.3. Most students (89%) responded that they found the tool helpful to some degree. A related question asked students to rate how likely they were to recommend Pendant to other students on a scale of 0 to 10; the average response was 7.1 (*SD = 2.5*). Finally, the survey contained a series of questions that sought to determine how that usefulness translates to specific student learning goals, reported in Figure 4.4. Interestingly, each of these responses skew strongly towards higher perceived usefulness with the sole exception of learning about and fixing design issues.

These results suggest that a majority of the participants perceived the Pendant tool as useful. However, the discrepancy around perceived usefulness of detecting design issues in the students' code bears a closer look. An analysis of how the students were actively using the Pendant web tool may explain the discrepancy. Several questions in the survey addressed this by asking when students created their branch (Question 9, reported in Figure 4.6), committed to that branch (Question 10, also reported in Figure 4.6), and the strategies they employed for using the Pendant Tool (Question 23, reported in Figure 4.13).

Pendant validations are created when the student commits code to a feature branch. With that in mind, if a student does not commit their changes until they feel they have completed the milestone, they <u>should</u> have already addressed any design issues. A majority of the students were either committing only at the end of a coding session (41%), which implies they probably did not look at the validations at that point, or when they felt they had completed the milestone (29%). In other words, they were using Pendant as a final quality check, not a tool to support development.

This is not good design practice – we want students to commit after every major code change, ideally multiple times in a coding session (just as you would save an important document multiple times in an editing session). Even more disappointing is that a number of students did not create their feature branches before beginning to work on the assignment, either delaying until much code had been written (7%) or until they believed they were done with the assignment (16%).

Because so many students delayed the use of validations, they often had to do significant work to fix issues that would have been detected early on. This could lead to significant frustration. For example, one of the students who found pendant *moderately unhelpful* reported, “Around halfway through the semester, every milestone I would have to refactor my code to make Pendant happy such as not using **nameof()** to get property strings in getters.” The frustration in this statement reveals a common tension students have for adopting best practices – if the program already works, why is the extra work necessary? In this case, the use of **nameof()** allows the static type checker to verify the property names, avoiding issues where a student misspells a property name as a string, i.e., “Calores” for “Calories,” which introduces a subtle error where a data-bound GUI will not update when the bound **Calories** property changes.

Finally, some students struggled with the process of using Git and milestone branches, as can be seen in Figure 4.5. Interestingly, students reported less difficulty in the creation of a release (also appearing in Figure 4.5). This likely reflects the value students placed on learning these procedures. Creating a release was required to turn in the milestone assignment, while using milestone branches was highly encouraged but not strictly required (i.e., the students were not graded on it). Students tend to prioritize learning tasks that are reflected in grades, so for some learning the use of feature milestones, even with the potential benefits of validations, was not worth the effort.

Impact on Student Learning

While not specifically addressed in the research questions which focused on the feasibility of using code analysis for targeted supplemental instruction, it is natural to ask the question ‘does it help students learn?’ While my ability to carry out an analysis to answer this question was limited, it does indeed seem that for struggling students, it can have a positive impact on their learning, based on the linear models discussed at the end of Chapter 4. That said, we should be careful in ascribing too much belief in this one finding, as statistical significance was only found between one pair of quartiles, and the relationship between prior programming proficiency and final grades was not particularly strong.

That said, this finding certainly fits into the expectations created by the Developmental Epistemology of Programming theory proposed by Lister (2016). The theory posits a less proficient student has less-developed schema for reasoning about aspects of writing a program, and therefore encounters a higher cognitive load as they work through programming tasks. The Pendant tool provides some relief from this cognitive load, allowing a student to dismiss some concerns (such as documenting and naming conventions) until after they have completed the program logic – an approach that it is clear many students were taking based on the answers to survey questions 23 and 10.

However, the more important factor that may be in play is that the analysis also was able to find issues with students' code that reflect a fundamental *misunderstanding* of how the program works. Take the **INotifyPropertyChanged** interface, which when implemented on a class indicates the class needs to send **PropertyChanged** events whenever the value of one of the class's properties change. While it is covered in both lecture and in the tutorial assignments, students in the course continue to struggle with understanding this requirement when writing milestones. Some common issues include 1) attempting to add **set** methods to properties derived from other properties, creating a **set** that is never used, and 2) attempting to add the firing of the **PropertyChanged** event in a **get** method, which creates a recursive cycle similar to the Property Self-Reference issue discussed in Chapter 3, or 3) simply not firing the **PropertyChanged** event at all.

The Pendant tool was able to catch each of these cases and provide a detailed discussion of how the students' approach would cause problems in their code, as well as setting out examples of how to properly handle the **PropertyChanged** event. This is exactly the kind of instruction I would normally offer a student who encountered this issue in class – but now the student could benefit from it *outside* of class, at the exact moment they were working on the assignment and encountered the problem. This learning experience could be further augmented with video recordings and interactive coding exercises to help the students more firmly grasp the concept at a time they are most receptive to it – when they realize they need a stronger understanding.

Conclusions

Based on the findings of this study, the most impactful approach seems to be creating a pedagogically focused NuGet Code Analysis package and hosting the associated tutoring content on a static website. This allows anybody learning C# to benefit from targeted error detection and supplemental instruction with a minimum of effort – just installing the package on a project they are working on. Similarly, instructors can recommend it to their students or integrate it directly into starter projects, making it easy to integrate into any C# using course. Releasing the source code for the NuGet package and supplemental instruction content as an open-source project on GitHub would also allow for contributions from other teaching and programming professionals, potentially expanding the number of issues that can be handled by the system.

The web tool approach embodied in the Pendant application requires significantly more investment to set up with a course and the use of webhooks and feature branches is appropriate for only a narrow band of courses. However, if the submission mechanism was changed – perhaps integrated directly into the courses' learning management system as a custom assignment type – then this approach is usable for a broader range of beginning programming courses. When implemented this way, the web tool becomes an auto-grader that can provide meaningful and timely feedback to students. It also opens the possibility of allowing students to correct and resubmit programs that contained errors, allowing them to both see and learn from these mistakes – adding a layer of robustness and recursion to the curriculum that is normally not possible due to the need for hand grading.

Finally, integrating a plugin into Visual Studio beyond the basic functionality of a Code Analyzer does not seem a viable path. If I were to start over, I would instead target VS Code, Microsoft's lightweight, open-source IDE. VS Code uses a language service approach to providing a rich editor experience, which allows for far greater flexibility of implementation. Beyond the ability to integrate more functionality into a pedagogical extension, VS Code is not burdened with the same degree of 'productivity-enhancing' features as Visual Studio. These features are intended to support the professional programmer, but for a novice programmer they often provide too much assistance, short-circuiting the learning process. A prime example is the *code completion* feature – where the IDE will automatically complete a code statement the programmer begins to write. While this can speed up the development process, for a student learning programming it means they do not need to learn the full syntax for anything code completion can finish for them. And this is very evident in paper tests, where many students will start a statement and either add an annotation 'I don't know what comes after this' or just guess blindly. The problem will likely be even more pronounced in Visual Studio 2022, as it incorporates AI assisted code completion (Boucher et al., 2022).

Future Work

Now that we know these kinds of tools can be built and will be utilized by students, it makes sense to expand the approach to detect more student errors and provide more targeted feedback. This could include writing code analyzers that mimic those that already exist in Visual

Studio to send students to pedagogically grounded help webpages instead of the professional programmer-centric ones provided by Microsoft. This could be especially helpful for novice programmers in earlier courses who are just starting to learn C#. This could also provide a rich context for investigating the impact of this kind of tool on student learning.

While this study focused on the C# language, code analysis is possible in any language. The exact approach will need to be different, as this study utilized the code analyzers supported by the Roslyn compiler that have access to the compiled abstract syntax tree and symbol table. Similar support can be built into other compilers, or separate analysis tools like ANTLR could be leveraged to fulfill this role.

References

Ahmed, U. Z., Kumar, P., Karkare, A., Kar, P., & Gulwani, S. (2018). Compilation Error Repair: For the Student Programs, From the Student Programs. *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training* (pp. 78-87). New York, NY: ACM.

Aho, A. V., & Ullman, J. D. (1977). *Principles of Compiler Design.* Reading, MA: Addison-Wesley Publishing Company.

Atkins, D. E., Droegemeier, K. K., Feldman, S., Garica-Malona, H., Klein, M. L., Messerschmitt, P. M., . . . Wight, M. H. (2003). *Revolutionizing Science and Engineering through Cyberinfrastructure. Report of the National Science Foundation blue-ribbon advisory panel on cyberinfrastructure.* National Science Foundation.

Barr, D., Harrison, J., Conery, L. (2011). Computational thinking: A digital age skills for everyone. *Learning & Leading with Technology, 38*(6), 20-23.

Basawapatna, A. R., Koh, K. H., & Repenning, A. (2010). Using scalable game design to teach computer science from middle school to graduate school. *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education* (pp. 224-228). New York, NY: ACM.

Basawapatna, A., Koh, H. K., Repenning, A., Webb, D. C., & Marshall, K. S. (2011). Recognizing computational thinking patterns. *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education* (pp. 245-250). New York, NY: ACM.

Basawapatna, A., Repenning, A., & Koh, K. H. (2015). Closing the cyberlearning loop: Enabling teachers to formatively assess student programming projects. *SIGCSE '15: Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 12-17). New York, NY: ACM.

Ben-Ari, M. (2001). Constructivism in computer science eduation. *Journal of Mathematics and Science Teaching, 1*(45), 45-73.

Bennedsen, J., & Capersen, M. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin, 39*(2), 32-36.

Berges, M., & Hubwieser, P. (2015). Evaluation of source code with item response theory. *ITiCSE '15: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education.* New York, NY: ACM.

Boucher, F., Mads, K., & Downie, M. (2022, 11 17). *Let's Explore Visual Studio 2022*. Retrieved from Microsoft Docs > Hello World: https://docs.microsoft.com/en-us/shows/hello-world/hello-world-s2-lets-explore-visual-studio-2022

Brown, N. C. (2020). BlueJ 5: Still Going Strong. *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (p. 1420). New York, NY: ACM.

Brown, N. C., Kölling, M., McCall, D., & Utting, I. (2014). Blackbox: A large scale repository of novice programmers' activity. *IGCSE '14: Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 223-228). New York, NY: ACM.

Bureau of Labor Statistics, U.S. Department of Labor. (2014a). *Occupational employment statistics*. Retrieved March 6, 2015 from https://www.bls.gov/oes/

Bureau of Labor Statistics, U.S. Department of Labor. (2014b, January 8). *Computer and Information Technology Occupations.* Occupational Outlook Handbook. Retrieved March 6, 2015, https://www.bls.gov/ooh/computer-and-information-technology/home.htm

Bundy, A. (2007). Computational thinking is pervasive. *Journal of Scientific and Practical Computing*, 1(2), 67-69.

Camp, T., Adrion, R., Bizot, B., Davidson, S., Hall, M., Hambrusch, S., . . . Zweben, S. (2017a, June). Generation CS: The Growth of Computer Science. *ACM Inroads, 8*(2), 44-50.

Camp, T., Adrion, W. R., Bizot, B., Davidson, S., Hall, M., Hambrusch, S., . . . Zweben, S. (2017b, September). Generation CS: The Mixed News on Diversity and the Enrollment Surge. *ACM Inroads, 8*(3), 36-41.

Camp, T., Adrion, W. R., Bizot, B., Davidson, S., Hall, M., Hambrusch, S., . . . Zweben, S. (2017c, December). Generation CS: the challenges and responses to the enrollment surge. *ACM Inroads, 8*(4), 59-65.

Clear, T., Edwards, J., & Lister, R. (2008). The Teaching of Novice Computer Programmers: Bringing the Scholarly-Research Approach to Australia. *Proceedings of the Tenth Conference on Australian Computing Education (ACE '08)* (pp. 63-68). New York, NY: ACM.

De Ruvo, G., Tempero, E., Luxton-Reilly, A., & Rowe, G. B. (2018). Understanding Semantic Style by Analyzing Student Code. *Proceedings of the 20th Australasian Computing Education Conference (ACE '18)* (pp. 73-82). New York, NY: ACM.

Dingle, A. (2001). Assessing the Ripple Effect of CS1 Langauge Choice. *Proceedings of the Second Annual Conference on Computing in Small Colleges Northwestern Conference (CCSC '01)* (pp. 85-94). New York, NY: ACM.

Doll, W. E. (1993). *A Post-modern Perspective on Curriculum.* New York, NY: Teachers College Press.

Engelbart, D. (1962). *Augmenting Human Intellect: A Conceptual Framework.* Menlo Park, CA: SRI International.

Guzdial, M. (2004). Programming environments for novices. In S. Fincher & M. Petre (Eds.), *Computer science education research* (pp. 127-154). RoutledgeFalmer.

Guo, P. J. (2013). Online Python Tutor: Embeddable Web-Based Program Visualizations for CS Education. *Proceedings of the 44th ACM Technical Symposium on Computer Science Education* (pp. 579-584). New York, NY: ACM.

Gürer, D. (2002). Women in Computing History. *SIGCSE Bulletin, 34*(2), 116-120.

Hlebowitsh. (2021, March 25). *Ralph Tyler, the Tyler Rationale, and the Idea of Educational Evaluation*. Retrieved from Oxford Research Encyclopedias: https://oxfordre.com/education/view/10.1093/acrefore/9780190264093.001.0001/acrefore-9780190264093-e-1036

Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and Correcting Java Programming Errors for Introductory Programming Students. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '03)* (pp. 153-156). New York, NY: ACM.

Institute of Education Sciences. (2013). *Common Guidelines for Education Research and Development.* Washington DC: National Science Foundation.

Johannesson, P., & Perjons, E. (2014). *An Introduction to Design Science.* Cham: Springer.

Joint Task Force on Computing Curricula, ACM & IEEE. (2013). *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degrees in Computer Science.* New York, NY: ACM.

Karvelas, I., Li, A., & Becker, B. A. (2020). The Effects of Compilation Mecahnisms and Error Message Presentation on Novice Programmer Behavior. *Proceedings of the 51st ACM Technical Syposium on Computer Science Education (SIGCSE '20)* (pp. 759-765). New York, NY: ACM.

Keuning, H., Heeren, B., & Jeuring, J. (2019). How Teachers Would Help Students to Improve Thier Code. *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education* (pp. 119-125). New York, NY: ACM.

Koh, H. K., Basawapatna, A., Bennett, V., & Repenning, A. (2010). Towards the automatic recognition of computational thinking for adaptive visual language learning. *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 59-66). Washington DC: IEE.

Koh, K., Basawapatna, A., Bennett, V., & Repenning, A. (2010). Towards the Automatic Recognition of Computational Thinking for Adaptive Visual Language Learning. *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing.* Washington DC: IEEE Computer Society.

Kŏlling, M., Quig, B., Patterson, A., & Rsoenburg, J. (2003). The BlueJ System and its Pedagogy. *Computer Science Education, 13*(4), 249-268.

Larson, J., & Musser, J. (2021). PropertyChangedAnalyzers. DotNetAnalyzers. Retrieved from https://github.com/DotNetAnalyzers/PropertyChangedAnalyzers

Lifelong Kindergarten Group. (2020, 4 27). *Statistics*. Retrieved from Scratch: scratch.mit.edu/statistics

Lin, H. S. (2010). *Report of a workshop on the scope and nature of computational thinking.* Washington DC: National Research Council.

Lister, R. (2011). Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer. *Proceedings of the Thirteenth Australiasian Computing Education Conference (ACE 2011).* New York, NY.

Lister, R. (2016). Toward a Developmental Epistemology of Computer Programming. *Proceedings of the 11th Worshop in Primary and Secondary Computing Education (WiPSCE 16)* (pp. 5-16). New York, NY: ACM.

Lister, R., Fidge, C., & Teague, D. (2009). Further Evidence of a Relationship Between Explaining, Tracing and Writing Skills in Introdutory Programming. *Proceedings of the 14th annual ACM SIGCSE conference in Innovation and Technology in Computer Science Education* (pp. 161-165). New York, NY: ACM.

Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not Seeing the Forest for teh Trees: Novice Programmers and the SOLO Taxonomy. *Proceedings of the 11th annual SIGCSE conference on Innnovation and Technology in Computer Science Education (ITICSE '06)* (pp. 118-112). New York, NY: ACM.

Lu, J. J., & Fletcher, G. H. (2009). Thinking about computational thinking. *ACM SIGCSE Bulletin*, 6-10.

Ma, L., Ferguson, J., Roper, M., Ross, I., & Wood, M. (2009). Improving the mental models held by novice programmers using cognitive conflict and Jeliot visualizations. *ITiCSE '09: Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education* (pp. 166-170). New York, NY: ACM.

McCraken, J., Almstrum, V., Diaz, D., Thomas, L., Guzdial, M., Utting, I., & Hagan, D. (2001). A multi-national, multi-institutional study of assessment of programmign skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4), 125-180.

Mislevy, R. J. (1993). Foundations of a new test theory. In N. Frederiksen, & R. J. Mislevy, *Test Theory for a new Generation of Tests* (pp. 1-39). Hillsdale, NJ: Lawrence Erlbaum Associates.

Mühling, A., Ruf, A., & Hubwieser, P. (2015). Design and first results of a psychometric test for measuring basic programming. *Proceedings of the Workshop in Primary and Secondary Computing Education* (pp. 2-10). New York, NY: ACM.

National Academies of Sciences, Engineering, and Medicine . (2018). *Assessing and Responding to the Growth of Computer Science Undergraduate Enrollments.* Washington, DC: The National Academies Press.

National Research Council. (2000). *How People Learn: Brain, mind, experience and school: Expanded Edition.* Washington, D.C.: National Academies Press.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive Load Theory and Instructional Design: Recent Developments. *Educational Psychologist, 38*(1), 1-4.

Papert, S. (1980). *Mindstorms.* New York, NY: HarperCollins.

Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2nd Edition ed.). Raleigh, NC: Pragmatic Bookshelf.

Piaget, J. (1970). *Genetic Epistemology.* New York, NY: Columbia University Press.

Repenning, A., Owen, R. B., Smith, C., & Repenning, N. (2012). AgentCubes : Enabling 3D creativity by addressing cognitive and affective programming challenges. *Proceedings of the World Conference on Educational Media and Technology, EdMedia 2012* (pp. 2762-2771). Waynesville, NC: AACE.

Resnick, M., Maloney, J., Monroy-Hernández, A., Brennan, K., Millner, A., Rosenbaum, E., . . . Kafi, Y. (2009). Scratch: Programming for All. *Communications of the ACM*, 60-67.

Smith, M. (2016, 1 30). *Computer Science For All.* Retrieved from The White House: https://www.whitehouse.gov/blog/2016/01/30/computer-science-all

Sorva, J. (2013). Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE), 13*(2), 8:1-8:31.

Sudol, L. A., & Studer, C. (2010). Analyzing test items: using Item Response Theory to Validate Assessments. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education - SIGCSE '10* (pp. 436-440). New York, NY: ACM.

Sunitha, K. (2020). *Compiler Construction.* Delhi, India: Pearson.

Taub, R., Armoni, M., & Ben-Ari, M. (2012). CS Unplugged and Middle-School Students' Views, Attitudes, and Intentions Regarding CS. *Transactions on Computing Education (TOCE), 12*(2), 8:1-8:29.

Teague, D. (2015). *Neo-Piagetian Theory and the Novice Programmer.* Queensland University of Technology.

Traver, V. J. (2010). On Compiler Error Messages: What They Say and What They Mean. *Advances in Human-Computer Interaction*, 1-26.

Vogts, D., Calitz, A., & Grelying, J. (2008). Comparison of the Effects of Professional and Pedagogical Program Development Environments on Novice Programmers. *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer SCientists and Information Technologists on IT (SAICSIT '08)* (pp. 286-295). New York, NY: ACM.

Wagner, B., Mike, F., Naidile-P-N, & Kulikov, P. (2021, 10 12). *C# Coding Conventions*. Retrieved from Microsoft Docs: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions

Watson, C., & Li, F. (2014). Failure rates in introductory programming revisited. *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 39-44). New York, NY: ACM.

Wenzel, M., Wagner, B., Warren, G., Odendaal, W., brianwited, Schonning, N., . . . Cetkovic, I. (2020, 4 26). *Microsoft Docs*. Retrieved from The .NET Compiler Platform SDK: docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/

Wieringa, R. (2010). Design Science Methodology: Principles and Practice. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (pp. 493-494). New York, NY: ACM.

Wilson, C., Sudol, L. A., & Stevenson, C. &. (2010). *Running on Empty: The Failure to Teach K-12 Computer Science in the Digital Age.* New York, NY: ACM.

Wing, J. M. (2006). Computational Thinking. *Communications of the ACM, 49*(3), 33-35.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical, and Engineering Sciences 366*(1881), 3717-3725.

Wing, J. M. (2010). Computational Thinking: What and Why? *Link Magazine*.

Winters, T., & Payne, T. (2006). Closing the loop on test creation. *ACM SIGCSE Bulletin, 38*(1), 169.

Wirth, N. (1983). Program Development by Stepwise Refinement. *Communications of the ACM, 226*(1), 70-74.

Wirth, N. (1985). From programming language design to computer construction. *Communications of the ACM, 28*(2), 159-164.

Wirth, N. (2002). Computing science education: The road not taken. *ITiCSE '02: Proceedings of the 7th annual conference on Innovation and technology in computer science education* (pp. 1-3). New York, NY: ACM.

Appendix A - CIS 400 Syllabus

CIS 400 - Object-Oriented Design, Implementation, and Testing

Class Meeting Times and Locations

- **Section A**: M 1:30pm-3:20pm DUF 1092
- **Section B**: W 1:30pm-3:20pm DUF 1092
- **Section S**: F 1:30pm-3:20pm DUF 1092

Instructor Contact Information

- **Instructor:** Nathan Bean (nhbean AT ksu DOT edu)
- **Office:** DUE 2216
- **Phone:** (785)483-9264 (Call/Text)
- **Website:** https://nathanhbean.com
- **Office Hours:** TBD
- **Virtual Office Hours:** By appointment via Zoom. Schedule a meeting via email or Discord.

Preferred Methods of Communication

- **Chat:** Quick questions via Discord are the preferred means of communication. Questions whose answers may benefit the class I would encourage you to post in the `#cis400` channel, as this keeps a public history your classmates can review. More personal questions should be direct messaged to `@Nathan Bean`.
- **Email:** For questions outside of this course, email to nhbean@ksu.edu is preferred.
- **Phone/Text:** 785-483-9264 *Emergencies only!* I will do my best to respond as quickly as I can.

Prerequisites

- CIS 300

Students may enroll in CIS courses only if they have earned a grade of C or better for each prerequisite to these courses.

Course Overview

A study of concepts and techniques used to produce quality software programs. Object-oriented concepts, models, execution environments, design and testing techniques. Extensive application of these concepts and techniques to the development of non-trivial software programs.

Course Description

This course is focused on helping you to learn the concepts and skills needed to develop high-quality software. Along the way you will have ample opportunities to practice these skills developing non-trivial software projects. These are not the "baby programs" of early CS coursework, but rather applications that could be used in a production environment.

Accordingly, our goal is not just to write software that compiles without errors, but to develop ***well-written and maintainable software***. This goal demands extra attention to design, documentation, and testing. Additionally, we will explore some of the powerful features of the C# Language and the Visual Studio compiler, as well as other professional tools like Git and Visio.

Course Objectives

By the end of this course, we expect each student to be able to:

- Create class definitions that 1) utilize **encapsulation** to organize related data and behavior, 2) prevent uncontrolled internal state changes through **information hiding**, and 3) allow outside code controlled interactions through **message passing**.
- **Specify** expected object behavior and verify it through creating and executing **unit tests** against a class definition.
- Utilize **polymorphism** in the form of **inheritance** to minimize code duplication by moving shared fields and methods to a common ancestor.
- Reason about **dynamic dispatch** to determine what version of a polymorphic function or method will be invoked at runtime.
- Utilize **developer tools** like Visual Studio and Git to develop software reliably and efficiently.

Major Course Topics

- Encapsulation and data hiding
- Message passing
- Polymorphism and inheritance
- Dynamic Dispatch
- Event-based programming
- Testing
- Debugging
- Software Versioning
- Code Modularization

Course Structure

A common axiom in learner-centered teaching is ***"(s)he who does the work does the learning."*** What this really means is that students primarily learn through grappling with the concepts and skills of a course while attempting to apply them. Simply seeing a demonstration or hearing a lecture by itself doesn't do much in terms of learning. This is not to say that they don't serve an important role - as they set the stage for the learning to come, helping you to recognize the core ideas to focus on as you work. The work itself consists of applying ideas, practicing skills, and putting the concepts into your own words.

This course is built around learner-centered teaching and its recognition of the role and importance of these different aspects of learning. Most class periods will consist of short lectures interspersed with a variety of hands-on activities built around the concepts and skills we are seeking to master. In addition, we will be applying these ideas in iteratively building a series of related software applications over the course of the semester. Part of our class time will be reserved for working on these applications, giving you the chance to ask questions and receive feedback from your instructors, UTAs, and classmates.

The Work

There is no shortcut to becoming a great programmer. Only by ***doing the work*** will you develop the skills and knowledge to make your a successful computer scientist. This course is built around that principle, and gives you ample opportunity to do the work, with as much support as we can offer.

Tutorials

Each module will include many tutorial assignments that will take you step-by-step through using a particular concept or technique. The point is not simply to complete the tutorial, but to *practice* the technique and coding involved. You will be expected to implement these techniques on your own in the milestone assignment of the module - so this practice helps prepare you for those assignments.

Milestone Programming Assignments

Throughout the semester you will be building a non-trivial software project iteratively; every week a new milestone (a collection of features embodying a new version of a software application) will be due. Each milestone builds upon the prior milestone's code base, so it is critical that you complete each milestone in a timely manner! This process also reflects the way software development is done in the real world - breaking large projects into more readily achievable milestones helps manage the development process.

Following along that real-world theme, programming assignments in this class will also be graded according to their conformance to coding style, documentation, and testing requirements. Each milestone's rubric will include points assigned to each of these factors. It is not enough to simply write code that compiles and meets the specification; good code is readable, maintainable, efficient, and secure. The principles and practices of Object-Oriented programming that we will be learning in this course have been developed specifically to help address these concerns.

Exams

Over the course of the semester we will have a total of four exams. The primary purpose of these exams is formative; they are intended to help us (me as the instructor and you as the student) evaluate how you are learning the material. Thus, my testing policies may differ greatly from your prior courses.

These exams will cover the vocabulary and concepts we are learning and involve reasoning about object-oriented programming, including some code writing. The purpose of this style of assessment to assess your ability to recognize the problem and conceive an appropriate solution. Hence, you are encouraged to annotate your answers with comments, describing your reasoning as you tackle the problem. Additionally, I will include a "certainty scale" for each question, and would ask that you mark how confident you are in your answer. Doing these extra steps helps me know how well you are grasping the material, and helps both of us to know what concepts and skills may need more work.

The first exam is a *pretest* that is used to help establish your knowledge and readiness coming into the course. You will earn a **100%** for completing this exam, regardless of your correct or incorrect responses.

The second three exams are *midterms*, which cover the content immediately proceeding them. You will have a chance to correct mistakes you have made in thee exams, potentially earning back your full points.

The *final* exam is comprehensive and covers the most important topics and skills we have developed in the course. This is considered a *summative* test (one that measures your mastery of a subject). It will count for twice the number of points as the earlier exams, and you will not have a chance to correct mistakes made on it.

Grading

In theory, each student begins the course with an A. As you submit work, you can either maintain your A (for good work) or chip away at it (for less adequate or incomplete work). In practice, each student starts with 0 points in the gradebook and works upward toward a final point total out of the possible number of points. In this course, it is perfectly possible to get an A simply by completing all the software milestones in a satisfactory manner and attending and participating in class each day. In such a case, the examinations will simply reflect the learning you've been doing through that work. Each work category constitutes a portion of the final grade, as detailed below:

30% - Activities, Tutorials, and Quizzes (The lowest 4 scores are dropped)

36% - Programming Assignment Milestones (3.6% each, 11 milestones total; The single lowest assignment score will be dropped)

34% - Exams (8.5% each, with the final worth double at 17; 4 exams total)

Letter grades will be assigned following the standard scale: 90% - 100% - A; 80% - 89.99% - B; 70% - 79.99% - C; 60% - 69.99% - D; 00% - 59.99% - F

At the end of the semester, for students who have earned a borderline grade (i.e. a 89.98%, which is a B), I will bump their grade to the next highest letter grade based on the student's completion of exam annotations and confidence ratings on exam questions. That is to say, if the example student regularly gives detailed annotations of their thought process, and rates their confidence in their answer, I will bump their 89.98% to an A. Students who do not provide annotations and confidence ratings will not be bumped, for any reason.

Collaboration

Collaboration is an important practice for both learning and software development. As such, you are encouraged to work with peers and seek out help from your instructors and UTAs. However, it is also critical to remember that ***(s)he who does the work, does the learning.*** Relying too much on your peers will deny you the opportunity to learn yourself. As the skills we are working on are the foundations on which all future computer science coursework relies, any skills you fail to develop in this course can have long-ranging effects on your future success, in both classes and the working world.

Determining where the line between good collaboration and over-reliance on others can be challenging, especially as a student. I offer a few guidelines that can help:

1. If you can't yet put a concept into your own words and explain it to someone not versed in programming, you do not yet have a full grasp of the concept. Don't be tempted to use someone else's words - keep working at it until you can use your own. But "working at it" in this context can mean seeking out additional explanations from other people. Sometimes getting enough different perspectives on a concept is what you need to be able to synthesize your own.
2. Directly copying another student's code and turning it in as your own work is never acceptable. It is a form of plagiarism and constitutes academic dishonesty and can result in severe penalties (covered below). This does not mean you can't discuss the assignment and approaches to solving it with your peers - in fact doing so is often a useful learning practice. Just keep those discussions above the code level.
3. As a corollary to point 2, it is okay to ask another student to look at your code when you are struggling with syntax or errors. However, don't let them correct it for you - let them offer suggestions but make any changes yourself. The act of making these changes actually contributes to the stimulus your brain is using to develop programming skills. So don't let others shortchange your opportunity to learn (including instructors and UTAs).

Late Work

Read the late work policy very carefully! If you are unsure how to interpret it, please contact the instructor via email. Not understanding the policy does not mean that it won't apply to you!

Every student should strive to turn in work on time. Late work will receive a penalty of 10% of the possible points for each day it is late. Missed class attendance cannot be made up, though as mentioned above some areas will drop the lowest two scores. If you are getting behind in the class, you are encouraged to speak to the instructor for options to make up missed work.

Software

We will be using Visual Studio 2019 as our development environment. You can download a free copy of Visual Studio Community for your own machine at https://visualstudio.microsoft.com/downloads/. You should also be able to get a professional development license through your Azure Student Portal. See the CS support documentation for details: https://support.cs.ksu.edu/CISDocs/wiki/FAQ#MSDNAA

Additionally, we will create UML diagrams using Microsoft Visio, which can also be downloaded from the Azure Student Portal (see above).

We will use Xamarin workbooks to distribute some content. This free software can be downlaoded from: https://docs.microsoft.com/en-us/xamarin/tools/workbooks/install

Discord also offers some free desktop and mobile clients that you may prefer over the web client. You may download them from: https://discord.com/download.

Recommended Texts & Supplies

To participate in this course, students must have access to a modern web browser and broadband internet connection. All course materials will be provided via Canvas. Modules may also contain links to external resources for additional information, such as programming language documentation.

This course offers an instructor-written textbook, which is broken up into a specific reading order and interleaved with activities and quizzes in the modules. It can also be directly accessed at https://textbooks.cs.ksu.edu/cis400.

Students who would like additional textbooks should refer to resources available on the O'Riley For Higher Education digital library offered by the Kansas State University Library. These include electronic editions of popular textbooks as well as videos and tutorials.

Subject to Change

The details in this syllabus are not set in stone. Due to the flexible nature of this class, adjustments may need to be made as the semester progresses, though they will be kept to a minimum. If any changes occur, the changes will be posted on the Canvas page for this course and emailed to all students.

Academic Honesty

Kansas State University has an Honor and Integrity System based on personal integrity, which is presumed to be sufficient assurance that, in academic matters, one's work is performed honestly and without unauthorized assistance. Undergraduate and graduate students, by registration, acknowledge the jurisdiction of the Honor and Integrity System. The policies and procedures of the Honor and Integrity System apply to all full and part-time students enrolled in undergraduate and graduate courses on-campus, off-campus, and via distance learning. A component vital to the Honor and Integrity System is the inclusion of the Honor Pledge which applies to all assignments, examinations, or other course work undertaken by students. The Honor Pledge is implied, whether or not it is stated: "On my honor, as a student, I have neither given nor received unauthorized aid on this academic work." A grade of XF can result from a breach of academic honesty. The F indicates failure in the course; the X indicates the reason is an Honor Pledge violation.

For this course, a violation of the Honor Pledge will result in sanctions such as a 0 on the assignment or an XF in the course, depending on severity. Actively seeking unauthorized aid, such as posting lab assignments on sites such as Chegg or StackOverflow or asking another person to complete your work, even if unsuccessful, will result in an immediate XF in the course.

Students with Disabilities

Students with disabilities who need classroom accommodations, access to technology, or information about emergency building/campus evacuation processes should contact the Student Access Center and/or their instructor. Services are available to students with a wide range of disabilities including, but not limited to, physical disabilities, medical conditions, learning disabilities, attention deficit disorder, depression, and anxiety. If you are a student enrolled in campus/online courses through the Manhattan or Olathe campuses, contact the Student Access Center at accesscenter@k-state.edu, 785-532-6441; for K-State Polytechnic campus, contact Julie Rowe, Diversity, Inclusion and Access Coordinator, at jarowe@ksu.edu or call 785-826-2971.

Expectations for Conduct

All student activities in the University, including this course, are governed by the Student Judicial Conduct Code as outlined in the Student Governing Association By Laws, Article V, Section 3, number 2. Students who engage in behavior that disrupts the learning environment may be asked to leave the class.

Mutual Respect and Inclusion in K-State Teaching & Learning Spaces

At K-State, faculty and staff are committed to creating and maintaining an inclusive and supportive learning environment for students from diverse backgrounds and perspectives. K-State courses, labs, and other virtual and physical learning spaces promote equitable opportunity to learn, participate, contribute, and succeed, regardless of age, race, color, ethnicity, nationality, genetic information, ancestry, disability, socioeconomic status, military or veteran status, immigration status, Indigenous identity, gender identity, gender expression, sexuality, religion, culture, as well as other social identities.

Faculty and staff are committed to promoting equity and believe the success of an inclusive learning environment relies on the participation, support, and understanding of all students. Students are encouraged to share their views and lived experiences as they relate to the course or their course experience, while recognizing they are doing so in a learning environment in which all are expected to engage with respect to honor the rights, safety, and dignity of others in keeping with the (K-State Principles of Community).

If you feel uncomfortable because of comments or behavior encountered in this class, you may bring it to the attention of your instructor, advisors, and/or mentors. If you have questions about how to proceed with a confidential process to resolve concerns, please contact the Student Ombudsperson Office. Violations of the student code of conduct can be reported here. If you experience bias or discrimination, it can be reported here.

Netiquette

This is our personal policy and not a required syllabus statement from K-State. It has been adapted from this statement from K-State Global Campus, and the Recurse Center Manual. We have adapted their ideas to fit this course.

Online communication is inherently different than in-person communication. When speaking in person, many times we can take advantage of the *context* and *body language* of the person speaking to better understand what the speaker *means*, not just what is said. This information is not present when communicating online, so we must be much more careful about what we say and how we say it in order to get our meaning across.

Here are a few general rules to help us all communicate online in this course, especially while using tools such as Canvas or Discord:

- **Use a clear and meaningful subject line to announce your topic.** Subject lines such as "Question" or "Problem" are not helpful. Subjects such as "Logic Question in Project 5, Part 1 in Java" or "Unexpected Exception when Opening Text File in Python" give plenty of information about your topic.
- **Use only one topic per message.** If you have multiple topics, post multiple messages so each one can be discussed independently.
- **Be thorough, concise, and to the point.** Ideally, each message should be a page or less.
- **Include exact error messages, code snippets, or screenshots, as well as any previous steps taken to fix the problem.** It is much easier to solve a problem when the exact error message or screenshot is provided. If we know what you've tried so far, we can get to the root cause of the issue more quickly.
- **Consider carefully what you write before you post it.** Once a message is posted, it becomes part of the permanent record of the course and can easily be found by others.
- **If you are lost, don't know an answer, or don't understand something, speak up!** Piazza allows you to send a message privately to the instructors, or post anonymously so other students don't know your identity. Don't be afraid to ask questions anytime, as you can choose to do so without any fear of being identified by your fellow students.
- **Class discussions are confidential.** Do not share information from the course with anyone outside of the course without explicit permission.
- **Do not quote entire message chains; only include the relevant parts.** When replying to a previous message, only quote the relevant lines in your response.
- **Do not use all caps.** It makes it look like you are shouting. Use appropriate text markup (bold, italics, etc.) to highlight a point if needed.
- **No feigning surprise.** If someone asks a question, saying things like "I can't believe you don't know that!" are not helpful, and only serve to make that person feel bad.
- **No "well-actually's."** If someone makes a statement that is not entirely correct, resist the urge to offer a "well, actually…" correction, especially if it is not relevant to the discussion. If you can help solve their problem, feel free to provide correct information, but don't post a correction just for the sake of being correct.
- **Do not correct someone's grammar or spelling.** Again, it is not helpful, and only serves to make that person feel bad. If there is a genuine mistake that may affect the

meaning of the post, please contact the person privately or let the instructors know privately so it can be resolved.

- **Avoid subtle -isms and microaggressions.** Avoid comments that could make others feel uncomfortable based on their personal identity. See the syllabus section on Diversity and Inclusion above for more information on this topic. If a comment makes you uncomfortable, please contact the instructor.
- **Avoid sarcasm, flaming, advertisements, lingo, trolling, doxxing, and other bad online habits.** They have no place in an academic environment. Tasteful humor is fine, but sarcasm can be misunderstood.

As a participant in course discussions, you should also strive to honor the diversity of your classmates by adhering to the K-State Principles of Community.

Face Coverings

All students are expected to comply with K-State's face mask policy. As of August 2, 2021, everyone must wear face masks over their mouths and noses in all indoor spaces on university property, including while attending in-person classes. This policy is subject to change at the university's discretion. For additional information and the latest on K-State's face covering policy, see this page.

Academic Freedom Statement

Kansas State University is a community of students, faculty, and staff who work together to discover new knowledge, create new ideas, and share the results of their scholarly inquiry with the wider public. Although new ideas or research results may be controversial or challenge established views, the health and growth of any society requires frank intellectual exchange. Academic freedom protects this type of free exchange and is thus essential to any university's mission.

Moreover, academic freedom supports collaborative work in the pursuit of truth and the dissemination of knowledge in an environment of inquiry, respectful debate, and professionalism. Academic freedom is not limited to the classroom or to scientific and scholarly research, but extends to the life of the university as well as to larger social and political questions. It is the right and responsibility of the university community to engage with such issues.

Campus Safety

Kansas State University is committed to providing a safe teaching and learning environment for student and faculty members. In order to enhance your safety in the unlikely case of a campus emergency make sure that you know where and how to quickly exit your classroom and how to follow any emergency directives. To view additional campus emergency information go to the University's main page, www.k-state.edu, and click on the Emergency Information button, located at the bottom of the page.

Student Resources

K-State has many resources to help contribute to student success. These resources include accommodations for academics, paying for college, student life, health and safety, and others found at www.k-state.edu/onestop.

Student Academic Creations

Student academic creations are subject to Kansas State University and Kansas Board of Regents Intellectual Property Policies. For courses in which students will be creating intellectual property, the K-State policy can be found at University Handbook, Appendix R: Intellectual Property Policy and Institutional Procedures (part I.E.). These policies address ownership and use of student academic creations.

Mental Health

Your mental health and good relationships are vital to your overall well-being. Symptoms of mental health issues may include excessive sadness or worry, thoughts of death or self-harm, inability to concentrate, lack of motivation, or substance abuse. Although problems can occur anytime for anyone, you should pay extra attention to your mental health if you are feeling academic or financial stress, discrimination, or have experienced a traumatic event, such as loss of a friend or family member, sexual assault or other physical or emotional abuse.

If you are struggling with these issues, do not wait to seek assistance.

- Kansas State University Counseling Services offers free and confidential services to assist you to meet these challenges.
- Lafene Health Center has specialized nurse practitioners to assist with mental health.
- The Office of Student Life can direct you to additional resources.
- K-State Family Center offers individual, couple, and family counseling services on a sliding fee scale.
- Center for Advocacy, Response, and Education (CARE) provides free and confidential assistance for those in our K-State community who have been victimized by violence.

For Kansas State Polytechnic Campus:

- Kansas State Polytechnic Counseling Services offers free and confidential services to assist you to meet these challenges.
- The Kansas State Polytechnic Office of Student Life can direct you to additional resources.

University Excused Absences

K-State has a University Excused Absence policy (Section F62). Class absence(s) will be handled between the instructor and the student unless there are other university offices involved. For university excused absences, instructors shall provide the student the opportunity to make up missed assignments, activities, and/or attendance specific points that contribute to the course grade, unless they decide to excuse those missed assignments from the student’s course grade. Please see the policy for a complete list of university excused absences and how to obtain one. Students are encouraged to contact their instructor regarding their absences.

Appendix B - Pendant Survey

Programming with Pendant Evaluation Survey

Start of Block: Introduction

Instruction Over the course of this semester, you have had the opportunity to utilize a tool – Pendant – intended to assist you in learning Object-Oriented programming best practices. This survey is part of a study intended to evaluate that tool and how you have used it in your programming practice.

In order to receive points for this survey, please enter your eid:

____

Thank you for participating in this survey. Your answers will be used to improve the Pendant tool and design future teaching tools.

Q1 Overall, how helpful or unhelpful did you find the Pendant Tool?

- ○ Extremely helpful (1)
- ○ Moderately helpful (2)
- ○ Slightly helpful (3)
- ○ Neither helpful nor unhelpful (4)
- ○ Slightly unhelpful (5)
- ○ Moderately unhelpful (6)
- ○ Extremely unhelpful (7)

Q2 How useful did you find the Pendant tool for ensuring you structured your classes as expected?

- ○ Not at all useful (1)
- ○ Slightly useful (2)
- ○ Moderately useful (3)
- ○ Very useful (4)
- ○ Extremely useful (5)

Q3 How useful did you find the Pendant tool for ensuring you used XML comments as expected?

○ Not at all useful (1)

○ Slightly useful (2)

○ Moderately useful (3)

○ Very useful (4)

○ Extremely useful (5)

Q4 How useful did you find the Pendant tool for ensuring you followed C# naming conventions?

○ Not at all useful (1)

○ Slightly useful (2)

○ Moderately useful (3)

○ Very useful (4)

○ Extremely useful (5)

Q5 How useful did you find the Pendant tool for learning about and fixing design issues in your code?

○ Not at all useful (1)

○ Slightly useful (2)

○ Moderately useful (3)

○ Very useful (4)

○ Extremely useful (5)

Q6 Please feel free to offer any explanations of how you used Pendant during the semester:

____

____

____

____

____

Page Break

This next section concerns your use of Git in conjunction with the Pendant tool.

Q7 How difficult did you find the process of setting up the GitHub webhook with Pendant?

- ○ Extremely difficult (1)
- ○ Somewhat difficult (2)
- ○ Neither easy nor difficult (3)
- ○ Somewhat easy (4)
- ○ Extremely easy (5)

Q8 How difficult did you find the process of creating a feature branch for your milestones?

- ○ Extremely difficult (1)
- ○ Somewhat difficult (2)
- ○ Neither easy nor difficult (3)
- ○ Somewhat easy (4)
- ○ Extremely easy (5)

Q9 When did you normally create your milestone feature branches (i.e. ms1, ms2)?

○ Before I started working on the milestone (1)

○ Just after I started working on the milestone (2)

○ Pretty far into working on the milestone (3)

○ Once I thought I was done with the milestone (4)

○ I didn't normally create feature branches (5)

Q10 How often did you normally commit to your milestone feature branches?

○ After every minor code change (1)

○ After every major code change (2)

○ At the end of a session of coding (3)

○ Once I thought I had the milestone complete (4)

○ I did not normally commit to milestone feature branches (5)

Q11 How difficult did you find merging your feature branch back into your main branch?

○ Extremely difficult (1)

○ Somewhat difficult (2)

○ Neither easy nor difficult (3)

○ Somewhat easy (4)

○ Extremely easy (5)

Q12 How difficult did you find creating a release on GitHub?

○ Extremely difficult (1)

○ Somewhat difficult (2)

○ Neither easy nor difficult (3)

○ Somewhat easy (4)

○ Extremely easy (5)

Q13 Please feel free to offer any explanations of your answers to the questions concerning Git:

____

____

____

____

____

Page Break

This next series of questions concerns your time management.

Q14 When did you normally start your milestones?

○ At least a week before it was due (1)

○ At least five days before it was due (2)

○ At least four days before it was due (3)

○ At least three days before it was due (4)

○ At least two days before it was due (5)

○ The day it was due (6)

○ I didn’t normally start working on a milestone project until after it was due (7)

Q15 How much time did you normally spend on your milestones?

- ○ More than 10 hours (1)
- ○ Between 8 and 10 hours (2)
- ○ Between 5 and 8 hours (3)
- ○ Between 3 and 5 hours (4)
- ○ Between 0 and 3 hours (5)

Q16 How often did you attend your class session?

- ○ I attended almost all of my class sessions (1)
- ○ I attended more than half of my class sessions (2)
- ○ I attended about half of my class sessions (3)
- ○ I attended less than half of my class sessions (4)
- ○ I rarely attended my class sessions (5)

Q17 When did you attend one of the other class section sessions?

- ◯ I never attended another class section (1)
- ◯ I attended another class session only when I was unable to join my own (2)
- ◯ I attended another class session only when I had specific questions I needed answered (3)
- ◯ I attended most of another class section's sessions (4)
- ◯ I attended most of both the other class sections' sessions (5)

Q18 Please feel free to offer any explanations of your answers concerning time management:

____

____

____

____

____

Page Break

This next series of questions concerns your use of the Visual Studio Integrated Development Environment.

Q19 When using Visual Studio, how often do you use the Error List to identify issues in your code? The error list is this part of the Visual Studio editor:

○ Never (1)

○ Sometimes (2)

○ About half the time (3)

○ Most of the time (4)

○ Always (5)

Q20 When viewing an error in the Error List, how often do you click the error code hyperlink to get help about that specific error? The hyperlinked error code is circled in red:

○ Never (1)

○ Sometimes (2)

○ About half the time (3)

○ Most of the time (4)

○ Always (5)

Q21 What reason(s) did you not use the error code hyperlink in the Error List (Check all that apply)?

☐ I knew what my mistake was based on the error name. (1)

☐ I had followed the hyperlink previously for another error. (2)

☐ I found the linked help pages unhelpful on previous issues. (3)

☐ I did not realize that the error code was a hyperlink that could be followed. (4)

☐ I did not want to get distracted by trying something new. (5)

☐ I looked up the error using a Google (or other search engine/help site) search instead. (6)

☐ I posted my error to the class Discord channel instead. (7)

☐ I contacted the instructor or a TA about my error instead. (8)

☐ I asked a classmate or friend about my error instead. (9)

Q22 When using Visual Studio, how often do you set breakpoints in your code while debugging?

A breakpoint is represented by a red circle next to a line of code:

○ Never (1)

○ Sometimes (2)

○ About half the time (3)

○ Most of the time (4)

○ Always (5)

Page Break

This next series of questions concerns your use of the Pendant tool.

Q23 In what way(s) did you use the Pendant tool (Mark all that apply):

▢ I used it as a checklist of tasks I needed to complete to finish the milestone (1)

▢ I used it as a final check to ensure I did not miss a requirement for the milestone (2)

▢ I used it like a spelling/grammar check to ensure I followed naming conventions (3)

▢ I used it to find out where I needed to add XML-style comments (4)

▢ I used it while writing classes to ensure I followed the structural requirements (5)

Q24 The Pendant tool is under active development, and as such experienced some bugs and intermittent outages. Did you encounter some of these?

○ No (1)

○ Yes (2)

Q25 If you answered “yes” to the previous question, how frustrating did you find these issues?

○ 0 (0)

○ 1 (1)

○ 2 (2)

○ 3 (3)

○ 4 (4)

○ 5 (5)

○ 6 (6)

○ 7 (7)

○ 8 (8)

○ 9 (9)

○ 10 (10)

Q26 How likely are you to recommend using the Pendant tool to other students?

○ 0 (0)

○ 1 (1)

○ 2 (2)

○ 3 (3)

○ 4 (4)

○ 5 (5)

○ 6 (6)

○ 7 (7)

○ 8 (8)

○ 9 (9)

○ 10 (10)

Page Break

This section concerns the "Get Help" links Pendant validations provide for some issues, as seen in the image below:

Q27 How often did you follow the “Get Help” link for an issue Pendant found in your code?

○ Never (1)

○ Sometimes (2)

○ About half the time (3)

○ Most of the time (4)

○ Always (5)

Q28 When you did not use the “Get Help” link, please mark why (Check all that apply):

☐ I knew what the mistake was based on the issue text (1)

☐ I had visited the link for the same issue at a previous time (2)

☐ I found the linked help pages unhelpful on previous issues (3)

☐ I did not realize it was a hyperlink that could be followed (4)

☐ I did not want to get distracted by trying something new (5)

☐ I looked up the error using a Google (or other search engine/help site) search instead. (6)

☐ I posted the issue to the class Discord channel instead. (7)

☐ I contacted the instructor or TA about the issue instead. (8)

☐ I asked a classmate or friend about the issue instead. (9)

Q29 Please feel free to offer any explanations of your answers concerning the “Get Help” link in Pendant:

____

____

____

____

____

Page Break

This section concerns the “Show Location” links Pendant validations provide for some issues, as seen in the image below:

Q30 How often did you follow the “Show Location” link for an issue Pendant found in your code?

○ Never (1)

○ Sometimes (2)

○ About half the time (3)

○ Most of the time (4)

○ Always (5)

Q31 When you did not use the “Show Location” link, please mark why (Choose all that apply):

☐ I knew where the issue would be located in my code based on its text (1)

☐ I had visited the location a previous time the issue was reported (2)

☐ I found the displaying of the location of the problem unhelpful in fixing my issues (3)

☐ I did not realize it was a hyperlink that would show where the issue was located in my code (4)

☐ I did not want to get distracted by trying something new (5)

Q32 Please feel free to offer any explanations of your answers concerning the “Show Location” link:

____

____

____

____

____

Page Break

The next few questions concern integrating Pendant functionality into an IDE (a program for writing software like Visual Studio or VS Code). Please mark your agreement with the statements.

Q33 I would prefer it if the issues identified by Pendant were instead identified by my IDE (i.e., Visual Studio) as I wrote my code.

○ Strongly disagree (1)

○ Somewhat disagree (2)

○ Neither agree nor disagree (3)

○ Somewhat agree (4)

○ Strongly agree (5)

Q34 If the issues identified by Pendant were instead identified by my IDE (i.e., Visual Studio) as I wrote my code, I would have paid more attention to them.

○ Strongly disagree (1)

○ Somewhat disagree (2)

○ Neither agree nor disagree (3)

○ Somewhat agree (4)

○ Strongly agree (5)

Q35 If I could have accessed the “Get Help” pages of Pendant directly from my IDE (i.e., Visual Studio), I would have made greater use of them.

○ Strongly disagree (1)

○ Somewhat disagree (2)

○ Neither agree nor disagree (3)

○ Somewhat agree (4)

○ Strongly agree (5)

Milestone 3 Requirements

For this milestone, you will be creating base classes for entrées, sides, treats, and drinks served at GyroScope. This will involve refactoring some already written classes, as well as adding some new ones.

General requirements:

- You need to follow the style laid out in the C# Coding Conventions

Assignment requirements:

You will need to:

- Address the spelling errors found in Milestone 2
- Create base classes for:
  - Sides
  - Entrees
  - Treats
  - Drinks
- Create a base class for Gyros
- Create new enum for:
  - SodaFlavors
- Create new classes for:
  - Cancer Helvah Cake
  - Libra Libation
  - Capricorn Mountain Tea
- Refactor existing classes to use inhertance

Purpose:

This milestone serves to introduce and utilize aspects of polymorphism including base classes, abstract base classes, abstract methods, virtual methods, and method overriding. While the actual programming involved is straightforward, the concepts involved can be challenging to master. If you have any confusion after you have read the entire assignment please do not hesitate to reach out to a Professor Bean, the TAs, or your classmates over Discord.

Spelling Errors

GyroScope HQ is upset that a number of spelling errors were found in signature GyroScope dishes. These must be addressed in this new milestone. Specifically:

- “Ares” should be “Aries”
- “Pices” should be “Pisces”

You will need to correct these misspellings in all class names, file names, and comments.

> ***TIP***
> *In Visual Studio, you can right-click on a symbol like a class or variable name and choose Rename from the context menu. Then, type the correction and hit enter, and it will update the symbol everywhere it appears in your code. You can also mark the checkbox to update it in comments.*

Abstract Base Classes

You will need to create a base class for each of the kinds of items served at GyroScope: *Entrees*, *Sides*, *Treats*, and *Drinks*. Because you will never instantiate one of these classes directly i.e. you would never write:

```
Side side = new Side();
```

But rather:

```
Side side = new TaurusTabuleh();
```

You will want to declare these base classes abstract.

As you create these base classes, carefully consider what properties all items in that category have in common. These properties should then be implemented in the base class, which will be one of the following:

- **A regular property** if the exact same functionality will be used in the derived classes and should never need to change,
- **A virtual property** if the exact same functionality will be used in almost all derived classes, but at least one is a special case, or
- **An abstract property** if the derived classes all have the same property but the returned values are different for each one.

Each abstract base class should be placed in the corresponding file and namespace, i.e. the class Side should be declared in the GyroScope.Data.Sides namespace and in a file named Side.cs. The classes you need to implement are Entree, Side, Treat, and Drink.

To give you a head start, your Treat base class should look like:

```
namespace GyroScope.Data.Treats
{
    /// <summary>
    /// A base class for all treats sold at GyroScope
    /// </summary>
    public abstract class Treat
    {

        /// <summary>
        /// The price of the treat
        /// </summary>
        public abstract decimal Price { get; }

        /// <summary>
        /// The calories of the treat
        /// </summary>
        public abstract uint Calories { get; }
    }
}
```

Create a Gyro Base Class

Another issue GyroScope HQ has identified with the milestone 2 design is that customers expect to and routinely do completely customize a Gyro - substituting ingredients normally found in one Gyro into another one (such as adding wing sauce and peppers to a LeoLambGyro). The GyroScope software will need to support this.

The solution that you will be adopting is to create a base Gyro class that contains all the possible ingredient properties from the different Gyros available at GyroScope. This base class should be named Gyro. This Gyro class should inherit from the Entree class. The other Gyro classes (VirgoClassicGyro, LeoLambGyro, and ScorpioSpicyGyro) should inherit from this class, and set the appropriate default values for the boolean and enumeration properties through a parameterless constructor.

Create new Enum

You will need to create a new enum in the GyroScope.Data.Enums namespace to represent the soda flavors offered at GryoScope which should be named LibraLibationFlavor and contain:

- Orangeade
- SourCherry
- Biral
- PinkLemonada

Create new Classes

You will need to create three new classes, CancerHalvehCake, LibraLibation, and CapricornMountainTea.

CancerHalvehCake

The class CancerHalvehCake should be declared in the GyroScope.Data.Treats namespace. It should inherit from the Treat base class. It’s price is $3.00 and it has 272 calories.

LibraLibation

The class LibraLibation should be declared in the GyroScope.Data.Drinks namespace. It represents a Greek soda. It should have a Flavor property of type LibraLibationFlavor with getters and setters. It should also have a bool property for Sparkling (carbonated) which should default to true. It should have a Calories get-only property of type uint and a Price property of type decimal using the values laid out in the table below. It should also have a Name get-only property, which should return a string in the form “[Still or Sparkling] [Flavor] Libra Libation” where [Still or Sparkling] is Still if the Sparkling property is false, and Sparkling if it is true, and [Flavor] is Orangeade, Sour Cherry, Biral, or Pink Lemonada (based on the Flavor property). Note the use of spaces in the flavor.

Flavor	Price	Calories
Orangeade	$1.00	180
Sour Cherry	$1.00	100
Biral	$1.00	120
Pink Lemonada	$1.00	41

CapricornMountainTea

The class CapricornMountainTea should alsobe declared in the GryoScope.Data.Drinks namespace. It represents a tea brewed from the ironwort plant. It should have a Price property of type Decimal that returns $2.50, and a Calories property of type uint that returns 0 calories. It should also have a boolean property Honey indicating the tea is sweetened with honey that defaults to false. If it is true, the Calories property should instead return 64.

Refactor Existing Classes to use Inheritance

Once you have the Side base class, refactor your existing side classes (TaurusTabuleh, GeminiStuffedGrapeLeaves, SagittariusGreekSalad, and AriesFries) to inherit from it. You will also want to refactor their existing properties as they should now inherit many of them from the base class; some may be deleted, others will need to be refactored as overridden methods by adding the override keyword.

Likewise you will want to refactor the exisiting entree base classes (VirgoClassicGyro, ScorpioSpicyGyro, LeoLambGyro, and PiscesFishDish) to inherit from the Entree base class. You will want to refactor thier properties as well.

Finally, you will want to refactor the exisiting treat class AquariusIce to inherit from the Treat base class and also refactor its properties as needed.

The Milestone Feature Branch

You will want to create a feature branch and push it to GitHub for your validations to be generated on https://pendant.cs.ksu.edu. For this milestone, your feature branch should be named ms3.

Submitting the Assignment

Once your project is complete, merge your feature branch back into the main branch and create a release tagged v0.3.0 with name “Milestone 3”. Copy the URL for the release page and submit it to the Canvas assignment.

Grading Rubric

The grading rubric for this assignment will be:

25% Structure Did you implement the structure as laid out in the specification? Are the correct names used for classes, enums, properties, methods, events, etc.? Do classes inherit from expected base classes?

25% Documentation Does every class, method, property, and field use the correct XML-style documentation? Does every XML comment tag contain explanatory text?

25% Design Are you appropriately using C# to create reasonably efficient, secure, and usable software? Does your code contain bugs that will cause issues at runtime?

25% Functionality Does the program do what the assignment asks? Do properties return the expected values? Do methods perform the expected actions?
Projects that do not compile will receive an automatic grade of 0.

Appendix D - Final Exam

CIS 400 Final Exam

Name: ____________________________________ WID: ________________ Section: _____

Vocabulary Questions

The following questions concern THEORETICAL aspects of Object-Orientation

Q1. Describe how encapsulation is implemented in object-oriented languages.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q2. Describe the relationship between a class and object in an object-oriented language.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q3. Describe how message passing is implemented in an object-oriented language.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

The following questions concern specific aspects of C# syntax

Q4. What is the purpose of the *public, private,* and *protected* keywords?

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q5. Describe the differences between an *abstract class* and an *interface*.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q6. Describe the difference between the *new* and *virtual* keywords when used with a method.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Coding Questions

The following questions should be answered using the diagrams and tables on the last page of the exam

Q7. Describe the relationship between the classes **`CatalogItem`** and **`Loan`** indicated by the UML diagram.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q8. Describe the relationship between the classes **`Patron`** and **`Loan`** indicated by the UML diagram.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q9. Describe the relationship between the classes **`CatalogItem`** and **Book** indicated by the UML diagram.

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q10. Implement the **`Overdue`** property of the **`Loan`** class as specified by the UML diagram. Note: The static property **`DateTime.Today`** returns the current day.

```
public class Loan
{

}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q11. Implement the **`Renew()`** method of the **`Loan`** class as indicated by the UML diagram, and following the rules for loans:

```
public class Loan
{

}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q12. Implement the **`OverdueItems`** property of the **`Patron`** class as indicated by the UML diagram and following the rules for loans:

```
public class Patron
{

}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q13. Implement the **`Checkout()`** method of the **`Patron`** class as indicated by the UML diagram and following the rules for loans:

```
public class Patron
{

}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q14. Implement the constructor of the **Loan** class as indicated by the UML diagram, and following the rules for loans and initializing all properties:

```
public class Loan
{
```

```
}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q15. Complete the tests of **Patron**, following the assumptions of the UML diagram and the rules for loans. Assume that any method that would fail because of a rule for loans returns false rather than throwing an error, and that the state of the object is not modified. You should provide at least four entries of InlineData when needed:

```
using Library;
using Xunit;
namespace LibraryTest
{
    public class PatronTest
    {
        [Fact]
        public void PatronsShouldOnlyBeAbleToCheckoutEightBooks()
        {



         }
        [Theory]



      public void PatronCanCheckOutOneDVD(string title, string description)
      {
                var m = new Movie(title, description);
                var p = new Patron(){FirstName = “John”, LastName = “Doe”};
                var flag = p.Checkout(m);
                Assert.True(flag); // Checkout should return true if successful
                Assert.Contains(m, p.Loans); // The dvd should be in the loans
      }
      [Fact]
      public void ChangingLastNameShouldNotifyOfPropertyChanging(string lastName)
      {



        }
}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q16. Complete the tests of **Loan**, following the assumptions of the UML diagram and the rules for loans. Assume that any method that would fail because of a rule for loans returns false rather than throwing an error, and that the state of the object is not modified. You should provide at least four entries of InlineData when needed:

```
using Library;
using Xunit;
namespace LibraryTest
{
  public class LoanTest
  {
     [Fact]
     public void NewBookLoansAreForTwoWeeks()
     {



     }
     [Fact]
     public void NewBookLoansHaveThreeRenewals()
     {



     }
     [Fact]
     public void RenewingABookLoanDecrementsRenewals()
     {



     }
     [Fact]
     public void RenewingABookLoanWithNoRenewalsReturnsFalseAndDoesNotChangeDueDate()
     {



     }
}
```

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q17. Draw the UML Diagram for all classes defined in the **BearExample** namespace from the listing at the end of the exam:

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

Q18. Upon executing the program appearing in the code listing at the end of the exam, what output will be printed to the console?

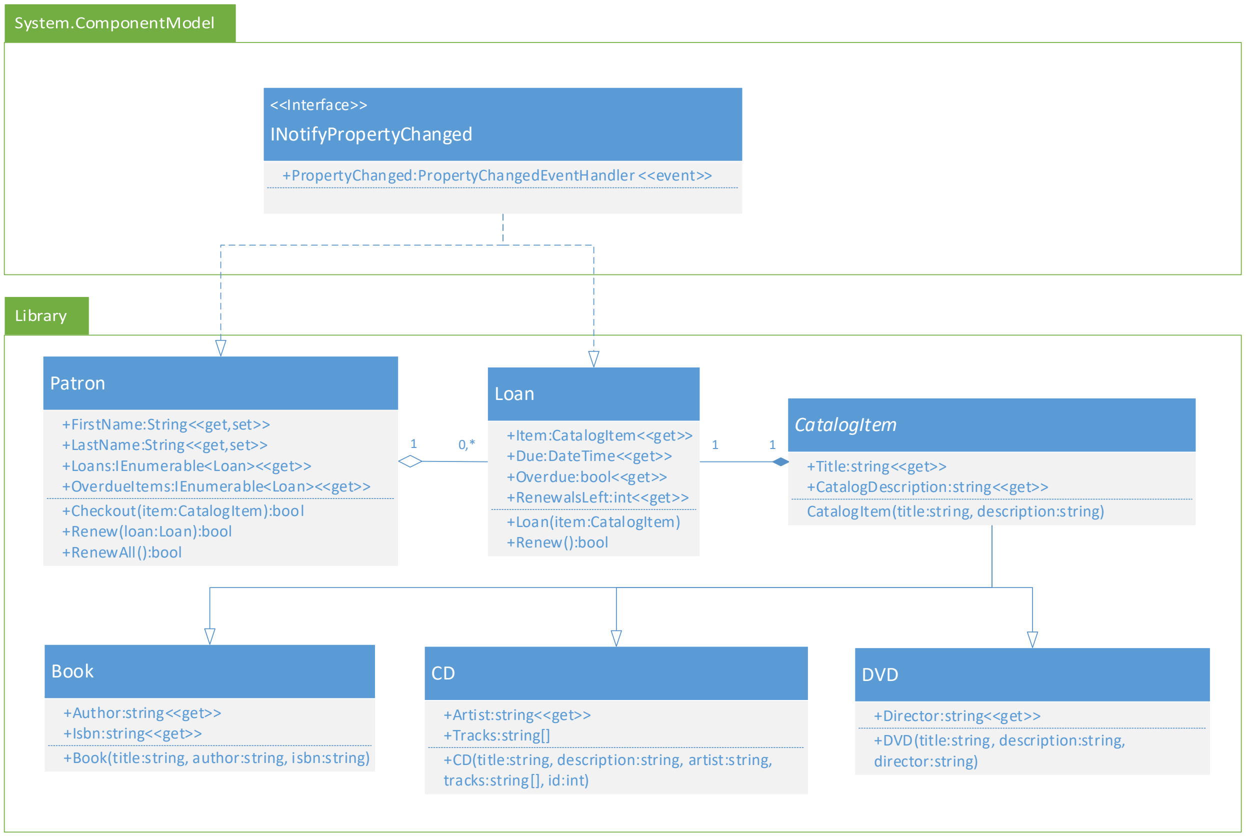
}

Confidence (circle one): very confident - (5) (4) (3) (2) (1) - Not confident at all

UML Diagrams, Data Tables, and Code Listings

The following UML Diagrams, Tables, and Code listings correspond to questions asked earlier in this test. You may want to remove this page for an easy reference.

Figure 1 – Library UML Diagram



System.ComponentModel
<<Interface>>
INotifyPropertyChanged
+PropertyChanged:PropertyChangedEventHandler <<event>>
Library
Patron
+FirstName:String<<get,set>>
+LastName:String<<get,set>>
+Loans:IEnumerable<Loan><<get>>
+OverdueItems:IEnumerable<Loan><<get>>
+Checkout(item:CatalogItem):bool
+Renew(loan:Loan):bool
+RenewAll():bool
1
0,*
Loan
+Item:CatalogItem<<get>>
+Due:DateTime<<get>>
+Overdue:bool<<get>>
+RenewalsLeft:int<<get>>
+Loan(item:CatalogItem)
+Renew():bool
1
1
CatalogItem
+Title:string<<get>>
+CatalogDescription:string<<get>>
CatalogItem(title:string, description:string)
Book
+Author:string<<get>>
+Isbn:string<<get>>
+Book(title:string, author:string, isbn:string)
CD
+Artist:string<<get>>
+Tracks:string[]
+CD(title:string, description:string, artist:string, tracks:string[], id:int)
DVD
+Director:string<<get>>
+DVD(title:string, description:string, director:string)

Figure 2 – Rules for Loans

Rules for Library Loans:

- *A patron may only have eight checked-out (loaned) books at a time*
- *Books are loaned (both checked out and renewed) for a period of two weeks (14 days)*
- *Books can be renewed three times*
- *A patron may only have six checked-out (loaned) CDs at a time*
- *CDs are loaned (both initially checked-out and renewed) for a period of one week (7 days)*
- *CDs can be renewed two times*
- *A patron may only have four checked-out (loaned) DVDs at a time*
- *DVDs are loaned (both initially checked-out and renewed) for a period of one week (7 days)*
- *DVDs can be renewed once*

Figure 3 – DateTime Hints

- *The property **DateTime.Today** evaluates to today's date with time set to 00:00:00.*
- *The **AddDays(double n)** method of a **DateTime** object returns a new **DateTime** that is **n** days after the **DateTime** it was called on.*
- *The **DateTime** struct overrides **CompareTo()**, so two **DateTimes** are considered equal if they represent the same date and time, and a later date is considered larger. This also works with the **>**, **<**, **==**, **>=**, and **<=** operators.*

Figure 4 – Code Listing

```
using System;

namespace BearExample
{
    public class Bear
    {
        public virtual string Speak()
        {
            return “Growl”;
        }
    }

    public class StorybookBear : Bear
    {
        public override string Speak()
        {
            return “\”Growl,\” said the bear”;
        }
    }

    public class Pooh : StorybookBear
    {
        public new string Speak()
        {
            return “Oh dear!”;
        }
    }

    public class Courdorouy : StorybookBear
    {
        public override string Speak()
        {
            return “I’ve always wanted a friend.”;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Bear[] bears = new Bear[]{
                new Bear(),
                new StorybookBear(),
                new Pooh(),
                new Courdorouy()
            };
            foreach (var bear in bears) Console.WriteLine(bear.Speak());
            foreach (dynamic bear in bears) Console.WriteLine(bear.Speak());
            foreach (Bear bear in bears)
            {
                if (bear is StorybookBear storybookBear)
                {
                    Console.WriteLine(storybookBear.Speak());
                }
            }
        }
    }
}
```

Appendix E - Survey Results

Table E.1

Responses to Q1 "Overall, how helpful or unhelpful did you find the Pendant Tool?"

Response	Frequency	Percentage
Extremely Helpful	11	22%
Moderately Helpful	28	56%
Sightly Helpful	8	16%
Neither Helpful nor Unhelpful	1	2%
Slightly Unhelpful	0	0%
Moderately Unhelpful	2	4%
Extremely Unhelpful	0	0%
Total	**50**	**100%**

Table E.2

Responses to Q2 "How useful did you find the Pendant Tool for ensuring you structured your classes as expected?"

Response	Frequency	Percentage
Extremely Useful	12	24%
Very Useful	14	28%
Moderately Useful	15	30%
Slightly Useful	7	14%

Not at all Useful	2	4%
Total	**50**	**100%**

Table E.3

Responses to Q3 "How useful did you find the Pendant tool for ensuring you used XML comments as expected?

Response	Frequency	Percentage
Extremely Useful	27	54%
Very Useful	8	16%
Moderately Useful	10	20%
Slightly Useful	2	4%
Not at all Useful	3	6%
Total	**50**	**100%**

Table E.4

Responses to Q4 "How useful did you find the Pendant tool for ensuring you followed C# naming conventions?

Response	Frequency	Percentage
Extremely Useful	19	38%
Very Useful	12	24%
Moderately Useful	11	22%

Slightly Useful	5	10%
Not at all Useful	3	6%
Total	**50**	**100%**

Table E.5

Responses to Q5 "How useful did you find the Pendant tool for learning about and fixing design issues in your code?"

Response	Frequency	Percentage
Extremely Useful	4	8%
Very Useful	9	18%
Moderately Useful	14	28%
Slightly Useful	16	32%
Not at all Useful	7	14%
Total	**50**	**100%**

Table E.6

Responses to Q7 "How difficult did you find the process of setting up the GitHub webhook with Pendant?"

Response	Frequency	Percentage
Extremely Easy	20	40%

Somewhat Easy	19	38%
Neither Easy nor Difficult	7	14%
Somewhat Difficult	4	8%
Very Difficult	0	0%
Total	**50**	**100%**

Table E.7

Responses to Q8 "How difficult did you find the process of creating a feature branch for your milestones?"

Response	Frequency	Percentage
Extremely Easy	28	56%
Somewhat Easy	14	28%
Neither Easy nor Difficult	5	10%
Somewhat Difficult	2	4%
Very Difficult	1	2%
Total	**50**	**100%**

Table E.8

Responses to Q9 "When did you normally create your milestone feature branches (i.e. ms1, ms2)?

Response	Frequency	Percentage

Before I started working on the milestone	28	56%
Just after I started working on the milestone	11	22%
Pretty far into working on the milestone	4	8%
Once I thought I was done with the milestone	7	14%
I didn’t normally create feature branches	0	0%
Total	**50**	**100%**

Table E.9

Responses to Q10 “How often did you normally commit to your milestone feature branches?”

Response	Frequency	Percentage
After every minor code change	2	4%
After every major code change	13	26%
At the end of a session of coding	21	42%
Once I thought the assignment was complete	14	28%
I did not normally commit to milestone branches	0	0%
Total	**50**	**100%**

Table E.10

Responses to Q11 “How difficult did you find merging your feature branch back into your main branch?”

Response	Frequency	Percentage

Extremely Easy	20	40%
Somewhat Easy	15	30%
Neither Easy nor Difficult	4	8%
Somewhat Difficult	11	22%
Very Difficult	0	0%
Total	**50**	**100%**

Table E.11

Responses to Q12 "How difficult did you find creating a release on GitHub?"

Response	Frequency	Percentage
Extremely Easy	40	80%
Somewhat Easy	8	16%
Neither Easy nor Difficult	1	2%
Somewhat Difficult	1	2%
Very Difficult	0	0%
Total	**50**	**100%**

Table E.12

Responses to Q14 "When did you normally start your milestones?

Response	Frequency	Percentage
At least a week before it was due	8	16%
At least five days before it was due	11	22%

At least four days before it was due	3	6%
At least two days before it was due	12	24%
At least two days before it was due	16	32%
The day it was due	0	0%
Total	**50**	**100%**

Table E.13

Responses to Q15 "How much time did you normally spend on your milestones?

Response	Frequency	Percentage
More than 10 hours	10	20%
Between 8 and 10 hours	20	40%
Between 5 and 8 hours	14	28%
Between 3 and 5 hours	6	12%
Between 0 and 3 hours	0	0%
Total	**50**	**100%**

Table E.14

Responses to Q16 "How often did you attend your class session?"

Response	Frequency	Percentage
I attended almost all of my class sessions	33	66%
I attended more than half of my class sessions	6	12%
I attended about half of my class sessions	6	12%

I attended less than half of my class sessions	3	6%
I rarely attended class sessions	2	4%
Total	**50**	**100%**

Table E.15

Responses to Q17 "When did you attend one of the other class sessions?"

Response	Frequency	Percentage
I never attended another class section	26	52%
I attended another class session only when I was unable to join my own	4	8%
I attended another class session only when I had specific questions I needed answered	15	30%
I attended most of another class section's sessions	3	6%
I attended most of both the other class sections' sessions	2	4%
Total	**50**	**100%**

Table E.16

Responses to Q19 "When using Visual Studio, how often do you use the Error List to identify issues in your code?"

Response	Frequency	Percentage
Always	29	58%
Most of the time	14	28%

About half the time	5	10%
Sometimes	2	4%
Never	0	0%
Total	**50**	**100%**

Table E.17

When viewing an error in the Error List, how often do you click the error code hyperlink to get help about that specific error?

Response	Frequency	Percentage
Always	4	8%
Most of the time	2	4%
About half the time	4	8%
Sometimes	26	52%
Never	14	28%
Total	**50**	**100%**

Table E.18

Responses for Q21 "What reason(s) did you not use the error code hyperlink on the Error List (Check all that apply)?"

Response	Frequency	Percentage
I knew what my mistake was based on the error name	23	46%

I had followed the hyperlink previously for another error	6	12%
I found the linked help pages unhelpful on previous issues	13	26%
I did not realize the error code was a hyperlink that could be followed	10	20%
I did not want to get distracted by trying something new	1	2%
I looked up the error using Google (or another search engine/help site search) instead	22	44%
I posted my error to the class Discord channel instead	7	14%
I contacted the instructor or a TA about my error instead	10	20%
I asked a classmate or friend about my error instead	12	24%

Table E.19

Responses for Q22 "When using Visual Studio, how often do you set breakpoints in your code while debugging?"

Response	Frequency	Percentage
Always	5	10%
Most of the time	13	26%
About half the time	7	14%
Sometimes	21	42%
Never	4	8%
Total	**50**	**100%**

Table E.20

Responses to Q23 "In what way(s) did you use the Pendant Tool (Mark all that apply):"

Response	Frequency	Percentage
I used it as a checklist of tasks I needed to complete to finish the milestone	17	34%
I used it as a final check to ensure I did not miss a requirement for the milestone	36	72%
I used it like a spelling/grammar check to ensure I was following naming conventions	28	56%
I used it to find out where I needed to add XML-style comments	29	58%
I used it while writing classes to ensure I followed structural requirements	11	22%

Table E.21

Responses to Q24 "The Pendant tool is under active development, and as such experienced some bugs and intermittent outages. Did you encounter some of these?"

Response	Frequency	Percentage
No	6	12%
Yes	44	88%
Total	**50**	**100%**

Table E.22

Responses to Q25 "If you answered "yes" to the previous question, how frustrating did you find these issues?"

Response	Frequency	Percentage
0 = Not Frustrating at all	3	6%
1	1	2%
2	5	1%
3	4	8%
4	7	14%
5	4	8%
6	10	20%
7	10	20%
8	2	4%
9	1	2%
10 = Very Frustrating	0	0%
Total	**47**	**94%**

Table E.23

Responses to Q26 "How likely are you to recommend using the Pendant tool to other students?"

Response	Frequency	Percentage
10 = Very Likely	7	14%
9	11	22%

8	13	26%
7	4	22%
6	4	8%
5	4	8%
4	1	2%
3	3	6%
2	1	2%
1	0	0%
0 = Very Unlikely	2	4%
Total	**50**	**100%**

Table E.24

Responses to Q27 "How often did you follow the 'Get Help' link for an issue Pendant found in your code?"

Response	Frequency	Percentage
Always	0	0%
Most of the time	4	8%
About half the time	6	12%
Sometimes	19	38%
Never	21	42%
Total	**50**	**100%**

Table E.25

Responses to Q28 "When you did not use the "Get Help" link, please mark why (Check all that apply):"

Response	Frequency	Percentage
I knew what my mistake was based on the issue text	20	40%
I had followed the link for the same issue at a previous time	7	14%
I found the linked help pages unhelpful on previous issues	6	12%
I did not realize it was a link that could be followed	4	8%
I did not want to get distracted by trying something new	0	0%
I looked up the error using Google (or another search engine/help site search) instead	8	16%
I posted the issue to the class Discord channel instead	5	10%
I contacted the instructor or a TA about the issue instead	6	12%
I asked a classmate or friend about the issue instead	10	20%

Table E.26

Responses to Q30 "How often did you follow the 'Show Location' link for an issue Pendant found in your code?

Response	Frequency	Percentage
Always	9	6%
Most of the time	10	30%

About half the time	13	26%
Sometimes	15	20%
Never	3	6%
Total	**50**	**100%**

Table E.27

Responses to Q31 "When you did not use the 'Show Location' link, please mark why (Choose all that apply):"

Response	Frequency	Percentage
I knew where the issue would be located in my code based on its text	10	20%
I had visited the location a previous time it was reported	7	14%
I found the displaying of location of the problem unhelpful in fixing my issues	2	4%
I did not realize it was a hyperlink that could show were the issue was located in my code	0	0%
I did not want to get distracted by trying something new	1	2%

Table E.28

Responses to Q33 "I would prefer it if the issues identified by Pendant were instead identified by my IDE (i.e. Visual Studio) as I wrote my code."

Response	Frequency	Percentage
Strongly agree	22	44%

Somewhat agree	20	40%
Neither agree nor disagree	7	14%
Somewhat disagree	1	2%
Strongly disagree	0	0%
Total	**50**	**100%**

Table E.29

Responses to Q34 "If the issues identified by Pendant were instead identified by my IDE (i.e. Visual Studio) as I wrote my code, I would have paid more attention to them."

Response	Frequency	Percentage
Strongly agree	12	24%
Somewhat agree	19	38%
Neither agree nor disagree	10	20%
Somewhat disagree	6	12%
Strongly disagree	3	6%
Total	**50**	**100%**

Table E.30

Responses to Q35 "If I could have accessed the 'Get Help' pages of Pendant directly from my IDE (i.e. Visual Studio), I would have made greater use of them."

Response	Frequency	Percentage
Strongly agree	6	12%

Somewhat agree	17	34%
Neither agree nor disagree	12	24%
Somewhat disagree	11	22%
Strongly disagree	4	8%
Total	**50**	**100%**