UNIVERSITY OF CALIFORNIA, SAN DIEGO

Designing Learning Experiences that Enculturate Novices

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Sarah Marie Esper

Committee in charge:

William G. Griswold, Chair
Michael Clancy
Jeff Elman
Scott Klemmer
Elizabeth Simon

2014

UMI Number: 3644305

UMI

Dissertation Publishing

UMI 3644305

ProQuest®

ال*منــارة* للاستشارات

www.manaraa.com

The Dissertation of Sarah Marie Esper is approved and is acceptable in
quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2014

iii

DEDICATION

To Adrian Guthals, the person who gave me a reason to stay and complete my doctorate.

To Irene Esper, a mother who has shown me that I can do anything.

To Scott Thomas Jr., a man who chose to love me and support me.

To Jessica Capó, an inspiring young woman who makes me want to be the best role model I can be.

To Beth Simon and Bill Griswold, two incredible life-advisors who have helped me develop in so many ways; my gratitude is indescribable.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Bill Griswold for his support as the chair of my committee. His guidance in research and life have truly changed my life.

I would also like to acknowledge Beth Simon for her support as my Co-Advisor. Because of her I discovered Computer Science Education, and because of her I am able to contribute this dissertation.

I would also like to acknowledge my committee members Michael Clancy, Jeff Elman and Scott Klemmer for the advice they have given me on my thesis and dissertation.

I would also like to acknowledge my closest research colleagues, including Stephen Foster, Elizabeth Bales, Tom Lieber, Leo Porter, and Lisa Cowan. Without the advice, guidance, and encouragement of this group, I would not have gotten through many difficult times.

Finally, I would like to acknowldge my close friends and family, who have understood my inability to spend the time I would like with them, and who have continue to love me throughout.

would also like to acknowledge the two elementary schools that allowed us to conduct our study during class.

Section 11.1, in full, is a reprint of the material as it appears in CCSC-SW 2014. Esper, Sarah, Consortium for Computing Sciences in Colleges, 2014. The dissertation author was the primary investigator and author of this paper.

VITA

2009	Software Intern, Microsoft, Bellevue, WA
2009-2010	Research Intern, NASA-JPL, San Diego, CA
2010	Software Intern, Viasat, Carlsbad, CA
2010	Bachelor of Science in Computer Science, University of California, San Diego
2010-2012	Grad Women in Computing President
2010-2014	UCSD CSE Mentor TA
2011	Software Intern, Microsoft, Aliso Viejo, CA
2011-2012	Microsoft Intern Ambassador
2012	Co-Founder and CTO/CPO, ThoughtSTEM, San Diego, CA
2013	Master of Science in Computer Science, University of California, San Diego
2014	Candidate of Philisophy in Computer Science (Computer Engineering), University of California, San Diego
2014	Doctor of Philosophy in Computer Science (Computer Engineering), University of California, San Diego

PUBLICATIONS

Sarah Esper, Stephen R. Foster, William G. Griswold, Carlos Herrera, and Wyatt Snyder. 2014. Codespells: how to design quests to teach java concepts. Koli Calling (November 2014), Submitted for Review.

Sarah Esper. 2014. A discussion on adopting peer instruction in a course focused on risk management. J. Comput. Sci. Coll. 29, 4 (April 2014), 175-182.

Sarah Esper, Samantha R. Wood, Stephen R. Foster, Sorin Lerner, and William G. Griswold. 2014. Codespells: how to design quests to teach java concepts. J. Comput. Sci. Coll. 29, 4 (April 2014), 114-122.

Colleen Lewis, Sarah Esper, Victor Bhattacharyya, Noelle Fa-Kaji, Neftali Dominguez, and Arielle Schlesinger. 2014. Children's perceptions of what counts as a programming language. J. Comput. Sci. Coll. 29, 4 (April 2014), 123-133.

Beth Simon, Sarah Esper, Leo Porter, and Quintin Cutts. 2013. Student experience in a student-centered peer instruction classroom. In Proceedings of the ninth annual international

ACM conference on International computing education research (ICER '13). ACM, New York, NY, USA, 129-136.

Sarah Esper, Stephen R. Foster, and William G. Griswold. 2013. CodeSpells: embodying the metaphor of wizardry for programming. In Proceedings of the 18th ACM conference on Innovation and technology in computer science education (ITiCSE '13). ACM, New York, NY, USA, 249-254.

Sarah Esper, Stephen R. Foster, and William G. Griswold. 2013. On the nature of fires and how to spark them when you're not there. In Proceeding of the 44th ACM technical symposium on Computer science education (SIGCSE '13). ACM, New York, NY, USA, 305-310.

Stephen R. Foster, Sarah Esper, and William G. Griswold. 2013. From competition to metacognition: designing diverse, sustainable educational games. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '13). ACM, New York, NY, USA, 99-108.

Sarah Esper, Beth Simon, and Quintin Cutts. 2012. Exploratory homeworks: an active learning tool for textbook reading. In Proceedings of the ninth annual international conference on International computing education research (ICER '12). ACM, New York, NY, USA, 105-110.

Quintin Cutts, Sarah Esper, Marlena Fecho, Stephen R. Foster, and Beth Simon. 2012. The abstraction transition taxonomy: developing desired learning outcomes through the lens of situated cognition. In Proceedings of the ninth annual international conference on International computing education research (ICER '12). ACM, New York, NY, USA, 63-70.

Quintin Cutts, Sarah Esper, and Beth Simon. 2011. Computing as the 4th "R": a general education approach to computing education. In Proceedings of the eighth annual international conference on International computing education research (ICER '11). ACM, New York, NY, USA, 133-138.

Beth Simon, Sarah Esper and Quintin Cutts. 2011. Experience Report: an AP CS Principles University Pilot. Technical Report CS 2011-0965. University of California at San Diego.

ABSTRACT OF THE DISSERTATION

Designing Learning Experiences that Enculturate Novices

by

Sarah Marie Esper

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2014

Professor William G. Griswold, Chair

Expertise in any field is a combination of skill and **enculturation**. When entering a field, novices gain expertise by developing skills, but more importantly, *novices need to become a part of the community*.

Apprenticeships and mentorships are common methods for modeling learning experiences that combine skill and culture to teach *and* enculturate novices. However, computer science is a field that is growing so quickly that the apprentice/mentor models have been stunted due to high student-teacher ratios. I hypothesize that we can scale the apprenticeship of novices by integrating enculturation with skills learning.

In this dissertation I contribute the design, implementation and evaluation of two

learning experiences that integrate skills and enculturation, thus scaling the teaching of novices ranging from 9-years-old to Bachelor's-level into computer science. The first contribution is to a large, Bachelor's-level, in-person course where I use *Peer Instruction* and *Exploratory Homeworks* to enrich student learning. The second contribution is *CodeSpells*, an immersive learning experience that enculturates and teaches children as young as 9 years old.

Chapter 1

Introduction

The importance of apprenticeship, the integration of culture and skill, in learning has been demonstrated since the 400s B.C. when Socrates took on students such as Plato and Xenophon, who not only studied under him, but *experienced life with him*. Evidence of this is the written accounts of Socrates, which are primarily based on conversations rather than a textbook of facts [162]. With the increased popularity of computer science for all ages, this critical experience is getting harder to deliver.

1.1 Enculturating Novices is Integrated in Learning

In this dissertation, I present two interventions to novices computer science learning environments that enculturate novices' into the field. "Enculturation", here, is defined as the novices engagement with skills and knowledge through activities such as:

- apply them with discipline-specific authenticity,

- discuss them with other members of that discipline,

- explain them to non-members at a basic level, or

- apply them to new problems that they have never seen before.

Enculturating novices means helping them develop an understanding of the decisions and reasonings for actions taken by members of the community for which the novice is studying

1

to be a part. Furthermore, the novice would develop an identity within this community that is recognized by other members of the community.

1.1.1 Enculturation is Traditionally Found in Apprenticeships

Apprenticeships have successfully enculturated novices for thousands of years [28]. They begin with a novice who studies under a master craftsman to understand the trade, both skills and culture. Shortly after, she might "journey" and further develop. Finally, she might become a master craftsman and take on apprentices of her own.

The journey of an apprentice typically begins with an anticipatory learning period [99] where novices begin to enculturate themselves in the field. This learning period is often filled with *legitimate peripheral participation*: performing useful tasks, but not taking on projects that directly result in a product of the field. An example of this can be found in Situated Learning [87], where Lave and Wenger describe the novice tailor that delivers fabrics, a useful part of the community, but not directly creating the clothing.

Through apprenticeships, the novice gets closer to becoming a master craftsman. For example, the tailor will begin by delivering fabrics, and in so doing observe the process of making clothing. The novice is then given a small garment to make, such as a hat. In making the hat, he begins to develop skills, and also has the opportunity to experience a subset of the process of making a garment. Slowly, as the novice tailor gains expertise, he will make larger and more complex garments until he becomes a master tailor himself. The legitimate peripheral participation slowly becomes **full legitimate peripheral participation**; participants are always apprenticing, even experts consult with their peers to develop new skills or expertise.

In computer science, an example might be that a novice constructs test cases for a system, then she might write a small helper method for a larger program, which allows her to develop programming skills while experience a subset of writing an entire program. Like the tailor, she will continue to write more complex and larger programs until she, too,

becomes a master in her craft.

Overtime, the apprentice engages in some form of *legitimate* participation. The set of knowledge and practices that comprise expertise is defined by the existing community's values and through apprenticeships, novices become enculturated in those values as they become more expert. Chapter 2 summarizes this theory with respect to my dissertation.

1.2 Leveraging Social and Visual Embodiments of Computer Science Ideas in Curricula

In the last decade, computational thinking has become a basic literacy that all students should have in the United States [42, 25, 31, 61], sparking new curricula, and software that aim to provide effective computer science education to students of all ages. A result of this shift is a focus on learning experiences for novices that provide concrete examples to traditional computer science concepts. For example, the Media Computation course [68] is a **curricula** that introduces computer science to a population of people that might not otherwise be inclined to pursue it; digital artists. Furthermore, by concretely representing computing concepts through pictures and audio, novices are better able to understand the the effects of the code edits that they are making, reducing their cognitive load and therefore yielding better learning outcomes [110, 149].

Software such as Scratch [101] provides an online community where students are encouraged to share their projects with other members of the community. Once a project has been shared, members are able to "remix" projects; use the projects as starter code to build something new from. This sharing creates a society around programming in Scratch that encourages members to engage with each other and become enculturated into the community. Additionally, Scratch has a visual representation of code effects (similar to Media Computation), but also offers a visual representation of the code. Representing code in visual blocks, rather than text, makes the program more accessible to younger people,

and focuses novices on the logic of computer science, rather than the syntax.

Scratch, and environments like it such as Alice, are open environments that allow for creativity and self-direction. Scratch's "primary goal is not to prepare people for careers as professional programmers, but rather to nurture the development of a new generation of creative, systematic thinkers who are comfortable using programming to express their ideas" [121]. When paired with a well-designed curriculum and computer science teacher, Scratch can enculturate and teach children who, after only a couple weeks, are able to build large, complex games and simulations [92]. However, without the addition of a textbook or guidance from an expert, novices are not often engaged in *deliberate* practice [53]. On the other hand, online environments such as Codecademy are *extremely* guided, which can limit novices' skills in writing code without the help of a tutorial [23, 24].

Leveraging the social *and* visual embodiments of computer science through curricula and software is being done in the examples provided above, however they require additional support from instructors. This proves to be a problem when the learning experiences begin to scale, as we will see in the next section.

1.3 The Computer Science Culture Crisis

Though the increase in popularity of computer science at all ages has had a very positive impact on our society [38], it has also caused student-teacher ratios to skyrocket [30]. More students are eager to learn to program, requiring a more scalable learning experience. Because apprenticeships by design are small groups that support highly interactive learning experiences, it is difficult to mimic apprenticeship when the student-teacher ratio begins to grow.The question this dissertation takes up is, **How might we scale, but still keep the benefits of apprenticeship?**

Specifically, I am interested in two large groups of novice programmers: undergraduates and children. Even by focusing on these two groups, there are a variety of places

Table 1.1. Students learn in many types of settings: this dissertation explores the three bolded ones.

	Undergraduates	**Children**
In-Person Classes	**Large University Class**	**Class taught by Non-Computer Science Teachers**
	Small University Class	Class taught by Computer Science Teachers
Online Classes	MOOCs (Massively Open Online Course)	
	cMOOCs (Community and Connections)	
Software	Novice Programming Environments	Novice Programming Environments
	Tutorials	**Scaffolded Learning Software**

and ways to learn computer science, and each one is being affected by the rapid growth. Table 1.1 highlights some of this variety.

Each learning experience shown in Table 1.1 represents an entire research field. In this dissertation I focus on the first row: undergraduate students in a large university class and children in a class taught by non-computer science teachers using scaffolded learning software, software that encourages novices to interactively engage in deliberate practice and provides the support to continue learning [129, 1, 53].

1.3.1 Undergraduates in Large Classes are Missing Culture

Formal learning institutions demonstrate that they *value* enculturation through MFA and Doctoral programs. In computer science, doctoral programs typically support legitimate peripheral participation, apprenticeship for incoming students by engaging Master's level students in pieces of larger projects led by advisors or senior students [156]. The journey is full of situations where they learn how to conduct research; developing a thesis, forming a supportive argument, and testing the thesis. This period also typically involves communicat-

ing this work to the research community, learning the language and discourse of the field. A doctoral program ranges from 4-6 years, and in that time students increase their role within the community, until they can demonstrate a contribution worthwhile to that community; written and presented in a dissertation. However, doctoral programs typically have a low student-teacher ratio, making this model possible.

On the other hand, undergraduate computer science enrollments have been increasing [163] *and* computer science is becoming a general education requirement for some schools and majors [29]. This is causing introductory computer science (CS0 and CS1) enrollments to drastically increase and results in the design of undergraduate education in computer science to lose the "apprentice-like" experience. At large institutions, Bachelor's students have limited interaction with the "master craftsman", or professor. Wenger claims that this is because those who designed the learning experience act on the belief that "learning is an individual process, that it has a beginning and an end, that it is separated from the rest of our activities, and that it is the result of teaching" [158].

Due to the large student-teacher ratio, graduate and undergraduate teaching assistants are widely used to lower the ratio. However, there is not enough support to transform 500-student classes into small apprenticeship-sized groups, which makes experiences outside of the classroom even more critical to the education of undergraduate students. Mentorship programs and industry internships are widely used as apprenticeship opportunities for students. Companies such as Microsoft and Google even provide internships for first and second year undergraduates [108, 66], because they recognize the need to enculturate throughout the learning experience.

Though external resources are available, and popular, universities still offer courses as a primary teaching agent. As a result, transformations to curricula for introductory programming courses have been developed by many computer science education researchers such as Garcia et al. at UC Berkeley [65], Guzdial et al. at Georgia Tech [68], and Simon et

al. at UC San Diego [139]. Chapter 2 dives deeper into one lecture intervention developed at UC Berkeley by Clancy et al. that enculturates undergraduates, but is not easily scalable.

1.3.2 Children Entering Computer Science Don't Have Access to Computer Science Teachers

Multi-subject teachers are not typically trained in computer science, and even single-subject teachers are rarely trained in computer science; in part because it is not a core subject and has the possibility of being cut in a fiscal crisis [160]. With the lack of trained computer science teachers, software itself needs to deliver the expertise of programming skills while also facilitating enculturation since the teacher does not necessarily have the domain knowledge. One goal of this work is to empower K-12 teachers to effectively teach computer science, without extensive training, by providing a well-designed learning environment for students.

1.4 An Approach: Integrating Enculturation into Skills Learning

In this dissertation I contribute improvements to computer science education for undergraduates and children. I have based my research on the theoretical framing of apprenticeship, which I introduced in Section 1.1 and more deeply explore in Chapter 2. Scaling apprenticeships is not straightforward; to explore scaling strategies I created a course where undergraduate students in large, university classrooms can effectively engage in computational thinking and authentic practices both in lecture and at home on their own. I also designed and evaluated an educational software program for children to learn programming and learn to think like a computer scientist. Based on my contributions and the theories behind them, I develop an initial framework for teaching novice computer scientists at various ages and with various programming languages. The contributions I make can be applied to new scaling attempts in the future (see Section 11).

The contribution of this thesis is two-fold: an approach for integrating enculturation into skills learning for undergraduates in a large university classroom and for children who do not have access to a computer-science teacher. **My hypothesis is:**

> **Advances in our understanding of learning, the subject of computer science itself, and computing automation can integrate enculturation into novice learning experiences and scale it to match large student-teacher ratios.**

I tested this hypothesis in two different settings:

Undergraduates. I transformed a large undergraduate lecture by integrating and designing a *Peer Instruction* pedagogy and I transformed the time spent outside of lecture through the design and implementation of *Exploratory Homeworks*, which enculturate students without an expert (See Section 1.5).

Children. I designed and built a serious game, *CodeSpells*, that enculturates and engages children in developing programming skills, particularly in the absence of a computer science teacher (See Section 1.6).

1.5   Improving Undergraduate Novice Experiences: Re-designing a CS0-Level Course to Include Enculturation

In 2010, UC San Diego sought to re-design the CS0-level computer science course. The course entitled *CSE 3: Fluency in Information Technology* was one of the first pilots courses under the CE21 NSF Grant for the Advanced Placement Computer Science Principles course [61, 155]. The AP CS Principles courses "seek to broaden participation in computing and computer science" [41]. Along with Dr. Beth Simon and Dr. Quintin Cutts, I redesigned CSE 3 to meet the goals set out by the College Board and NSF.

As described above, it is critical for novices to have legitimate peripheral participation when entering a field. We therefore created a course that aimed to enculturate

approximately 500 students at once by implementing a technique called *Peer Instruction* [39]. In a Peer Instruction course, students must complete pre-class reading. Though pre-class reading is common, PI tests the students completion of the reading at the beginning of class time with a short quiz. Following the quiz, class time is a series of multiple-choice questions interspersed with mini-lectures. For each multiple-choice question, students are required to vote individually, then discuss and vote as a group of 2-3. A class-wide discussion and mini-lecture then follows each question.

The first setting where I test my hypothesis is by transforming a large university classroom (CSE 3) to include enculturation.

To understand how the design of CSE 3 affects novice students in CSE 3, we first conducted a pilot course where we observed student behavior and experience. Chapter 3 reports on the initial development and deployment of this pilot course. The course is designed to introduce core computational concepts and instill computational thinking practices. We report on the initial offering with 571 university students, mostly non-CS majors taking the course as a general education requirement. We discuss the instructional design supporting the course; describe how the various components were implemented, and review student work and valuation of the course. Though the course appears to "teach programming" in Alice, students reported gaining significant analysis and communication skills they could use in their daily lives. We reflect on how instructional design decisions are likely to have supported this experience and consider the implications for other K-12 computing/IT education efforts as well as for regular CS1 courses.

The Peer Instruction questions and pedagogy that we implemented in CSE 3 were preliminarily validated in our pilot study and in Chapter 3 show that students who take this course experience a change in confidence, appreciation, problem solving, transfer and communication in computer science. To further test the effectiveness of this intervention and re-design, I then evaluated it from the students and course design point of view.

1.5.1 Evaluating the Peer Instruction Intervention from the Students' Perspective

In Chapter 4 we begin to understand how Peer Instruction enculturates *students* in a large lecture environment in computer science. Studies have shown Peer Instruction (PI) in computing courses to be beneficial for learning and retention, however study of the student experience has been limited to attitudinal survey results [115, 137]. We report on a study that provides a preliminary evaluation of student experiences in a PI course. We asked students to reflect on their role as a student in a PI lecture compared to a standard university lecture, then analyzed their written responded using Chi's Interactive-Constructive-Active-Passive (ICAP) framework which categorizes student activities by their value in a constructivist learning framework. We found that the majority of students reported activity in a PI lecture as "interactive" in contrast with "active" (e.g. taking notes) in a standard lecture. Additionally, a grounded theory open-coding analysis found that while students positively value learning-related aspects (feedback and increased understanding), a surprising breadth of value was noted around issues of affect and increased sense of community.

1.5.2 Evaluating the Peer Instruction Intervention from the Course Design Perspective

Chapter 5 describes a post-hoc analysis of our CSE 3 lecture materials. The purpose of this analysis is to identify what knowledge and skills we are asking students to acquire, as situated in the activity, tools, and culture of what programmers do and how they think. The specific materials analyzed are the 133 Peer Instruction questions used in class. This open coding yielded an *Abstraction Transition Taxonomy* for classifying the kinds of knowing and practices we engage students in as we seek to apprentice them into the programming world. Throughout the course, students engaged in three levels of abstraction: English, CS Speak, and Code. Moreover, many questions ask students to transition between levels of abstraction within the context of a computational problem. For example, Figure 1.1

Relational Operation Expression:
How would I write a relational operation expression that returns true when an igloo is blue?

A. whichIgloo . color == [ ]

B. whichIgloo . color != [ ]

C. whichIgloo set color to [ ] more...

D. None of these, you need an operator like < or >

Figure 1.1. Example MCQ: Transition from CS Speak to Code.

demonstrates a formative PI question that asks the students to transition from CS Speak
to Code. Interestingly, applying our taxonomy in classifying our final exam demonstrated
that we tend to emphasize a small range of the skills fostered in students during the forma-
tive/apprenticeship phase. This disconnect is troubling for both fairness and learning, in that
we are not testing what we teach or what we would like to measure.

1.5.3 The Design, Implementation and Evaluation of Exploratory Homeworks: An Out-of-Lecture Intervention

The classroom lecture is only one of the resources students commonly have for
learning in higher education. In particular, PI tests students understanding of pre-cass reading
at the beginning of each class and professors assume students have completed the reading,
and therefore do not re-teach it. In Chapter 6, we introduce *exploratory homeworks* – a
framework to support active learning for teaching programming languages. By interactively
exploring the language concepts that they are reading about, misunderstandings can be
immediately identified and corrected, in contrast to waiting until the next programming
assignment or lab session days later. By leveraging the ability for the student to interact

with the computer/compiler, we provide a *culturally authentic model* for students of how to explore and understand programming language constructs and concepts introduced in the reading.

We report on the use of 15 exploratory homework assignments used in CSE 3 with 440 students. We provide a model and advice for others wishing to develop exploratory homeworks for their programming courses and present quantitative and qualitative evidence regarding students' positive valuation of the homeworks.

1.6 Integrating Enculturation into Learning Environments for Children

In 2011, Stephen Foster and I sought to engage children in learning to program through an immersive learning environment. Our goal was an environment that did not require an expert computer scientist to motivate, teach, enculturate, or guide the child. Though this goal is difficult to achieve, I present the first iterations of the design, implementation and evaluation of our environment and show that children can become enculturated while learning Java through our environment.

Our environment is a 3D immersive video game called CodeSpells. In this game, players take on the role of an apprentice wizard, they engage in quests put forth by the gnomes of the land they have come to save. To solve the quests, the players must write programs that interact with the world around them. We chose to use a video game so that we could build on the intrinsic motivators that games provide [60], we chose to use the metaphor of magic and wizardry to build on the popularized view that programming is magic [18], but magic is also learnable [125].

In the beginning we designed a basic sandbox game where players could walk around, interact with three gnomes, and write and cast three spells. **To evaluate enculturation within the game, we categorized expert programmers' experiences when they were**

Table 1.2. Theoretical Framework used to examine novice programming environments.

Category	Ways of Describing
Learner-Structured Activities	Self-driven, Access to immediate feedback, Access to support
Exploration/Creativity /Play	Creation of "meaningful" artifacts, Investment in the results of the code
Programming as Empowerment	Enjoyable/emotional connection, confidence/belief that you can succeed, wrongness is on a spectrum; not binary
Difficulty Stopping	"Flow-state" while programming, Feeling addicted
"Countless" Hours	Lots of hours/re-visitation, Losing track of time

novices (see Table 1.2) and applied that categorization to the sandbox version, as well as the designs for future iterations for CodeSpells. In Chapter 7 we take a step toward illuminating the critical qualities of novice computer science learning spaces by engaging in a grounded-theoretical examination of first-hand accounts of novice learning as told by computer science experts. To further study the topic, we attempted to recreate (in the lab) a learning environment with many qualities that characterize non-institutional learning. To make this possible, we employed a modified version of *CodeSpells* - a video game designed to teach Java programming in a way that engenders the sense of sustained, playful, creative exploration driven entirely by the learner. The study introduced forty girls, ages 10 to 12, to programming for the first time. Table 1.2 demonstrates native programmer experiences and, when integrated into learning experiences provides an authentic cultural experience for novices.

We found that by **leveraging the affordances of 3D video games, such as persistence, we could enhance enculturation of novice programmers.** However, mimicking these categories isn't simple. The **story and metaphor** of the game, and the **activities and challenges** posed to the students, if well-designed, integrate the expertise of computer science into CodeSpells so that the teacher does not need to have domain knowledge (see

Figure 1.2. Students are presented with a series of quests that ask them to write code to affect the environment, such as levitate a crate to collect bread.

Figures 1.3 and 1.2).

1.6.1 Wizardry is an Effective Metaphor for Computer Science and Programming in a Video Game

Based on what was learned in the prototype study in Chapter 7, we iterated on the design of CodeSpells and focused on supporting the embodiment of the players. Massive multi-player online (MMO) games have been shown to enculturate players [144] and a multi-player version of CodeSpells found similar results [60]. Chapter 8 discusses enculturation embedded in a single-player version by reinforcing the metaphor of magic and wizardry throughout the game-play. We address how CodeSpells uses the metaphor of wizardry, along with an embodied API to engage students in learning to program in Java. Giving novice programmers a visual and concrete representation of code, such as modifying an image or moving objects on the screen, helps students understand the concept more easily than abstract representations, such as sorting an array of numbers, but an *embodied* API personalizes the experience even further. In our embodied API the code affects the player
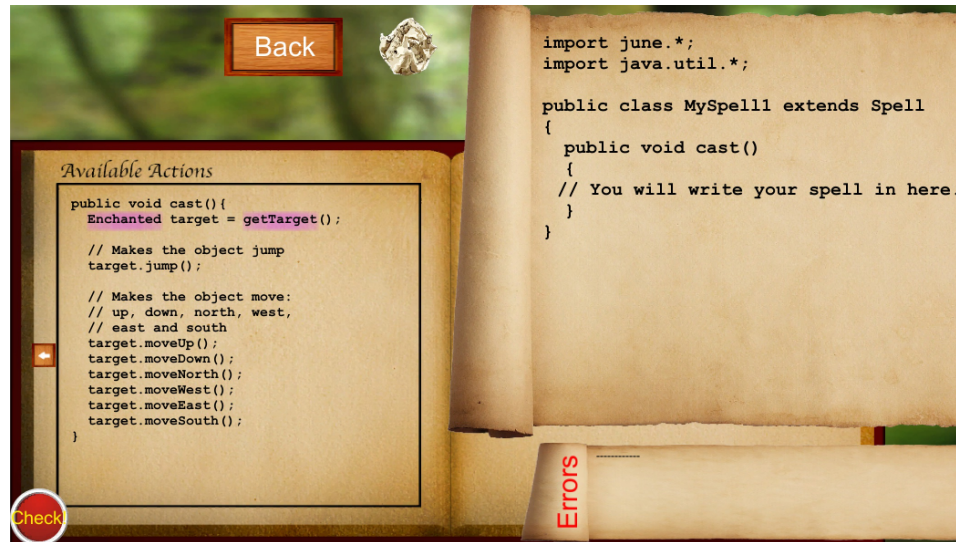
Figure 1.3. Students engage with code in a simple IDE where they have access to textual code references.

and the world around them. This is an improvement on environments like Scratch or Alice where the code affects a sprite because increased embodiment has been shown to improve learning [96].

There have been many attempts to improve the novice learning experience by providing: a visual programming language [101, 46], a hardware component [90, 2] or an application that is more approachable [68]. CodeSpells' contribution beyond previous work is that students are better able to understand how abstract code affects the environment.

CodeSpells builds on previous work by immersing novices in the abstraction of code through embodiment to allow them to understand complex and abstract programming problems as if *they* were being affected by what they wrote. We present a new approach to novice programming environments, one that embodies the user and encourages a quick grasp of introductory concepts followed by a deep understanding through exploration.

1.6.2 In-Game Activities that are Designed to Guide Skill Learning and Enculturation for Children

The discussion has focused on the enculturation of children, Chapter 9 continues with the skills development side of programming through CodeSpells. This Chapter presents a study conducted using CodeSpells with a focus on the design and evaluation of quests (scaffolding to support students in learning). The study analyzed how 16 students aged 8-12 understood and modified basic Java programs to complete programming exercises. Based on game-play from an exploratory study, quests were added to engage students *earlier* and in *more complex code edits*. Both student understanding of programming and their comfort with modifying code was studied. I present the findings and lessons learned in quest design, and show that quest design should set the expectation for students to *engage* with the code, not just use the code. Therefore, the in-game activities that guide code-editing and writing must *require* code-production so that students cannot play without producing any code.

Since code-writing is a major goal of CodeSpells, we determined that the design of in-game activities that guide code-editing and writing result in novices actually writing code. Similarly, when in-game activities *fail* to scaffold code production, students find a way to play without ever achieving that goal.

1.6.3 Children Can Learn Skills and Become Enculturated While Playing CodeSpells

Thus far I have presented results in designing a metaphor or magic and code-producing quests. Chapter 10 combines the results presented in Chapters 7 through 9, and implements a modification to the quests that *requires* code writing for *all* quests which shows through an 8-week study that students can learn to program and become enculturated through game-play.

The 8-week study was conducted on fifty-five 9-10 year old students across two different schools. Throughout the study, students not only played CodeSpells, but also used

a guided workbook to explore Java code outside of the CodeSpells virtual environment. Through both immersive interactions and the guided workbook, students demonstrated their understanding of introductory computing concepts and their ability to program in Java, both on the computer, and on paper. Taken together, Chapters 6-10 show that a 3D role-playing video game, employing an embodied metaphor and quests that demand coding provide the elements of novice enculturation missing from traditional curricula.

1.7   Discussion and Future Work

The results from the preceding chapters, while powerful, are really just about two interventions, the question remains about the generalize-ability of these contributions. In Chapter 11, I show how the two initial interventions have potential to generalize past the two settings I present in this dissertation. Specifically, based on the results found in Chapters 3 through 6, Section 11.1 explores how they could be applied to higher level computing courses, not just introductory courses. Similarly, I explore how to apply what was learned in Chapters 7 through 10 to other scalable learning environments. Before delving deeper into the research of this dissertation, I will provide the background that supports this work.

Chapter 2

Learning and Enculturation

This dissertation contributes two frameworks for designing novice learning experiences in computer science that are grounded in how people learn. In Chapter 1 I briefly described how enculturation plays a major role in novices gaining expertise. In Chapters **??** through 10 I provide background research to support the contributions of each chapter. In this chapter, I explore the theoretical background that supports enculturation and examples in computer science. I focus on How People Learn [15] and Legitimate Peripheral Participation [87] because How People Learn is a survey of learning literation compiled from many disciplines and Legitimate Peripheral Participation is a theory that clearly demonstrates apprenticeship.

2.1 How People Learn: An Education Shift from Literacy to Critical Thinking

In 2004, Bransford, Brown and Cockling compiled and edited "How People Learn" [15], a book that collects recent developments in learning theories from various disciplines and develops a cohesive understanding of how people learn. From the very beginning of this book, they address the focus of this dissertation:

> Work in social psychology, cognitive psychology, and anthropology is making clear that all learning takes place in settings that have particular sets of cultural and social norms and expectations and that these settings influence

learning and transfer in powerful ways. [15]

Bransford et al. explain that formal education has historically focused on literacy skills (reading, writing and calculating), but with societal developments there is a desire to shift to teaching critical thinking and problem solving skills [15]. The learning experience for students shifts to learning how to find information, not just understand facts. This process is complex, students have to learn how to ask the right questions, understand the relationship between the questions they are asking and the information they are receiving, and draw conclusions to inform their learning. Essentially, the goal in changing their learning experience is to contribute to "individuals' more basic understanding of principles of learning that can assist them in becoming self-sustaining, lifelong learners" [15]. Though knowing facts is critical to problem solving and thinking critically, expertise requires that students develop knowledge that is specifies context to support understanding and transfer to other contexts, rather than "only the ability to remember" [15].

Furthermore, experts differ from novices in that they "have acquired extensive knowledge that affects what they notice and how they organize, represent, and interpret information in their environment" [15]. This supports the notion that "Learning is influenced in fundamental ways by the context in which it takes place, [therefore] a community-centered approach requires the development of norms for the classroom and school, as well as connections to the outside world, that support core learning values." Essentially, expertise is achieved when skills and culture are integrated, culture representing the norms, values, experiences, and contexts of a field.

In the next section I describe how the theory of Situated Learning, as described by Lave and Wenger, can be used as the foundation for shifting to a more critical thinking learning experience.

2.2 Legitimate Peripheral Participation Supports Critical Thinking Development

As described above, learning should specify the contexts in which it is applicable and depends on the development of community and culture within that discipline. Jean Lave and Etienne Wenger introduce Situated Learning and Legitimate Peripheral Participation that provide a framework for shifting the learning experience to be focused on critical thinking [87] .

2.2.1 From Apprenticeship to Situated Learning: Legitimate Peripheral Participation

Situated learning is a theory that is far more complex than a series of apprenticeship empirical cases. Though apprenticeship is a useful metaphor for representing the goal of situated learning, Lave and Wenger developed a cognitive and cultural theory that encompassed more than what was observable in historical examples. They defined learning, similarly to Bransford, et al., emphasizing "comprehensive understanding involving the whole person rather than 'receiving' a body of factual knowledge about the world; on activity in and with the world; and on the view that agent, activity, and the world mutually constitute each other" [87]. Furthermore, this supported their claim that *all* learning is situated "in the lives of persons and in the culture that makes it possible" [87].

2.2.2 Full Legitimate Peripheral Participation is Expertise

Lave and Wenger explain that each word in 'Legitimate Peripheral Participation' is required throughout the *entire* journey of a member of a discipline.

- **Legitimate** is what makes the participant *belong* to the community, defining a critical performance to that community.

- **Peripheral** implies that there is no center of the community, because it is a *community*,

and there isn't a way to know **all** the knowledge of the community, because it is ever changing and growing, particularly with the introduction of new members.

- **Participation** is required for learning to occur, which is supported with the theory of "learning by doing".

As a novice to a discipline continues to interact and participate within the community, they become a full legitimate peripheral participant, where they have expertise in some areas, but might be novice in others. The goal is to understand how to engage with the community, how to discover and grow the area, how to communicate effectively with other members, and how to guide newcomers into the community.

Thus far I have presented work that generally defines effective novice learning experiences. Lave and Wenger define learning as "a theory of social practice in which learning is viewed as an aspect of all activity" [87]. Furthermore, legitimate peripheral participation was not intended to be an educational form that would transform schooling, rather a pedagogical strategy for transforming learning. Both contributions have aimed to deepen the learning experience for novice student; therefore concrete examples of these approaches are necessary for evaluating them and improving learning for novices. I apply these literatures to teaching computer science to undergraduates and children in a problem-solving-based curriculum and software. In the next section I will focus on transformative, concrete examples in computer science education research that align with the interventions presented in my dissertation: novice undergraduates with access to a computer science teacher and novice children without access to a computer science teacher.

2.3 In-Person Undergraduate Computer Science Courses *Can* Integrate Enculturation

Undergraduate, in-person learning experiences in can be hugely beneficial to students because a "master", professor, is physically present and can guide the "apprentice" to develop

Figure 2.1. In elementary school, students often have places where they can work collaboratively and be interactive with their curriculum.

the skills and culture of the discipline. However, in-person education is difficult to scale; it is limited by space, time, and people. For example, large undergraduate institutions have designed large lecture halls that encourage information delivery: students all sit in rows facing the front, acoustics are designed such that they can hear the professor who stands at the front of the classroom looking towards them, and though boards are not always visible from the back, large monitors can be seen from all around the room. This physical design of undergraduate lecture halls is often restrictive to collaboration and interaction amongst students; promoting, instead, a teach-centered environment where students are expected to "retain" information and apply it on exams (see Figures 2.1 and 2.2).

Numerous studies show that traditional lecture-style courses that do not engage novices in activities that would enculturate them yield lower learning results than courses that are student-centered and include enculturation [139, 128] One approach to engaging students in deeper understanding is to change the teaching technique used. Particularly in higher education, various collaborative pedagogies have been employed and studied.

As Titterton et al. show [151], lac-centric courses improve student learning through interactivity. Though an outstanding issue is scalability, their findings are insightful and

Figure 2.2. The design of university lecture halls encourages students to listen to the professor, limiting collaboration and interactivity.

can be applied to new contexts that *are* scalable. Titterton, Lewis and Clancy describe their experiences engaging students in a lab-centric course in computer science and show that students benefit from a constructivist, hands-on learning experience [151].

The course that they designed engaged the students in one hour of lecture and six hours of closed-lab each week. Closed-lab is where students are expected to complete a certain amount of tasks in a supervised lab in a certain amount of time. Labs are typically led by teaching assistants, while lecture is typically taught by the instructor. Lab-time employed a series of activities to engage the students, including:

- Quiz - Typically at the beginning of lab to test knowledge learned from the previous class.

- Gated Collaboration - A question posed online; after answering the question, students can see responses made by their classmates.

- Self Assessment - Embedded assessments with instant feedback.

- Exercise - Activity (programming or debugging) with no explicit assessment.

- Brainstorm/Reflection - A prompt to encourage reflection on the exercise.

- Design/Defend discussion - Evaluation and refinement regarding design.

- Discussion.

- Project - A three- to four-week extended programming project

Table 1 in Titterton et al.'s paper describes each of these activities in more detail [151].

Titterton et al. provide a pedagogy that "fosters a kind of apprenticeship learning" and "builds a community where we strive to avoid pitfalls present in other computer science classrooms, such as the defensive climate" [151]. These support the goal to provide novices with a sense or community and culture that integrates into their learning.

However, there are a number of constraints that can arise with this format, such as an increased need for lab staff, an increased workload for the course instructor, and a need for extensive lab space. Considering an issue facing computer science is an increase of students, these constraints are only magnified. Though it is difficult to scale this exact model, there are many lessons learned and best practices that can be drawn from it. The main take-aways that apply to the integration of culture for novice computer scientists are:

- Providing a space for collaboration

- Conducting formative assessment throughout the learning experience

- Structuring and scaffolding programming

Each of these take-aways has many other examples that support their importance in students' learning, as described by Titterton et al. [151]. The difficultly lies in executing an in-person course that can be offered to hundreds of students at one time with limited teaching staff in a traditional lecture-hall. The intervention that I designed, implemented and evaluated as part of this dissertation follows similar designs to the course designed by Titterton et al., however it scales while retaining similar learning outcomes. Chapters 3 through 6 more deeply explore each aspect of my intervention, including related work that supports each piece.

2.4 Software Programs for Children Learning Computer Science *Can* Account for Lack of Teachers

In 2014 only 2% of public schools offered AP Computer Science and only 4% of public schools offered *some* kind of computer science course (see Figure 2.3), meaning nearly 10 million high school students do not have access to *any* computing course in school. Furthermore, middle schools and elementary schools rarely offer computer science, and when it is offered is it often a part of a university-led intervention that ends when the grant expires. The cause of this major issue is the lack of trained teachers at these lower levels. In 2010 NSF sought to tackle this issue with the CS10K Project [150], however they quickly realized that the money and time it takes to effectively train a K12 teacher to teach computer science (or computational thinking) is beyond the reach of the NSF and public schools' funding and support.

Software programs are being developed to better support children learning computer science since training teachers is not easily scalable. These software programs range from tutorial-based applications such as Codecademy [23] to sandbox-like environments such as Scratch [101].

Scratch [101] is an online programming language and environment for novice programmers. It was originally designed for children to increase the accessibility of programming environments for kids. One Scratch's biggest contributions is its collaborative, community-driven online presence (see Figure 2.4. Scratch users are encouraged to build programs of their own with a visual, drag-and-drop interface (see Figure 2.5), share with the entire Scratch community (see Figure 2.6) and even remix other users projects (see Figure 2.7). These features align with community-driven learning environments, as described above.

Scratch integrates culture and community into its programming experience through a website that provides millions of users the ability to share, comment on, and further engage

US Public Schools

- High Schools that don't offer CS
- HS that offer AP CS
- HS that offer CS
- Middle Schools and Elementary Schools
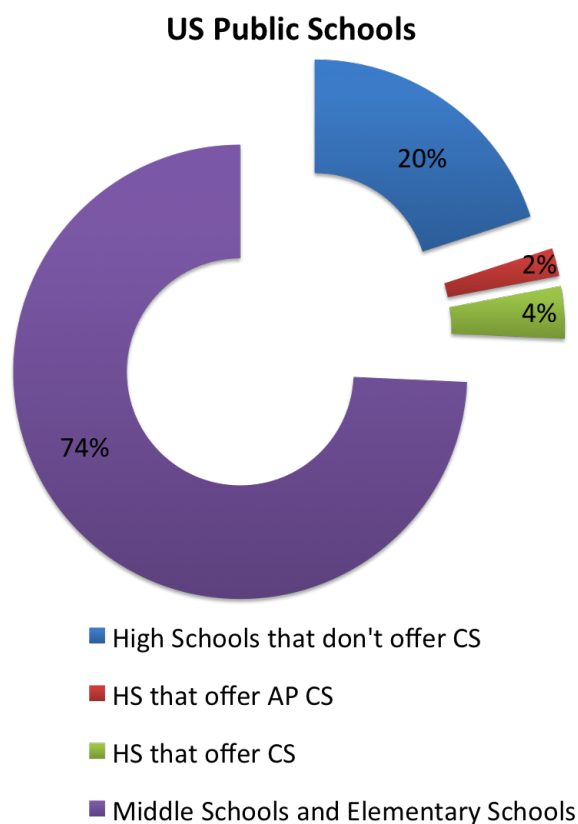
Figure 2.3. There are few computer science courses offered in public schools in the United States

Figure 2.4. Scratch homepage highlights the collaborative community.

user-written programs [131]. To date, the empirical literature on Scratch shows its success in classrooms [107]; there are no published studies of unstructured learning. The intervention I created as part of this dissertation, CodeSpells, expands upon the efforts of Scratch to further engage children in skills development, as well as enculturation. Chapters 7 through 10 explore each feature of CodeSpells and the specific features pertaining to prior work.

2.5 My Two Interventions: Next Steps in Computer Science Education Research

I seek to combine interactive learning strategies to scalable platforms for teaching novices computer science. I have designed scalable interventions for undergraduates and children, while retaining skills development *and* enculturation.
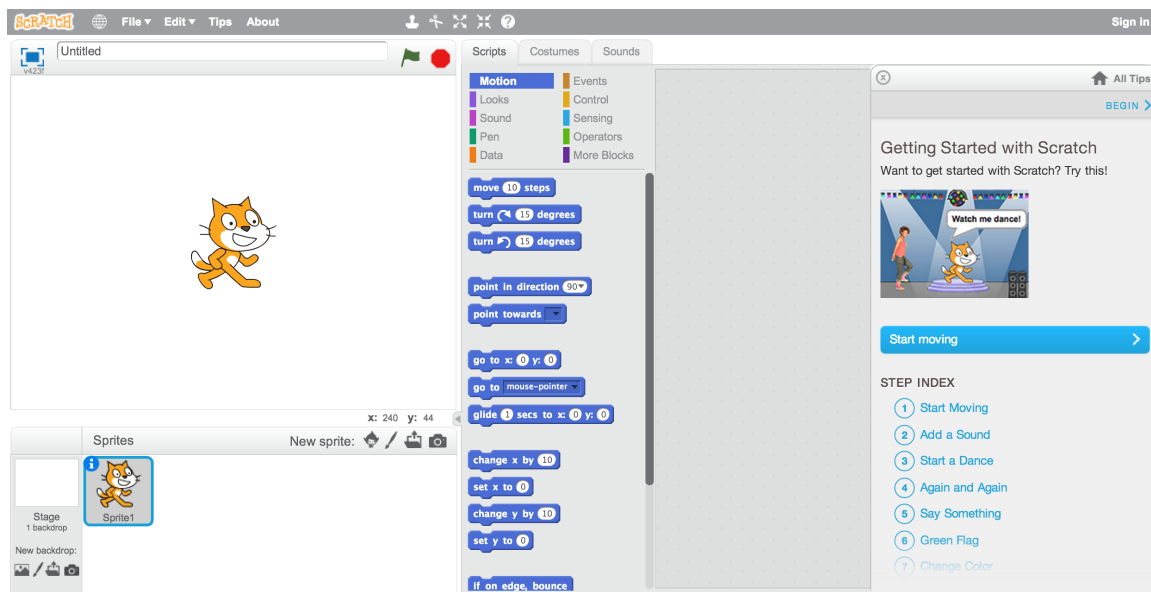
Figure 2.5. Scratch's interface offers a colorful, easy-to-learn programming language that doesn't rely on typing abilities. It also users students with tips to highlight software features (seen on the right).
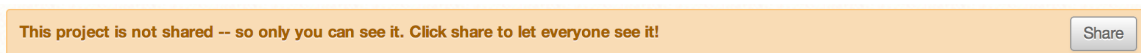
Figure 2.6. Scratch encourages users to share their projects with each other, forming a community of users.
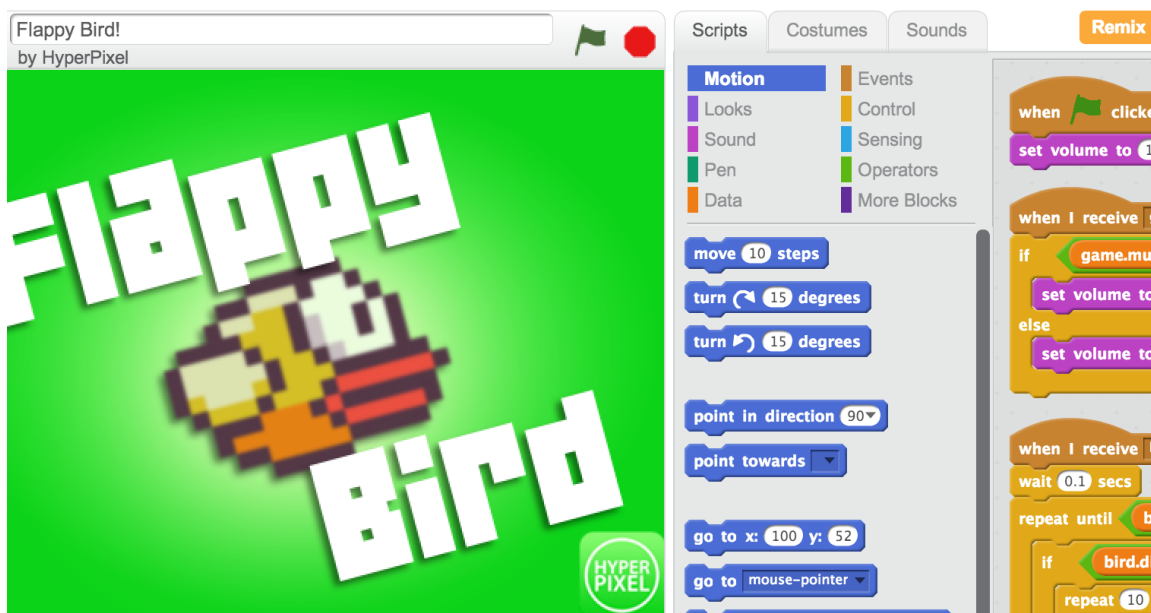
Figure 2.7. Scratch encourages users to view other users' projects and edit them and make them their own.

Chapter 3

A Model Course to Enculturate Novices in Computer Science using *Peer Instruction*

This chapter introduces *CSE 3: Fluency in Information Technology*, a course we designed to enculturate non-computer science undergraduate students in into the field, and sheds light on how any novice might become enculturated into computer science while learning programming skills. Specifically, we describe a) the instructional design and implementation and b) the student experience of a general-education computing course at the University of California, San Diego.

Computing education is seen as increasingly important, with Wing and others arguing for a grounding in fundamental computational principles for the entire population [161]. Actions are being taken to address this. For example, the UK Royal Society has been commissioned to report on the state of computing education in UK schools [141], and the US National Science Foundation and the College Board are supporting the development an Advanced Placement course, CS Principles, which aims to "broaden participation in computing and computer science" [41].

In Fall 2010, we ran a pilot of the CS Principles course at a US R1 institution. One of five pilots, ours was unique in that it served the needs of a university general education (GE) course for 570 students. This raised the question of how a GE computing course should

be defined, or, put another way, what should every person know, assuming they never take another computing course? In this report, we tell the story of our experiences in putting together a GE course grounded in the CS Principles framework, and of how it impacted students and our views of GE computing.

As we taught the course we paid close attention to the student experience, with all authors attending all lectures and listening directly to students' discussions in the class. When prompted (in Week 8), *the vast majority of students self-reported a range of long-term gains* as a result of taking the course. Analysis of their open-ended responses shows students reporting increased confidence, changed views of technology in the world around them, increased technical problem solving skills, transfer of computing skills to other areas of their life and increased communication skills. Reports of "learning to program" were very rare (and mostly limited to computing majors). These results were very compelling to us as computing educators, with the following examples being representative:

We learned in Alice that computers do exactly what you have them do.

Programming allows a person to think more logically, thinking in order and debugging allows the user to gain valuable problem solving skills. Aspiring to go to law school, thinking logically is extremely important and I think this has helped.

It has given me confidence that I'm able to figure things out on a computer that I never would have thought that I could do.

We will argue that *the gains reported by the students form an excellent definition of a general education course in computing.* Notably we believe that engaging students in "learning programming" is critical to the experience - as it provides students very direct control over the computer. Finally, we draw conclusions on the ordering of computing courses at the introductory level to maximize opportunity of access to computational thinking skills development for all students.

3.1   Instatiating a CS General Education Course

Here, we briefly overview the instructional design of our course - though we refer the reader to full details in citepilot. Our course was based around:

- existing university needs for an academically-rigorous digital literacy course involving logical thinking and the ability to create digital artifacts in subsequent courses,

- the CS Principles framework, particularly the six defined computational thinking practices of analyzing effects of computation, creating computational artifacts, using abstractions and models, analyzing problems and artifacts, communicating processes and results, and working effectively in teams, and

- published experiences in teaching CS0-type courses.

It included 7 weeks of Alice programming [34] and 2 weeks of Microsoft Excel. Alice is a beginners' graphical programming environment: graphical both in the sense that programs create and manipulate 3D worlds, and that writing programs consists of snapping visual tiles together on screen. Most critically, we designed the course around a best-practice pedagogy, Peer Instruction (PI) [104], to engage students in deep learning of computing concepts. We made this decision a) based on the evidence from physics and other disciplines that its use dramatically increases learning [39] and b) because it had worked well in previous programming courses [137]. We were particularly interested in PI's ability to focus students on *understanding how programs work*, not just getting them to work.

In the standard PI model, before class, students gain preparatory knowledge typically by reading the textbook and then complete a pre-lecture quiz on the material. In this class, we leveraged the ability to have students work directly in the Alice programming language to assign, exploratory homeworks - which guided them in reading the textbooks for understanding [138]. During class, lecture was largely replaced by a series of multiple-choice questions (MCQs). These typically focused on deep conceptual issues, common

student misconceptions or problems [138]. Students followed a process by which they answered a question individually (using a clicker), discussed in an assigned group of 3, and answered a second time. This was followed by a class-wide discussion led by both the students and the instructor. This is the core of the PI pedagogy. Additional course components included a weekly 2-hour closed-lab programming assignment, one midterm, one final, and a multi-week Alice programming project ("make a digital contribution to communicate your views on an issue facing society").

We hoped the PI methodology, with its focus on analysis and discussion, would influence the students' experience positively. Rather than simply "playing around" with Alice, we believed that the PI activities would engage them in the authentic practices [19] that underlie computing experts' thinking and activities; that by asking them to analyze code and discuss it with each other, they would experience via legitimate peripheral participation what actually happens in software developers' cubical walls, or in the IT support center of a major company.

3.2   Student Experience

We sought answers to questions like: "What if this is the last computing course these students ever take? What are they getting out of it? Does this satisfy us with regards to what an informed populace should know?" We were teaching "programming" but our lectures focused students on analyzing programs to illuminate core concepts. We personally don't believe "ability to write a program" should be a GE course goal. We asked students about their experience in lab during Week 8:

Learning computing concepts may have opened many doors for you in your future work. Although you may not ever use Alice again, some of the concepts you have learned may become useful to you. Some examples include:

- *Understanding that software applications sometimes don't do what you expect, and*

being able to figure out how to make it do what you want.

- *Being able to simulate large data sets to gain a deeper understanding of the effects of the data.*

- *Understanding how software works and being able to learn any new software application with more ease, i.e. Photoshop, Office, MovieMaker, etc.*

Aside from the examples given, or enhancing the examples given, please describe a situation in which you think the computing concepts you have learned will help you in the future.

Though this question may seem leading, the topic had been discussed frequently in lecture, and the question needed to be explicit enough for 500+ students to clearly understand what "kind" of answer we sought. Students were informed that any thoughtful answer would receive full credit. A few responses did imply that students had already "been good" as using computers, and class hadn't changed that. Through analysis of this data, we consider students' perceptions of the "general education in computing" effect of the course.

3.2.1 Methodology

After preliminary, ad-hoc review of the responses (N=521) by two of the authors, one author developed a set of descriptive categories reflecting commonly observed themes. Next that author and one other separately coded a random 10% sampling of the dataset, discussed the results, and refined the categories and descriptions until reaching agreement on that sampling. Then both individually coded a new 10% sampling, reaching an 85% inter-rater reliability (counting matches for agreement on each code for each response). Then one of those authors and the third author coded the remaining data (with the third author reviewing the first 10% sample as a training set).

Table 3.1. How the Course Will Help in Your Future: Categories of Student Responses ordered by Prevalence

Category	Example Response
Transfer, Near (64%): can apply new skills in software use	*Using new machinery like sound editing equipment will require the ability to manipulate and design using the basic commands to form unique creation. Similar to Alice we will be restricted to the amount of actions we can perform sometimes but through our creativity we can manipulate the basic commands of the music program to create variations not standard to the system. Like how we mad[e] frogs appear to be hopping when in actuality the Alice program does not have a specific method that makes frogs hop.*
Personal Problem Solving Ability: Debugging (39%): can logic it out, attempt to or deal with unexpected behavior	*I have learned how to target problems when I am working on a computer and use the process of elimination to try to fix the problem instead of just restarting the computer like I used to. This skill partially developed from taking CSE3 and becoming more comfortable with working with new computer programs and dealing with bugs in Alice.*
Personal Problem Solving Ability: Problem Design (29%): can develop plan to solve technical problem, can see what requirements exist	*We learned in Alice that computers do exactly what you have them do. Using this knowledge, we can understand how programs like Excel and Numbers work and learn that when we are using these programs, we need to specify and be exact with what we are doing in order for the programs to meet our needs and plans.*
View of Technology (25%): greater appreciation or understand of technology	*Now, every time I find myself playing a video game, I actually understand what makes it work. That these games are not magically produced, that it takes time, skill, and sufficient funds to create these games. I appreciate these games more than before taking this class.*

Table 3.1. (Cont'd) How the Course Will Help in Your Future: Categories of Student Responses ordered by Prevalence

Category	Example Response
Transfer, Far (23%): can use problem solving skills in other areas of life	*I feel that learning the language of computing definitely helps you understand dense reading a lot more efficiently. I personally have noticed that my in-depth understanding of Computer Science wording has helped me understand my mathematical theorems and proofs more regularly than before.*
Confidence (21%): increased ability to do things on computer, a can-do attitude	*The things I learned in Alice can help me not to be so frightened in general when dealing with technology. Although I am not certain I have absolutely mastered every concept in Alice, I am certain that I have learned enough to bring me confidence to apply these ideas in the technological world. This is a big deal for me, as I do consider myself quite technologically challenged. I think this class has given me tools for life, that can be applied to both my life at home, socially, and at work.*
Communication (7%): communicate better about technology	*In today's technologically-centered world, using a program like Alice gives us valuable exposure to discussing things technically with other people and explaining clearly what we are trying to do.*

3.2.2 Results

The categories we identified are shown Table 3.1, along with prevalence (one response could be coded into more than one category, avg. 2.1), with an example illuminating the category.

3.3 Discussion

Overall, we were satisfied at the ways in which the students felt the course experiences had impacted them. We patently did not want students to think they were "made to learn programming" and we specifically tried to differentiate the course from one seeking to attract students into the CS major or prepare them to take another programming course. Although the content of our syllabus doesn't differ much from such courses, we utilized the course design to engage students in a different experience - specifically through the in-class peer instruction discussions.

3.3.1 The Student Responses Define General Education Computing

We argue that the students' statements form a core understanding about what general education in computing should be.

We recognize the students' descriptions as a set of transferable skills and attitudes: confidence to have a go with technology; a new appreciation/awareness of that technology; problem solving skills to plan out solutions to problems and then to enact them, detecting and correcting bugs along the way; and communication skills appropriate for discussing issues about computing systems.

To rate the value of these skills, consider the typical knowledgeable IT person, the colleague any office worker calls over when they're having trouble with their PC. He or she is the confident problem solver who can talk to you about your problem. Even though they may not know directly about your software or your issue, they know they'll get there with some educated exploration. Their skills and attitudes bear a striking similarity to those described by our students.

As to whether such skills should form part of a general education requirement, there are two pertinent questions: do all citizens need this skill/attitude set; and is it necessary to formally teach it? The recent push for a broader computing education indicates that

society is beginning to accept the importance of computing skills for all; and we use Turkle [153] to argue that a concerted effort is required. Turkle argues that the adoption of computing technology to support our thinking processes has in fact shaped the way we think. Specifically, the Apple Macintosh-style direct-manipulation interfaces introduced in the 1980s encourage us not to look under the surface and not to attempt to understand or appreciate systems deeply. She argues that we have been seduced into an expectation that systems will be easy to use and we are surprised and unprepared when they aren't.

As an example, consider a modern word processing package - a far cry from early, glorified text editors like MacWrite. The underlying document model of the modern version would have been the domain of a professional typesetter in years gone by, yet users expect to be able to intuit the model largely via direct manipulation with what they see on screen.

The combination of increasing complexity with incorrect expectations only leads to frustration. When software does something unexpected, most users have no training in how to go about understanding what is going on, and few skills in identifying or correcting the problems they are experiencing. Consequently, to them, software has become something magical and beyond their control.

We can relate each of our students' response categories to the manner in which Turkle's argument suggests most computer users are likely to think.

- Confidence: Software systems are too complex for me to understand. When they don't do what I want, I don't know what to do. I can't have an effect.

- Appreciation: I don't have any insight into how the technology works and I've never been encouraged to look "under the hood".

- Problem solving: Software and computers are meant to be easy to use I shouldn't need to plan ahead to complete my task; when something happens I don't expect, I haven't a clue where to start I have to get someone to help me.

- Transfer: I've only just mastered Word. Now I've got to start all over again with Excel. Nightmare! It's a different world.

- Communication: I can't get the IT person to understand my problem at all. It's as if he's from a different planet.

We suggest that through our GE course, students gain the ability to balance the inherent complexity of software against the knowledge that, with effort and use of appropriate skills, they can understand the software or "figure it out". In particular, they can understand the complex models underlying software via a process of inductive reasoning based on experimenting with the software.

3.3.2 Comparison with Existing "First" Computing Courses

In schools, there exist current courses that could possibly be viewed as a GE in computing, varying from training in the use of IT, through programming courses, to the introduction of computer science concepts. We assess whether these styles of courses are likely to deliver experiences that our students described.

Before beginning, we acknowledge Papert's early radical general intellectual training based around programming in Logo [112]. There is much commonality between the skills he describes his students developing and those described by our students. A key difference is that of scale: our students are in a traditional mass education system whereas Papert describes a more personalized self-exploratory learning environment.

IT Training Courses

IT Training is typically centered on the direct use of typical office-oriented packages. For example, the Scottish education system features a 5-14 Information and Communication Technologies (ICT) strand in its national curricula - traditionally involving follow-the-steps-style worksheets [67]. Assessment often features simple factual recall or production of

artifacts - and transferrability of skills is hard to assss. Crucially, such courses drive towards outcomes such as "I can create a PowerPoint presentation", rather than anything to do with the understanding of or communication about how to be an effective IT user. In a survey of over 2000 Scottish school pupils [109], it was clear that this curriculum was found to be both boring and a totally inappropriate forerunner to later computing courses. Worse, anecdotal evidence suggests that many incoming university students are barely-adequate IT users. Furthermore, contrary to popular opinion, Bennett [8] demonstrates that the evidence for Digital Natives [120] is far weaker than is widely reported.

Preparation for Programming Courses

These courses introduce the excitement of creating programmed artifacts without going into the traditional heavyweight detail of a standard CS1. Examples include courses that use robots or the Scratch [121], Alice, or Greenfoot [72] programming environments.

We are unable to ascertain whether students taking these classes have also experienced changes similar to those our students report though published work does not report such findings. In [111], students' attitudes regarding interest in computing increased in an elective Alice-based CS0 course. Our students were given the same survey, but no statistically significant increase in attitudes occurred - perhaps because students' interpretation of the terms in the questions changed from pre-test to post-test; perhaps because they did not choose to take the course and were not as likely to be pre-disposed to come to like computing. In future work, we seek to better understand this result.

We speculate that the focus in these courses is typically on the excitement of getting programs working, rather than on the deep understanding and articulation of what the students did. For example, in [121], the digital fluency associated with Scratch involves "designing, creating and inventing". Teachers of course do want the deep understanding, but much of the student activity and assessment, where there is any, is most-likely focused on "can you do it?" As Section 4.3 shows, we view the core difference between our course and

other programming-oriented courses is the emphasis on articulating deep understanding.

Non-Programming Introduction to Computer Science: Excite Programmers

There is a wide range of programs that aim to introduce computer science without involving programming at machines. The best of these is CS Unplugged [6], and author Cutts has run a similar effort called CS Inside [43]. Both the US and the UK are considering adopting aspects of these programs into nascent school curricula. We refer to these as *excite* programs, because a key aim is to excite participants about core aspects of CS in order to increase enrollment in future computing courses. Indeed, the origins of both programs lie firmly in university outreach activities. The activities of the programs were originally designed for one-off, non-assessed sessions where excitement is the core goal, with learning as a secondary goal. They do use active and often kinaesthetic learning methods that undoubtedly are highly engaging for the participants.

We speculate that the learning activities of these programs will not form an effective general education, as our students' responses define it, for a number of reasons:

- Their main focus is to raise awareness of a broad range of computer science topics, (e.g., data representation, algorithms, cryptography, intractability, etc.) rather than on a narrower core set of transferable skills and attitudes.

- Whilst the active learning embedded in the activities does foster core skills such as problem solving and group work, or core attitudes such as the deterministic nature of algorithms (and hence programs and computers), the rather self-contained nature of each learning activity goes against on-going step-wise development of these skills.

- Their separation from the world of software and machines is likely to make transfer of core generic realizations about the structure and use of computer systems difficult.

A Matter of Speculation

Here, we have only been able to speculate that alternative course formats considered for introductory computing do not effectively fulfill a general education role. We urge those teaching any of the formats covered here to replicate our open-ended reflection question, presented in Section 3, with their students. Particularly interesting would be the effect on students taking such courses as a requirement, as ours did, and not by elective choice.

3.3.3 Key Effects of the Instructional Design

The Peer Instruction Effect. We believe our instructional design centered in analyzing code (in homeworks, discussion questions in class, and (naturally) programming labs) impacted students. Certainly, instructors hope students in programming courses with standard lecture develop code analysis skills, but it is rare that we focus class time engaging students in that practice for themselves. Even in lab-based lecture environments, students' work with live programming may not engage them in analysis. As Stephen Cooper advised us [33], some students may just play around randomly trying things until they get the desired result. From our classroom observations (two authors observed and engaged students in their group discussions during lectures), the use of PI gave students the opportunity to viscerally develop the understanding that computers are, likely contrary to their previous experiences, deterministic, precise, and comprehensible. Through vigorous, constant engagement in the struggle to not just *create* programs or *learn to use* computing concepts like looping and abstraction, but instead to *analyze, debug, and critique* Alice code, students seem to have internalized these three core attributes of computational systems. We see evidence of this in some students' responses regarding their experiences when something goes wrong on the computer. They now recognize the problem might be the fault of the computer or it might the fault of the user. This stands in contrast to their stated previous beliefs that it was always their fault (or in some cases always the computer's fault). This seems a critical first step in

an increased sense of empowerment that should serve them positively in their futures.

Furthermore, the general education literature provides strong evidence in support of the PI process as a way of promoting deep learning. Teasley [148] demonstrates that speaking out one's understanding improves learning; articulating it to a peer even more so. Craig et al. [37] show that paired learners gain as much from watching a video of a tutor at work with another student as from one-to-one tutoring - interesting for the similarities to class-wide discussion (a form of dialogue between individuals seeking clarification and the instructor). Finally, Karpicke has shown in a number of studies, e.g. [82], that testing promotes more learning than studying. We are testing students in every class session, both with the quiz and discussion questions.

Programming with a Visual Execution Model. Could we provide students an equivalent experience by teaching a PI-based course using Excel or other computing applications? Our experience suggests that the value of a visual, scaffolded novice programming environment like Alice is that it provides students the most direct form of interaction with the computer possible programming language- level control without the distraction of syntax errors and in a way such that every part of their program's execution is visible to them (we didn't cover the topic of variables). Crucially, the mapping from their program code to an observable execution model is very straightforward. To the extent that other existing or future environments meet these criteria, we believe they would work effectively, too. Key is that students engage with a basic programming interface that manages cognitive load, enabling them to focus on core computational concepts.

Instructor Recommendations. Specifically because the technical content of this course matches that of typical introductory programming courses, it is especially important for the instructor to stay focused on the GE goals of the course. It is challenging to change one's habits from rewarding and assessing success in creating programs to success in analyzing and communicating about programs. How does this challenge play out in class?

While clicker questions in class may ask students to select a line of code to complete a program, or to read a program and select a description of what the code does - the manner in which the instructor must interpret students' clicker votes to the question must reflect the goal of analysis, not correctness. Even if more than 95% of the class gets a question correct, that doesn't mean that students have a thoroughly correct understanding of why the answer is right. Moreover, they must still be given the opportunity to practice discussion of the question, providing their explanations to each other, engaging in interactive questioning and justification, and modeling for each other's methods of thinking about the problem. In class-wide discussions, as many students as possible should be asked to explain in their own words, both why the correct answer is correct, but also how they figured out the other answers were wrong.

Even more challenging for the instructor is to consider completely different kinds of questions than one traditionally asks on introductory programming exams; questions that ask what is the best explanation of why something is (e.g. why do we used a counted (for) loop instead of a while loop) and even questions (on exams) that ask students to not only give an answer, but to explain their analysis that led them to that answer. Testing whether students can merely "write code", with no other explanation or analysis required, seems to be of limited importance.

3.3.4   General Education First: An issue of Equity?

From our experiences of deep reading of students' reports on the impact of the class, we propose that one feature underlies many of our coded categories: the experience of coming to a *new understanding of what a computer is and how one can interact with it.* Overall students seem to grasp that computers are:

1. Deterministic - they do what you tell them to do

2. Precise - they do exactly what you tell them to do, and

3. Comprehensible - the operation of computers can be analyzed and understood.

Is it possible that this visceral understanding (compared to acceptance of telling or quasi-belief) lies at the core of the development of computational thinking skills? Moreover, if one does not yet have this core understanding (as it seems many of our highly-selected college students did not), what is the impact of, for example, a CS Unplugged activity on cryptography, or a course on using Excel effectively for data analysis?

Author Cutts has extensive experience of working with Scottish school teachers and pupils to instill discipline-appreciation through CS Unplugged-style activities. From his experiences, students may overwhelmingly report increased excitement or interest from these experiences, but measurements of learning vary - with a large portion of students seeming having missed even the basic points of the session. This is reflective of learning reports in introductory computing courses. Even in those courses (perhaps CS0) targeted to work with students of any ability, the performance gap for some students seems unassailable. Every instructor has anecdotes of students trying earnestly to master programming, but still failing, if not the course, then failing to develop deep understanding of the core concepts. It is only natural, given repeated experiences, that this may lead instructors to adopt a fixed mindset regarding *some* students' abilities to program. The myth of the programming gene is not so easily dismissed by any experienced instructor.

We posit that lack of understanding that computers are deterministic, precise and comprehensible may be a key factor leading many to struggle, seemingly in vain. Certainly, many students might enter our courses lacking this belief. But some may come to develop it on their own and others may simply be willing to accept yet more incomprehensible magic in the process of programming. We suggest that only some students, with a possibly indefinable set of life experiences, enter our classrooms believing computers can make sense and be reasoned with. Reiterating Turkle's argument [153], as computing has embraced "more intuitive" human interfaces, we have likely actively discouraged attempts to reason

about computer interactions.

Core Competencies Before Appreciation. We propose that the community further study the effect of combinations of general education and excitement or discipline-appreciation courses. Based on our students' claims of the confidence and ability they will have in future engagement with computers and in their increased understanding of where computing concepts exist in their everyday technology use, we propose excitement and discipline appreciation courses will be much more effective when preceded by a GE computing course. As a comparison, multiplication (let alone any advanced mathematical concept) is likely a mystery when taught to students lacking understanding of addition.

It's true, as outreach instructors, we may not have as much fun or personal excitement in teaching a course with the design and goals as outlined here. Not surprisingly, English teachers usually prefer to teach specializations such as poetry or Shakespearean Literature over basic composition. This may be a combination of the fact that students have already moved a bit up the expertise ladder making them easier to communicate and work with. It may be because these courses allow instructors to better share their passion for a deeper and more nuanced engagement with their subject. It may be that students are more likely to be in such courses based on their own choice, rather than as a requirement. But we suggest that instructors consider the deeply rewarding contribution that lies in opening the eyes of all to the skills and attitudes required to live in the computing age.

Where Have You Left Them? Is 7 weeks of Alice and 2 weeks of Excel, with a carefully supporting instructional design, sufficient to define the grounding in the fundamental principles of computation? Perhaps not. This course didn't even cover variables. Yet students seem to feel they have been given the keys to do something useful, something meaningful - with a minimum subset of computational elements. Given more time, one can prioritize more experiences or understandings we want all citizens to have. However, unless *starting* with programming, these efforts will be hamstrung. We look with interest to

see how others adopt and expand this curriculum. Interestingly, by the end of this course, students not only change their views on computing, but they get a significant springboard into traditional introductory programming education. In the short term, this seems a valuable component of any computing course taken by many.

3.4   Conclusions

We encourage the community to consider the needs of a GE curriculum in computing - in contrast to and in conjunction with courses designed to interest students in the field. We provide an example of engaging best-practice pedagogy in teaching a supportive programming language (e.g. Alice) and see that students report gaining long-term skills and confidence as a result of the course, outcomes that we view as core for a GE in computing. Based on our experiences, we hypothesize that GE computing courses should be taken before other computing courses: including application skills courses, excitement courses, or more mainstream programming courses. Moreover, we posit that doing so is a key matter of improving the equity of access to learning in those courses. We encourage the computing education community to engage with GE courses that lift the veil of secrecy and elitism from the field and use of computing.

3.5   Acknowledgments

Chapter 4

Student Experiences with Enculturation in a Student-Centered Peer Instruction Classroom

In Chapter 3 we discovered that Peer Instruction affected the students enculturation throughout the course. In this chapter, we focus on the students' experiences with peer instruction, compared with courses that do not offer peer instruction. Using Chi's ICAP framework (described below) we determine the differences in engagement with the material during lecture, showing that Peer Instruction leads to a more interactive lecture, supporting students in a collaborative, community-based learning experience.

4.1   Support for Using Peer Instruction

There are numerous reports showing Peer Instruction, as one form of a student-centered learning environment, to be successful. This has been measured at the course level through pre-post concept inventory tests [39, 70] and final exam performance [139]. Even more directly, learning from the attempt to solve a "clicker" question, peer discussion, and instructor explanation and feedback has been measured in-class through the use of isomorphic questions in the classroom [118, 140].

Recent results showing a dramatic 61% average reduction in course fail rate [116] across 16 PI instances of 4 different computing courses taught by 7 instructors leads one to

wonder: is there some other effect impacting student persistence? Or is it just the difference in the 3 hours a week spent learning in the PI classroom? Certainly, students in PI courses benefit from having a clear idea of what an instructor wants them to know, and whether or not they know it (assuming clicker questions reflect knowledge and skills measured in course assessments). But, as instructors in similar courses, we have noted something else - a different student attitude towards the course and towards learning.

Existing literature has reported Likert-scale attitudinal surveys showing students both enjoy and value PI classrooms [115, 137]. But why? Is it because they are getting better grades? Because they don't fall asleep?

In this study, we provide preliminary documentation of both student experience and student valuation of their experience in a successful PI class [139]. We asked students to self-report their experience in the PI classroom through a post-term open-ended question asking them to describe what they did (in lecture) in this class and compare it to their experiences in other classes. Their answers indicate a deep, thoughtful, and affective experience with our PI learning environment. We believe the character of these responses will help us better understand what contributes to the overall "success" of PI - spanning both cognitive and socio-psychological effects. Moreover, these results may have bearing on other student-centric techniques (i.e., POGIL, PLTL, PBL).

First we use the Interactive-Constructive-Active-Passive (ICAP) framework proposed by Chi for rigorously delineating student activity in learning environments [22]. This framework differentiates overt, observable classroom activities based on their engagement of students in the learning process. Most valuable for student learning are interactive and constructive activities. Less engaged "active" behaviors include taking notes and listening to lecture, compared to "passively" daydreaming or browsing the web. Coding our data, we find students describe activities in the PI course as constructive and interactive, and behavior in other courses as passive and active. These results help document why we expect improved

student learning: students, not surprisingly, report engaging in activities associated with better learning.

Additionally, in these rich responses, students shared their views on their experience in the classroom - the grand majority of which were positive. Students describe the PI classroom as providing a better setting, and therefore an increased opportunity, for learning. They cite this improvement based on increased feedback and an opportunity to develop deeper understanding through engagement with, and discussion of, challenging questions.

However, the collective student response ranged far beyond the more measurable "learning" experience. Specifically, prominent in many of the student responses are the very socio-psychological issues often noted in the research on why post-secondary students leave STEM fields [152]. These range from affective experiences of enjoyment and comfort in the classroom to extensive and varied experience with an increased sense of community. We perform an open-coding analysis of student responses to better understand the student experience. By taking a "student-centric" approach to understanding the student experience (i.e. asking them directly), we contribute to a more nuanced understanding of effective student-centered classrooms.

4.2   Related Work

Peer Instruction (PI) is the pedagogical classroom practice of (1) asking students to answer a question individually (often using clickers), (2) having them discuss the question in groups, (3) having them answer the question again (often using clickers), (4) having the instructor lead a class-wide discussion based on the student responses, and (5) the instructor dynamically adjusting class content based on student performance. Peer instruction was originally developed in physics [39] and has been shown to cause a two-fold improvement in student performance on concept inventories in multi-institutional studies [39].

PI is relatively new in computer science classrooms, but a recent wave of computer

science education research has shown that students value PI and recommend more instructors use this practice in both large institutions and small colleges [115, 137], that instructors value PI [117], that real learning occurs during student discussions [118], that PI students achieve a 6% higher final exam grade than their standard instruction counterparts on identical final exams [139], that PI reduces failure rates by 61% on average across four computer science courses [116], and that PI (along with Media Computation and Pair Programming) contributes to a 31% increase in retention of majors [119].

The courses we are describing here were also developed under very specific learning theories. In [19] Brown et al. describe situated cognition wherein they argue that "knowledge is situated, being in part a product of the activity, context, and culture in which it is developed and used." They posit that learning, or knowledge, is not in the information itself, but in the way in which learners acquire the knowledge and practice it. Since the goal of the course was to have students understand and be able to discuss computing concepts and problems, the instructor wrote PI questions that would engage the students in addressing the problems they would address in the future in a similar context. In this course students acquired knowledge through actual discussions and interaction with peers.

Furthermore, constructivist learning [15] describes learning to occur only when students are able to build on pre-existing knowledge. Encouraging the students to discuss during the PI course allowed students to evaluate their understanding, compare it to the understanding of their peers, and discuss why an understanding was incorrect. This allowed students to self-identify when their model was incorrect and construct a new model. In [40] we have seen that demos often hides the students' misunderstandings from them, and therefore students attempt to force what they are watching or hearing into their incorrect model. We can see that Chi's ICAP framework clearly describes constructivist learning to occur in **constructive** and **interactive** activities whereas in **passive** or **active** activities students are not modifying their existing model [22].

The work most closely related to this investigation is the work by Gaffney et al. [63] on how active learning pedagogies violate students' initial expectations of the classroom. Investigating multiple classes using the Student Centered Active Learning Environment for Undergraduate Programs (SCALE-UP), Gaffney et al. found that the shift in student expectations was generally positive. They also provide advice to instructors on how best to orient students to the changed atmosphere. Their work is complementary to this work, as this work focuses how students feel their role in the classroom changes and their work aims to help us better prepare them for the new classroom environment.

4.3   Methodology

The CS0 in this study was taught at a large, research-intensive, public university in Winter 2012 using a PI approach (N=84). Of specific note, as this was a non-majors computing course, the instructor explicitly communicated to students that the goal of the course was not simply to teach them to program, but to help them understand and develop skills in analyzing computer programs in detail and to understand the core concepts underlying all computer program execution (repeated and conditional execution, abstraction, etc.). A core mantra of the course was to learn to think like computer scientists think, so as to be better prepared to go on to learn any new computer program or application needed in one's future career in any discipline. We believe this explicit focus on "learning to learn" is important to this study and should have impacted students' views on their role in the class.

4.3.1   Course Characteristics

Previous reporting on this course documents it as a "successful" PI course in that students experienced better learning outcomes than in a comparable standard instructional format (scoring higher on an identical final exam) [139]. In accordance with standard PI methodologies, students were assigned pre-class preparation (reading the textbook and being

asked to follow along creating the Alice programs described in the book [56]). This work was incentivized by a short quiz at the beginning of each class. Lecture was comprised, in the main, by a series of "clicker discussion questions" where a challenging issue from the day's material was asked and answered via the following process:

1. Students answered individually with a clicker

2. Students discussed the question in assigned groups of three. The instructor and tutors would walk around listening in and could interact with students.

3. Students voted again. To incentivize groups to engage in step 2, groups were told to come to a consensus and all vote the same way.

4. A class-wide discussion was run, usually started by the instructor asking several groups to share their thinking and discussion. The instructor would wrap up the discussion, varying her explanation depth and detail based on students' success in answering and explaining the question.

Questions were interspersed with mini-lecture and Alice live coding demonstrations to back-up and/or further extend the concepts in the clicker questions.

4.3.2   Student Survey

Students were surveyed on their experiences in the course as part of a "take home final" at the end of the course. Here we analyze student response to the following open-ended question:

Compare and contrast your role as a student in *this* course's LECTURE with other "standard" course lectures. That is, what are you here to do in this lecture? In other courses' lectures? Answer with an essay of 150 words or longer. Any thoughtful answer (positive or negative) is desired.

To counter concerns that the reported experiences may be purely instructor-driven, we analyze student responses in both this PI section and a standard instruction (SI) class (N=124), led by the same instructor, at the same institution, in the same term, and with the same learning goals. The standard instruction version of the class is described in depth in [139], but included slide-based lecture and live coding demos where the instructor posed questions to both engage students and help them attend to important issues. Notably, the instructor received identical, very high student satisfaction evaluations in both courses.

4.3.3 Student Classroom Activities

In 2008 Chi recognized that research in cognitive and learning science inconsistently and unclearly used common terms such as passive, active, constructive and interactive [22]. She proposed a framework specifically describing these phrases in terms of observable, overt activities. Her ICAP framework provides evidence from literature to support a hypothesis that increased learning comes at the highest level of interaction. Each student response to the "role" question in 4.3.2 was coded passive, active, constructive or interactive according to overt activities they describe. Since the question asks students to compare their role in this class against a "standard" course, we coded both their response to the computer science course as well as the "standard" course (if they describe behaviors in the "standard" course). Our coding scheme adhered to the following descriptions of each level of student activity:

- **Passive:** Not engaged in lecture by doing something else (daydreaming, facebook, zoning out, etc.) or not attending.

- **Active:** Listening to lecture and/or taking notes.

- **Constructive:** Solving problems on their own. Reflecting on their own understanding. Justifying, explaining, and asking questions of the instructor.

- **Interactive:** Argue/explain concepts with other students. Learn from or teach a

partner.

An important note for the reader who is familiar with the term "Active Learning" is that Chi describes **active** activities to be what some might call passive (e.g. taking notes, or listening to lecture). We believe that "Active Learning" would fall into either **constructive** or **interactive** in Chi's Framework.

Student submissions, predominately for the SI course, would often vaguely refer to improved learning, more engagement, or more interaction as a result of the in-class demos. Though students were describing these experiences, we only coded on *actual activities* they described. For example, if a student said they "learned a lot from watching demos and the class was interactive," this would be categorized as **active** because they did not describe any activity aside from watching demos.

Quotes were categorized using the highest Chi category, for example, if a student described attentively listening and then also described interacting with their peers, it would be marked as **interactive**. We categorized the data from both the standard lecture and the peer instruction lecture combined together with no indication of course, and then separated the data. We had two authors categorize 10% of the data at a time until they reached an 85% inter-rater reliability. Then the rest of the data was split evenly between the two authors and categorized.

4.3.4 Student Reflection and Valuation

To further describe the variation of student experience in the PI lecture, we performed a grounded theory analysis. Due to both the exploratory nature of our work, and the limitations of our dataset, we performed only the first level of analysis in the grounded theory methodology - open coding. Open coding involves "fracturing data and making connections" through a process of noting and labeling categories [35]. Also known as the constant comparative method, the goal of analysis using grounded theory is not to quantify

or reduce a data set to what is frequently experienced, but instead to seek to provide a rich description of the scope of variation of experience in a situation - so as to inform future theory development that should be applicable in the setting observed.

Open coding was performed by two authors, each reviewing one-half of the PI section responses and identifying phrases which augmented understanding of student experience beyond strict activity definition. Each author developed an emergent set of category names, which provided an organizational structure to the phrases. After discussing both categorizations and after one author coded the half originally coded by the other author, a structure of categories emerged that communicated the variation observed in the data.

4.4   Threats to Validity

There are threats to the validity of this study, as is common for analysis of qualitative data. Of greatest note, and as mentioned in the methodology, is that students' responses and depths of thought about their role as a learner in this course may have been affected by the explicit goal of the course to prepare students to be confident and more successful in future computer use. Compared to, for example, a foreign language course where the primary learning outcome would be to develop student ability to understand and speak in a foreign language, our course specifically did not emphasize a learning outcome of being able to write code in Alice. Instead, we explicitly and frequently reinforced with students that we were only using Alice to help them understand the basic "language" which underlies all computer applications; and that our goal was that they would be able to use this understanding in whatever technology and computer use they might face in their careers in the future. This greater focus on "learning to learn in future situations" may have impacted students' metacognitive view of their goal as a learner in lecture.

Secondly, three authors were heavily involved in the development of the course, though only one served in an instructional capacity in these specific course offerings. All

Table 4.1. Chi categories for CS0 PI course versus another course

	Passive	Active	Constructive	Interactive
CS0 Peer Instruction	0%	2.4%	12.2%	85.4%
Other college courses	16.4%	79.1%	4.5%	0%

authors have been involved in the offering of PI-based courses and bring to this study their perception of the changed experience and engagement of students, as they perceive it during lectures. However, the purpose of the post-hoc analysis of students' own responses in this study serves as a different and hopefully more individual and accurate documentation of students' experiences in the classroom than merely a report from the instructor's point of view. Still, a post-hoc, short answer question cannot possibly provide sufficient evidence for the development of a theory of student experience in the PI classroom. This study means to serve as initial data to help inform effective design of a more thorough ethnographic study.

4.5   Results

We analyze our data from two perspectives: characterizing type of activity and describing student perception of that activity.

4.5.1   Student Activity Characterization

In Table 1 we find that the activities during the CS0 PI lecture are very different from other courses the students described. The majority (85%) of the students said they did some kind of interactive activity: "*I felt I had more of a role as a "student-teacher" than a student. I was extremely happy about this because it helped me reexamine the material in a different way while explaining it to another student.*" No student described any **interactive** activity in other courses they took at the university, but instead described activities as **passive** "*In other lectures it is easy to get distracted and not really be paying attention to what the professor*

is teaching" or **active** "*In other course's, I feel like a lump on a log that soaks up whatever information I can squeeze onto my notes/slides that the professor has us print off.*"

Students sometimes did not describe the **interactive** activities that might usually occur in a PI course (peer discussions), but still described **constructive** activities where they were constructing their understanding through effortful activities. For example, "*In this class, you must stay active in order to answer the questions, so it keeps you focused on the task at hand.*" Though the clicker questions were coupled with group discussions, this student only described the activity of answering the questions throughout lecture, but did not describe the interactions with their peers.

There were still some students who described only an **active** role in the PI course. Though students might have been implying a **constructive** or **interactive** role, we only coded those with specific evidence. For example, this student only describes an **active** role in the course "*The course is really hands on and visual, which helps me learn.*"

A possible concern is that it is merely a characteristic (e.g., the personality) of the instructor that causes students to describe the engagement that they do - regardless of instructional method. To assess this possibility, we coded student responses to the role question given in a standard instruction (SI) version of the course taught by the same professor in the same quarter (reported in detail in [139]). In the SI course, only 4% of the students described **interactive** activities and 24% described **constructive**. 67% of the students described only **active** activities, which indicates that the difference in instructional method is what drove the students to be **interactive** rather than **passive**. However, we urge the reader to interpret these results with care: these students had not experienced peer instruction and their responses are not comparable to those of PI students. Indeed, these SI students respond to the question by comparing a "computer science" course (with live coding) to non-computing courses. But the results indicate student activity in the course is not merely a result of a characteristic of the instructor.

4.5.2 Analyzing Student Reflection/Valuation

Affect

The course had a strong influence on student affect, both on how comfortable they felt as learners because of a perceived permission to be unsure and to ask questions, and on how much they enjoyed the course. ***Comfort.*** One student wrote that the course design "*made for a very comfortable learning environment. There was no stress or embarrassment; one could speak their mind freely.*" A student who had previously failed classes and was on academic probation wrote "*I recognize my responsibility to do pre-reading, but sometimes I struggle with certain topics. The professor made sure to cover every detail and as a student I felt way more confident in this class than in other classes.*" This student earned an A grade in the class. Casting this in the light of Dweck's work on self efficacy, the learning environment clearly promotes a growth mindset, where it is accepted that learning involves struggle and asking for help and feedback is necessary at times [50]. More importantly, it was clear to the students that the instructor knew this: "*It made me comfortable to know I could ask a question and others could too, and receive feedback (that was actually wanted by the instructor).*"

Enjoy. Students noted their enjoyment of the class, with a sense of surprise: *"for once this class was actually quite fun"*, *"the transition of my attitude [towards enjoying in-class discussion] is unexpected"*, and *"it made me realize how boring other classes can be when the prof is just rambling on about something half the class might not understand"*. One student alluded to how the negativity associated with getting an answer wrong was turned around through discussion with peers: *"we get to redeem ourselves by talking amongst one another to further our knowledge. I enjoy the discussion part of class because it allows my classmates and I to participate in learning too."*

Social/Community

Many students wrote of the shift from individual learner to being a member of a community of learners.

Responsibility to team. The requirement to work in groups increased the importance of preparing for class: *"having done the homework actually had an effect on what I did in class and could possibly affect my partners' group discussions."* They were no longer doing it only for themselves but also to ensure that their peers fared well. Note that students refer to this responsibility in terms of their peers' learning and not to their grade, when in fact credit for working effectively together was based only on the members of each discussion group answering the same way. Furthermore, the learning of the whole class emerges as an important feature. For example, *"It wasn't only about your learning but also everyone else in the class learning as well", "in this class, everyone is encouraged to learn and everyone must learn so that no one is left behind"*, and *"I had a wider array of responsibilities in this class because I had to interact and be involved in the group and class-wide discussion ... almost the entire time"*. Students appear almost proud of their contributory role in class: *"I was not only a receiver but a giver in the class", "encouraged to help others to learn while also furthering our understanding", "we [discussion group] made sure that all of us understood what was going on, taught each other along the way."*

Connection to professor. While not explicitly responsible, students also indicated a greater connection to the professor - even in the large lecture hall. *"I feel more connected with the professor who takes her time to hear our opinions."*

Participation. Students report having permission to speak out in the class, and thereby being an active participant in the lecture: *"Every student has the opportunity to participate", "I was involved in the lecture, not just a sponge trying to absorb everything the instructor is saying"* and *"I can express my opinion to all"*. There is no sense of coercion in these statements, no sense that the students had to speak out, further contributing to the

inclusive quality of the learning environment. Some found participating hard initially, but worked it: *"I may still be reluctant to speak in front of the whole class, but now I feel really comfortable to express my thoughts and opinions to my partners."*

Social and academic inclusion. Other lectures are reported to be isolating experiences: *"It is easy to feel lost in a lecture hall of hundreds of students"*, *"in other classes I often feel isolated ... since there isn't a value placed on discussion with other students to cement the topics"*. The PI class is significantly different: *"I now enjoy it because I know my partners pretty well and have made new friends from the class"*, *"This has been my favorite class for the simple fact that it was an open environment"*. The isolation of traditional lectures spills over into the students' perception of learning: *"In other classes, I have to learn everything by myself"*. This isolation transfers to study outside lectures as well:*"In other classes, I have to do homework and study alone and hope that my explanation is correct"*. This student may be indicating that in the PI class, he/she works with other students outside lecture, or that homework/study outside lecture is still undertaken on their own, but that the in-lecture experience provides feedback on their learning.

Negative: group consensus. A student admitted to being initially skeptical about the format, but came to the realization that he learned by discussing with others. Nonetheless, he continued to be bothered by the incentives to work in his group saying, *"I did not like how we needed to have a group vote for discussions."*

Learning

The main bulk of student comments that we identified relate, not unsurprisingly, to the students' learning in the class.

Understanding. Students note in general terms that their level of understanding is deeper: *"[the course helps to] solidify the topics so we remember and understand rather than memorize"*, *"discussions helped me to understand ... 'why' I chose an answer"*, *"I can see all sides and get a thorough understanding of what is right or not"*, *"better than*

... (reading textbook, just listening, taking notes (frantically)) I come to lecture to solve problems and not just have information thrown at me".

Students explicitly refer to critical thinking: *"[being required to] provide input in lectures is what makes us think critically", "in other classes the teacher just lectures instead of promoting critical thinking".* Students note that they are developing reasoning skills: *"one may be encouraged to actually use reasoning, if one does not know the answer"* and *"by discussing with my classmates, I learn how to defend my choice by backing it up with evidence or by examples."*

Crucial aspects of deeper learning are identified. Examples include: the benefit of working through wrong answers - *"Discussions helped me to understand why I chose the wrong answer along with why the right answer is the correct one"* and *"the professor was really good at picking [distractor] answers because it gave us a chance to think the answers through critically"*; multiple ways of thinking about a problem - *"I had a role as a student-teacher. [This] helped me re-examine the material in a different way while explaining to another student", "having others participate and share ... their thoughts ... definitely helped in opening my eyes to different ways of thinking"* and *"conversations ... with my classmates help me to see their point of view and how they think about questions"*; and finally, the opportunity to consolidate learning - *"repeating concepts really helped me understand"* and *"my skills that I had learned in lecture were constantly tested in every lecture and every lab".*

Feedback. Students wrote both about the feedback they received from, and also about the feedback they provided to, the instructor, underlining the connected nature of the class.

Votes and discussion in class were *"encouraged and desired"* by the instructor, enabling her to *"adjust the class discussion, which was very helpful in building knowledge when my group decided it wanted to talk to make it [the class discussion] focus more on*

areas people don't truly understand." This adjustment to the students' needs positively affects the pace of classes: *"This course moved at a reasonable pace, allowing me to take time to make sure I understood each concept, and asking the TA for help when I encountered any confusion."* Other classes are noted as *"impersonal, move too fast, lose my attention"*.

Feedback to the students was clearly important also: *"it gave me instant feedback and allowed me to know if I made a mistake ... not something I can normally do in other lectures"*, *"knowing right away whether or not my answer was correct was also really helpful to my learning because I could understand what areas I was lacking in right away"*, and *"in this forum-like, communal class, I had the assurance that my understanding ... was correct and allowed me to further my knowledge more effectively"*. A comparison with knowledge-filled lectures was made: *"in other courses, I usually do not participate because the class is big and everybody just receives knowledge instead of feedback"*. The course structure amplified what the student needed to know or be able to do: *"makes it clear what we have to know and what we need to focus on"*.

Efficiency. Students found the time spent in class to be more beneficial than in other classes: *"I learn the information a lot easier, more efficiently"*, *"I have never had a class like this in which I felt by the end of class that I truly had a grasp of the material"*, compared to *"in most other classes I walk out feeling as if I had learned nothing and have a lot more studying to do on my own"*. This represents a shift in effort from outside lecture to inside, with a consequent overall reduction in time spent on the course: *"as a student in lecture, the course demands much more - but overall, this course demands much less, seeing as demanding participation in lecture facilitates and requires learning"*, also expressed, at its simplest, as: *"better learning with less out of class study time"*. That said, some students wanted greater efficiency saying *"Sometimes it would get repetitive on very simple topics which would cause my group to stop working."*

textbfAuthenticity. A few comments referred to a perceived lack of authenticity in

the learning environment. *"Still not doing what you are learning... it would have been more helpful if we had done most of the learning in the labs with the tutors showing us how to do it instead of trying to make us figure it out from the reading"* indicates that the instructor had failed to make it clear to this student that a primary aim of the course was to get students doing exactly this kind of "figuring out". *"Clickers create a fake learning environment where most of the students are just sitting there clicking buttons and not really caring about the learning but just about getting points"* does not seem to be supported by the responses from other students, though was felt by at least one.

Flipped Structure

The format of the course, with preparatory work assessed using a quiz at the start of each lecture followed by regular questions and discussion, influenced students' preparation for and attendance at class, as well as the level of their focus on the material covered.

Preparation. Students reported that the PI format requiring them to be prepared for class was a benefit, both as a primer to the class itself: *"mandatory pre-reading really useful because you get the general idea what you gonna learn in class"* and to enable participation in the group discussion: *"practically required that you do the homework and pay attention just so you can contribute in class"*. Quizzes were a driving factor in completing the preparatory work, although this represented a challenge to the workload balance between courses: *"the quizzes ... made it critical that you had done the reading and the exploratory homework and that also helped learn better. The only difficult part was keeping up because of other class engagements and assignments ... The only thing is it is easier in other lectures to slack off and critical you keep up in this class."* It is interesting to speculate how students would fare taking a number of PI courses simultaneously.

Attendance. The course structure affected students' attitudes towards attending lectures. Comments ranged from a coercive influence: *"Clickers made me attend (and pay attention)"*, to a realization of the learning benefits: *"It was exponentially more beneficial*

to actually attend class". Clear changes of practice were noted: *"I never ever usually go to class, ever. At first I was sad that I had to go to class. I usually read the book and follow along on my own rather than attending class. But after taking this course I realize the importance of actually attending class. It made a huge difference in how I learned and remembered the information"*. Not all comments were positive on the perceived requirement to attend: *"People should be able to choose whether or not they're gonna go to lecture without being penalized"*.

Attention. Some students noted that they had to pay attention more in the PI class; others stated that it was easier to do so. *"it forced me to engage with the material while I was in class"* and *"it was important to pay attention and be engaged because if you did not you would miss the point of discussion and clicker questions"* exemplify the need to pay attention. On the other hand, *"in this course it was hard not to pay attention"*, *"[clicker questions] really helped me keep focus and take a stronger interest in what was being presented in lecture"*, *"interacting with others allows our brain to function again and draws our attention back to the lecture"* and *"helped me focus and learn the material in class, instead of losing focus thinking about other things I have going on in my life right now"* exemplify how the course design aids students to concentrate on the material.

Transfer of Behavior to Other Courses

Quite unexpected were reports of how experiences in this course were impacting behavior in other courses. Realizing the value of repeated review one student noted *"clicker questions constantly forced me to review the material over and over.... if I reviewed a clicker question I remembered it... I got a perfect score on the midterm. I realized that I should be doing [this] for other classes."* Another changed his/her approach to lectures, *"I am currently a senior and in my 4 years of college I have probably attended 30% of my lectures. They never hold my attention, I sit on my laptop all class. I am proud to say that I started attending all my lectures and actually tried to pay attention."*

Research-Ralted or Evidence-Based Practices

Whether or not we, as instructors, find these reported experiences compelling, a number of student reflections very directly connect with findings or recommendations in educational research.

For example, in work on reasons students leave STEM in university, Tobias [152] reports that in STEM classrooms students feel like their contribution is not valued - that they should merely take in the knowledge of the professors. Our students felt differently: *"I felt like my feedback was encouraged and desired"* and *"this has been my favorite class here ... for the simple fact that it was an open environment."* In Talking about Leaving [133], a study of STEM leavers, women who leave STEM are reported to have higher grades than men who stay - perhaps indicating that women incorrectly understand their relative abilities compared to their peers. Our students told us the following: *"the best part [of clickers] was seeing the results of the whole class. This kind of gave me an idea as to where the whole class was and if we were on the same page or not."* Additionally, metrics like the US's NSSE (National Survey of Student Engagement) are highly scrutinized because of research showing improved outcomes for more highly engaged undergraduates. One area of engagement recognized by NSSE is the positive effect of engaging with the professor and also having a professor express high expectations for student learning. Our students experienced both: *"because of the frequent clicker questions I feel like am forced to be on top of my game as these points are vital for my grade"* and *"students get opportunities to have discussion with the professor during the lecture."*

In the US National Research Council's report *How People Learn*, educators are implored to employ learning environments that are grounded in the research of how people learn [15]. In our students' responses we see reference indicating experiences with the following:

- **Constructivist learning:** *"Because I was asked to apply myself often, I think it helped*

form stronger connections to the material in lecture."

- **Situated cognition:** *"Here in this class I was a student of computer programming while in others they [profs] are there to present us information and we have to spit it back on pointless exams."*

- **Metacognition:** *"This lecture focused on the learning aspect of programming rather than the programming itself. In other classes the focus was on the material rather that the learning of the material. I think this is inappropriate for certain subjects where the material is challenging."*

Students also reported impacts on their studying. Numerous studies have demonstrated the importance of spaced learning (versus massed learning, i.e., cramming) for long-term retention, e.g. [10] and the value of repeated testing compared to repeated studying for delayed recall [124]. One student specifically recognized this benefit: *"I actually studied less for the midterm, because I has already mastered a lot of the materials required before I started reviewing contrasting to many other courses where I had to cram because I did not really understand the materials taught in class."*

One challenge with instructor lecture is that an expert experiences lecture differently, by virtue of their expertise [73]. Students concurred: *"Hearing a different type of explanation from a fellow student who uses terms I may understand better, makes learning new concept easier"* and *"It was nice to hear other class members' opinions about a certain solution because their way because their way of breaking down an answer was at times easier to understand than the tutor's or instructor's"*.

Finally, in our goal to develop critical thinkers rather than merely repeaters of information, this quote related to the scientific process is pleasing *"In this lecture, I felt it was OK to be wrong and to work through what caused us to think that way."*

4.6   Discussion

We believe the Chi framework for differentiating student activity as passive, active, constructive, and interactive is valuable in examining the fidelity of faculty's adoption of the PI process in classrooms. As Turpen and Finkelstein report, many faculty who describe themselves as implementing PI may in fact evidence a significant lack of fidelity of implementation to PI [154]. For example, faculty who believe they are implementing PI but are perhaps only encouraging their students in an **active** or **constructive** role, not an **interactive** role. This is also studied in 11 research-based instructional strategies in engineering [13]. We believe the Chi framework could be valuable in training faculty in use of student-centric practices - giving them a clearer distinction in the kinds of activity to foster in the classroom.

Along similar lines, the extensive range of student valuation comments was pleasantly surprising to us. In spite of our years of engagement in our own PI classrooms, and given our intense scrutiny of them through our other research in the PI process and student performance and retention, we learned far more than we expected from students' descriptions of their role. While PI has a notable core of research regarding its impact on learning gains, the categories of student report make it clear to us how unique the PI classroom experience is in contrast to students' daily lecture routine. Most notable to us is how much more motivated students must be in this class - whether that motivation comes from having to be prepared each lecture, to looking forward to working with their team and feeling "part of something", to actually learning something, to the pleasure at getting effective feedback on their learning, to simple enjoyment of the experience. Though this data doesn't provide an answer to our initial query of whether "other effects" (beyond increased understanding and feedback) contribute to the successes of PI classes, it helps provide traction to our feelings that our PI classes seem much more motivated.

4.6.1 Instructor Experience

A few things not mentioned by students stand out from our instructor experience that we would like to report in support of future research on the student experience in the PI classroom. From going around and listening in on student conversations in the group discussion phase we see the need to explore both the process and the value of having students put together arguments in more "novice" terms. In particular, we can see that a scaffold for supporting students in experimenting with the use of new vocabulary to deepen their understanding might be useful. Additionally, further support and instruction in helping students have productive discussions, perhaps by leveraging transactive discourse methods, should be explored [148].

Additionally, we have wondered about the experience of discussions for students who are not native speakers of the language of instruction. Although in our case, we've noticed some likely benefits (e.g., conversations in Spanish) and can imagine scenarios where students can use this time to clarify language issues, one could also imagine these students feeling disadvantaged or isolated in discussions.

Finally, in the PI section a number of undergraduate tutors were used as additional support in circulating during group discussions. This was partly to support training of tutors for a future course and partly from experience in much larger instances of this course. However, we would be interested in seeing in more detail how additional, near-peer tutors impact the student learning attitude and experience.

4.6.2 Negative Experience

Of particular interest for future ethnographic studies are any negative comments made by students, including those where a student says they were first unhappy about something, but later came to see its value. This was the third time the instructor had taught this course and she had made conscious and significant effort to communicate to students

the rationale behind the classroom method including discussion of specific components and their value (especially the discussion). It would be helpful to explore in greater depth (and over the course of the term, not just at the end) the viewpoints of students about the specific issues that were, even at one point, viewed negatively. Greater depth of information and situation of those concerns would be very helpful in developing resources and activities for faculty using PI to be able to use to help their students "come around" to seeing past their (possible) initial negative views. We believe a theory of student experience in PI classrooms would be helpful to faculty wanting to best support their students in learning in this new environment.

4.6.3   Import for STEM Retention/Education

After repeated readings and analysis, we were quite struck by how our students have become so resigned to the standard university lecture model. They feel isolated: from each other, as having no "part" in class, and as being on their own to learn the material. Worst, they are clearly failing to see the subject matter as something with which they can, and should, engage: they are sponges, they listen to professors "ramble", and they try to copy as much into their notes as possible. We highly recommend the chapter "Introductory Physics: The Eric Experiment" in [152] and to compare the findings there with what our students report in their "other classes". In 20 years, it appears we have failed to make much progress. On the positive side, our students do want to be engaged (mostly). They are willing to prepare for class on their own, they are willing to work hard with their peers, and they are willing to engage as a scientist: to get things wrong and to work to puzzle things out. They recognize the unusually proscriptive structure of PI, but they also (again, mostly) recognize how they benefit by having an excuse to not space out, to come to class, and to stay caught up.

But perhaps the most exciting is students' recognition and value of themselves as a community of learners - along with the professor. Both in Tobias' work and in *Talking*

About Leaving, the negative impact of lack of supportive community is clear. Why should we care about community? There are many arguments, but as a community of scholars, we understand the value in and the motivation of working with colleagues and having your work critically analyzed for the purpose of improvement. As educators in the 21st century, perhaps looking to define our value beyond that of a dispenser of information via video, our ability to engage students in the process of learning in community is exciting. Certainly, it makes teaching class much more fun and increases our motivation to teach. And, perhaps, if students are having more fun, and feel more ability to learn, maybe they are spending more time trying to learn – maybe they are more motivated to persist. Whether and how this motivation may affect out-of-class learning and activity is something we believe should be explored.

4.7 Conclusions

This work evaluates students' perceptions of their "role" in a PI computing course. Analysis of students' self-reported activities demonstrated that the majority of students report "interactive" activities (arguing, discussing, explaining to peers) in PI classes whereas the majority report "active" activities (listening, taking notes) in their standard courses. This substantial shift in classroom activities unsurprisingly causes students to also change their perception of their role in a classroom. A grounded theory open-coding investigation of student perceptions finds students positively affected by the shift in terms of class enjoyment, improved attendance and better attentiveness. They also report on improved meta-cognition skills facilitated by more frequent feedback and class discussions. Lastly, a few even report on trying to apply their self-reported improved learning behaviors from PI classes to other classes. This work has potential implications in identifying common desirable characteristics in student-centered learning environments and speaks to the issues raised in the STEM retention literature.

4.8 Acknowledgments

Chapter 4, in full, is a reprint of the material as it appears in ICER 2013. Simon, Beth; Esper, Sarah; Porter, Leo; Cutts, Quintin, ACM, 2013. The dissertation author was a primary investigator and author of this paper. Thank you to the anonymous reviewers for their contributions to this work. This work was supported by NSF grant 1140731.

Chapter 5

Evaluating and Classifying Formative Peer Instruction Questions and Summative Exam Questions through the Lens of Situated Cognition

In Chapter 4 we showed that engaging students in Peer Instruction during lecture resulted in a more interactive experience, and added enculturation to the learning experience. Though we saw positive effect of the pedagogy, the first iteration of the summative and formative questions posed to the students had not been evaluated. In this chapter we focus on the formative and summative questions we asked the students and show that the questions themselves require the students to authentically engage with the material. Learning materials in this course should encourage the students to continue spiraling towards expertise, engaging them in authentic analysis of computing concepts creates an opportunity for them to legitimately participate.

5.1   Using Situated Cognition to Evaluate Formative and Summative Multiple-Choice Questions

Situated cognition is a learning theory in which learning is seen not from the isolated cognitive, conceptual, or abstract perspective of the individual learner, but rather as situated

within "the activity, context, and culture in which it [learning] is developed and used." [19] This viewpoint would seem to resonate with the computing education community. We are a discipline with a strong professional focus, grounded in the development of a tool (i.e., the computer) for use in solving problems for the benefit of society. As such, it is *implicit* in our curriculum that we seek to apprentice students in becoming (more) masterful computing professionals. Could instruction be improved by developing more *explicit* learning goals generated through the lens of situated cognition?

In this paper we consider the learning materials used in a pilot of the CS Principles course, under development in the United States. It provides an interesting case because of two factors:

1. The course sought to provide general education (not preprofessional) computational thinking skills. As developers of the course, we were vigilant in considering our target audience. Our mantra "if this is the last computing course a student ever takes, what do I want them to know" resulted in an emphasis on acculturating students into the ways computing professionals see, understand, and solve problems - because it is these ways of thinking that will serve them as they engage with computation throughout their lives. The ability to write programs was important primarily in service to the acculturation goal, not as a goal in its own right.

2. The course was highly successful in how it impacted students' perceptions of future computing use. In [44] we report on the ways students said they could solve problems better, transfer what they had learned to new situations, and more generally, how their confidence had increased and that they could now see technology in a new way.

Because of these factors, we have performed an analysis of the course learning materials to provide an explication of what it was that students were engaged in doing that might have contributed to these results. Using situated cognition theory in this analysis is appropriate because the course design centered on Peer Instruction - a pedagogy that

supports cognitive apprenticeship in the classroom. The 133 clicker questions we analyzed were developed by the instructor through daily consideration of the programming concepts and constructs presented in the text-book in light of the following questions: "How can I get students to see this concept/construct the way I see it? To understand why I use it the way I do to analyze, solve, or debug problems?" Because clicker questions are presented as multiple-choice questions (MCQs), this required focusing on one small aspect of computing thinking at a time.

The *Abstraction Transition Taxonomy* resulting from our analysis forms a description framed by situated cognition: the amalgamation of activities, tools, and culture that the instructor felt would acculturate the students into the world of computing. We express our analysis in terms of a taxonomy because of the nature of the analyzed materials - a number of taxonomies exist to classify assessment questions. Taxonomies are useful because they afford communities a shared vocabulary for discussing learning. We propose that the categories of clicker questions define both a more explicit and more extensive set of learning goals, a set that we should adopt for introductory programming classes. Nonetheless, we find, through analysis of summative assessment items from 7 different introductory programming courses, that a much-restricted set of these learning goals are measured by final exams - including our own.

5.2   Motivation

5.2.1   Expertise Development

The goal of any introductory-sequence programming course is not simply to create students who can generate working programs. Rather, more holistically, we want students to begin to see problems as programmers do and learn to apply programming concepts and constructs in the "right way" to solve computational problems. Perhaps because we are aware of our responsibility to produce computing *professionals*, compared to disciplines

such as biology or math, our curriculum embraces the intertwining of theory and application and we start students immediately down the path of 10,000 hours of deliberate practice required to develop expertise [53]. That is, we start them solving problems by writing computer programs as they learn the vocabulary/language and concepts underpinning the discipline. We posit that our typical educational practices could be better informed by the literature on expertise development.

Let us consider a long-utilized method for developing expertise - the master- apprentice model. Consider an analogy: the apprenticeship of tailors.[1] Apprentice tailors work closely with their masters in two important ways we want to contrast to common models of programmer education:

1. *Scaffolding.* Tailors are scaffolded in the development and practice of their skills by the master - starting with tasks like ironing which engage them in legitimate peripheral participation (the opportunity to observe and acculturate while making a contribution) [27]. They are also set smaller, simpler tasks - they are not assigned to make an entire suit, they might be given various piecework a bit at a time; the button holes, cloth selection and preparation, fitting, etc.

2. *Process.* Tailors are not judged merely on the final outcome or garment. The master works in close proximity to the apprentice with the responsibility to observe critical professional skills: processes, analysis and decision-making, and intermediate results.

We propose that there is a lesson for us to learn from the master tailor. Computing instructors focus far too much on the final product - the small program/the simple garment. We do not provide enough direction in the preparatory steps - moreover, while we may "teach" students about various steps, we don't provide enough guidance on the specific masterful ways we approach the steps, the ways in which we evaluate our activities in the steps, and the considerations we take in deciding upon steps. We require our students to

---

[1]An example inspired by the work of Jean Lave, but expanded and interpreted by the authors.

experience these for themselves through their experiences doing programming problems. We really only look at their finished products - we don't go through the process with them and observe whether they are thinking and deciding as we would like - not just "doing" as we do.

5.2.2 Defining Learning Outcomes through Situated Cognition Theory

Situated cognition theory embraces the notion of learning and knowing as inseparable from the way in which that knowledge will be applied in real life. This stands in contrast to the individualistic view of learning, where learners can be taught conceptual knowledge abstracted from the situation in which it is learned and used [19]. Instead, situated cognition theory states that *knowledge is situated in a triad of activity, tools, and culture:* that it's not just what steps to take using what resources; a critical aspect comes from the context and culture in which that knowledge is to be exercised.

In the language of situated cognition theory, in computing education we engage students in *activities* using computing *tools*. What is missing is the *cultural* portion - the fact that our students are not doing real piecework that will contribute to a real suit for a real customer and, most importantly, under the auspices and detailed guidance of a master. To support development of expertise we need to consider all three components - activities, tools, and culture.

Table 5.1 proposes some programming-specific interpretations of activity, tools, and culture. Culture is critically interdependent in this creation; it determines how and why practitioners choose the tools they do, what activities they engage in and how they know to make those choices. As stated in [19], "[t]he activities of many communities are unfathomable, unless they are viewed from within the culture." Perhaps this sheds light on bizarre programming behaviors, completely unexpected questions, and often intense frustration expressed by so many of our students.

Table 5.1. The core components of situated cognition, using programming as an example.

Activity	Running a program, observing program behavior or output, inspecting code, making edits, compiling, hypothesizing
Tools	Programming constructs, programming concepts, IDEs
Culture	Knowing to match up runtime behavior with static codebase, knowing what variables to trace, knowing the right way to decompose a problem, knowing to look at a problem in light of what constructs are available to solve it.

To shed light on the difference between activity and culture as defined in Table 5.1, consider the difference between novice and expert debugging. Instructors joke about stories of novices using the following activities: run program, observe output, note that it's not what was expected, make an edit at random, recompile, run again, and so on. This is a novice culture. The expert would: run the program, observe the output and note that it wasn't what was expected, examine the code and the problem statement or maybe add diagnostic print statements in order to develop a hypothesis as to the problem, run the program with particular test data to check the hypothesis, and so on. The individual activities are simple and should be relatively easy to enact. It is the choice of activities and the way the activities are combined that defines the difference between novice and expert, and hence the cultural aspect.

One of the primary manners of supporting learning within the lens of situated cognition is through the development and support of communities of practice [158]. A community of practice is made up of experts or practitioners who share a profession. Member development (from novice to more expert) is embedded within all of the shared information and activities of the community. Members' expertise is developed through interactions with others in the community, while working on real problems of community interest, expressed using the community's own vocabulary..

Communities of practice have inherent conflicts with traditional modes of academic

instruction. Most critically, the members of the community (the students) are novices (or relative novices in the course in question). The expert is the lone instructor (who may or may not be involved in a professional community of practice in the subject area). Additionally, the customs of individual assessment often lie in conflict with authentic practice processes and problems.

Cognitive apprenticeship is a method developed by [27] seeking to "try to enculturate students into authentic practices through activity and social interaction in a way similar to that evident - and evidently successful - in craft apprenticeship" [19]. In this course, we employed Peer Instruction to support cognitive apprenticeship [105]. We deliberately and consciously developed clicker questions *not* with the goal of asking students to program, but asking them to perform various tasks and consider various analyses that the instructor felt would help them see how the computing community sees such things.

In this paper we discover how incorporating the consideration of culture helped generate a more specific and larger set of desired abilities and knowledge than are traditionally discussed in literature of desired competencies of introductory programming students (see related work). Since these competencies have mostly been studied within the realm of summative assessment (exam) classification and evaluation of student performance, this is not too surprising. However, we believe the community will benefit from consideration of this new approach and from comparison of these competencies with those previous discussed.

5.3   Related Work

Learning Taxonomies. Learning taxonomies are important for computing education because they give the community a vocabulary to use when discussing student understanding and learning - and curriculum supporting it. Taxonomies should support educators in having conversations and in reporting on their courses and efforts to improve what goes on in them.

Bloom's Taxonomy is a highly popular taxonomy. It consists of 6 levels to describe students' cognitive development. In 2001 Anderson produced a revision of Bloom's Taxonomy consisting of 24 categories within two dimensions. The Knowledge Dimension consists of Factual, Conceptual, Procedural and Metacognitive knowledge and the Cognitive Process Dimension consists of Remember, Understand, Apply, Analyze, Evaluate and Create. A more detailed description of each category can be found in [1] (pp 29-31).

The Revised Bloom's Taxonomy's value has been called into question in computer science [79] and has been modified to include "higher application" in order to make it relevant to the field [62]. Other taxonomies, such as SOLO [9] assess cognitive level through student response-type. Our work differs from these in that we move our focus beyond assessment of students' cognitive skill development to consideration of their development in the situated goal of coming to think like and perform the actions of a programmer. Most recently, work by Lister, et. al, has moved to consider desired programming skill development through Piagetian constructivist learning theory [36].

Apprenticeship and Deliberate Practice. Our clicker questions sought to support students during the apprenticeship phase of learning. Similarly, Faulkner created a more authentic experience for novices through Worked Examples: "These are designed to encompass the spectrum of the problem solving process:

- observing the application of programming concepts

- observing authentic problem solving

- cooperative problem solving" [58]

Furthermore, Faulkner asserts that the instructor must display the problem solving in the same environment that the students will be working. [58]

Bareiss takes an approach similar to *Worked Examples* where students are engaged in cognitive apprenticeship learning techniques through coaching. [3] More specifically,

Bareiss defines five techniques needed to coach students: Modeling, Scaffolding and Fading, Articulation, Reflection and Exploration. Scaffolding and Fading is similar to *Worked Examples* in that it focuses on Task Design, Direct Guidance and Feedback.

Worked Examples and coaching are two ways to encourage deliberate practice [53] in novice programmers. As described by Ericsson:

> The theoretical framework of deliberate practice asserts that improvement in performance of aspiring experts does not happen automatically or casually as a function of further experience... The principal challenge to attaining expert level performance is to induce stable specific changes that allow the performance to be incrementally improved. [53]

Deliberate practice involves practicing with the goal of making small improvements at each step, and more importantly, getting feedback along the way to improve the practice when improvement attempts are unsuccessful.

Acculturation. There have been efforts to acculturate students in programming using professional-world approaches. Pair programming [106] is a technique where one student (the driver) works on the lower level parts of programming (e.g., coding) and the other student (the navigator) works at the higher-level (e.g., design and integration). The students switch over being driver and navigator throughout the assignment. Pair programming has been effectively used in the classroom [16] and is an effective way to encourage students to engage in confident practice while learning and consequently doing better in their programming. [17]

Coding Dojos [64] harken back to something like the masterapprentice model to support acculturation. Seemingly not formally studied, Coding Dojos are places where programmers can go and watch others program or be watched programming. The idea is if novices are watching experts they can see the choices experts make and if experts are watching novices they can guide them. Compared to pair programming this emphasizes the community and culture aspects of situated cognition.

5.4 Creating the AT Taxonomy

5.4.1 Methodology

The Transition Levels

We developed the ***Abstraction Transition (AT) Taxonomy*** through analysis of learning materials from a CS0-type course teaching Alice serving approximately 570 students as a general education course at a large research-intensive institution in the US. The learning materials we analyzed were multiple-choice clicker questions (MCQs) posed *during lecture* for students to answer and analyze in groups (via the Peer Instruction pedagogy [105]). These formative assessment questions sought to support learning through cognitive apprenticeship. However, most exam questions for this course were modeled very closely on these MCQs.

All but one author (Foster) was involved in the teaching and delivery of the course - creating questions and interacting with students discussing them during lecture. After the end of the term, Fecho reviewed the in-class clicker questions and generated categories stemming from her instructional experience with those questions. She identified three levels of abstraction in the questions: English, CS Speak, and Code.

The primary instructional team (Simon and Cutts) did not consciously consider "levels of abstraction" in clicker question development but, in retrospect, this focus was not surprising. On the second day of class, the instructor used Figure 5.1 as an overview to students of what they would be asked to do in the course.

In our analysis we refined Scenario to English, Design to CS Speak and Implementation to Code. We use English instead of specification to indicate that no specialized language is used describing the intended behavior. In Alice, an example of English would be "make a skater spin around 3 times". In CS Speak this might be "have the iceSkater turn 3 revolutions". Figure 5.2 shows what this would be in code.

In Alice, for the simplest cases, English and CS Speak can be quite similar, and are

Ways to demonstrate
understanding of programming

Writing

Reading

Scenario: Idea in your head

Design: Storyboard

Implementation: Program
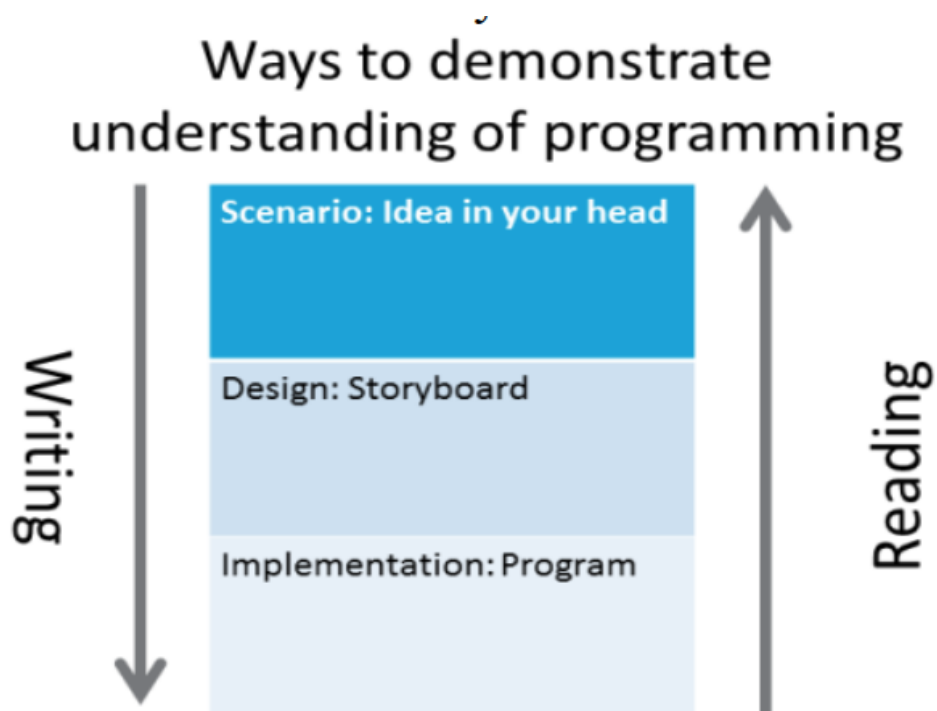
Figure 5.1. Slide presented in second lecture revealing (in hindsight) instructional focus.

iceSkater ▽   **turn** left ▽   3 revolutions ▽   more... ▽
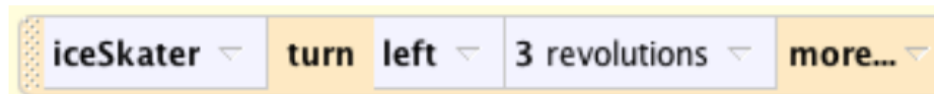
Figure 5.2. In CS Speak, this might be "have the iceSkater turn 3 revolutions."

Relational Operation Expression:
How would I write a relational operation expression that returns true when an igloo is blue?

A. whichIgloo . color == [ ]

B. whichIgloo . color != [ ]

C. whichIgloo   set color to [ ] more...
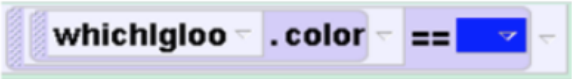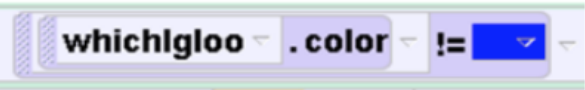
D. None of these, you need an operator like < or >

Figure 5.3. Example MCQ: Transition from CS Speak to Code.

differentiated by the specific methods and objects defined in Alice.

Through continued discussion we developed a taxonomy that describes those activities and tools, and ways of thinking about using those activities and tools, that the instructor felt students need to engage with in order to understand how computing and computing professionals work. A clear finding involved our emphasis on developing students' skills in transitioning between levels of abstraction. This was especially evident because of our use of MCQs - with their "stem" and "option" components. Figure 5.3 represents a transition from the CS Speak abstraction level to the Code abstraction level.

To better clarify the difference between English and CS Speak, we also provide a question transitioning from English to CS Speak. This question is asked in English and the answer choices require the students to use their understanding of CS Speak:

Suppose there are customers waiting in line at the store. You want to serve each customer one at a time, so each one should walk to the counter one at a time. How could you do this? A. Use a DoTogether tile, B. Use a DoInOrder tile, C. Use a ForAllTogether tile, or D. Use a ForAllInOrder tile.

After categorizing a few sample questions, we began to acknowledge that some

question and answer sets remained within one level of abstraction, mainly CS Speak or Code. However, these non-transition questions still lie explicitly within an abstraction level and bring out interesting issues from the point of view of cognitive apprenticeship. CS Speak was especially interesting - as it represents the "lingo" of expert programmers. We found we asked two types of tasks with these questions. We term them: CS Speak Apply and CS Speak Define. That is to say, a CS Speak Apply question would engage students in identifying what CS Constructs/Concepts were involved in an algorithm or in deciding which/how CS Constructs/Concepts might solve a goal described by a CS Speak description. For example, CS Speak Apply would be:

> If we write a method called drive, which would not makes sense as a parameter to control how drive occurs? A. Destination, B. How Fast, C. Which car, or D. Car color

A CS Speak Define question, on the other hand, would engage a student in explicitly reflecting on what CS Construct/Concept does or what it is used for in the programming community. For example, a CS Speak Define question would be:

> Which of the following is the best explanation of what makes a good parameter? A. It's something that supports common variation in how the method is done, B. It's got a meaningful name, C. It can be either an Object or a number, or D. It helps manage complexity in large programs

Questions that remained in the Code abstraction level were codetracing questions. Although this may not be a transition through abstraction levels, it represents a skill all experts have and sometimes employ in that it requires tracing of code without any analysis or deeper understanding past that of the logic required to solve the problem. For example, a question in this transition would be:

Furthermore, we agreed that due to the nature of Alice, there might be questions that were unrelated to computing concepts and solely related to Alice-specific issues. These, we gave a classification of "other" and did not expect these types of questions to appear in other programming languages. We do not report on these here.

Figure 5.4. *How far up will the bee move in the second instruction, given that the tulip is 0.3 meters high and our fly is 0.1 meters tall? A. 0.2 meters, B. 0.3 meters, C. 0.35 meters, D. It's not possible to tell, or E. I don't know*

Different Types of Questions

While performing this analysis we noted an orthogonal question classification descriptor: rationale questions and mechanism or definition questions. We summarized these as *why questions* and *how questions* (respectively) and they are defined in Table 4. *How questions* are a norm in computing education. The *why questions* are a natural outgrowth of making explicit the need to rationalize, culturally, decisions of activity and tool use in programming problems. That is, rather than just hoping students internalize how programmers think, we use *why questions* to *explicitly* engage students in discussion and evaluation of various programmer rationales.

An example of a why question, with AT level 32, shows two code snippets that have the same outcome and then asks: What is the BEST explanation of why one is better than the other? A. Option 1 is better because it is shorter, B. Option 1 is better because it does the least number of "checks" (or Boolean condition evaluations), C. Option 2 is better because it makes clear exactly what the "checks" (or Boolean condition evaluations) are, or D. Option 2 is better because it has a regular structure with empty "else" portions

On the other hand, Figure 5.5 is an example of a how question, with AT level 13 asking the student to choose code that would match the description.

To improve the clarity and support reproducibility of AT Taxonomy categorizations, we selected a random sample of 20% (27/133) of all the MCQs asked over the term and had two authors categorize and discuss them based on our taxonomy. From that the two authors iteratively refined the taxonomy. Once finalized, those two authors individually categorized
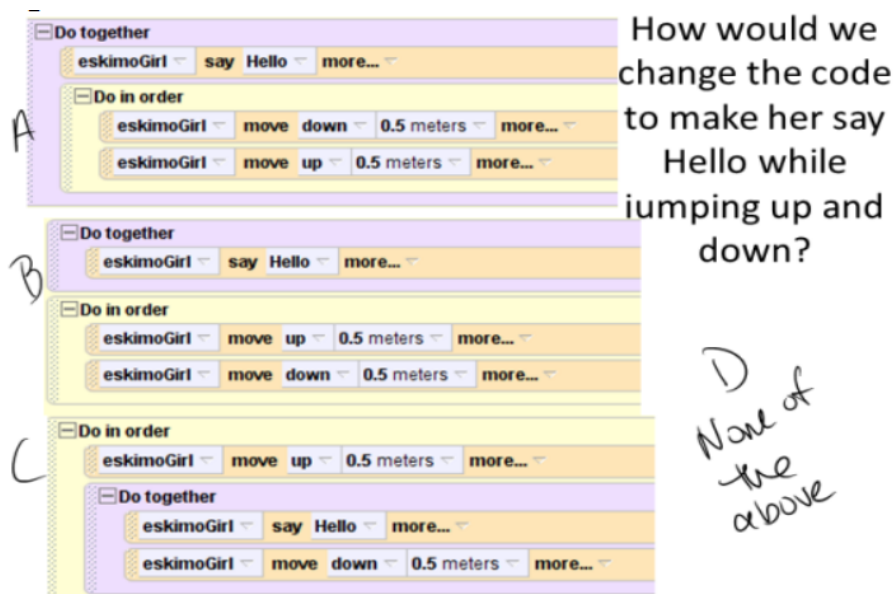
Figure 5.5. Example question showing a how-type question.

a second random 20% (28/133) of the MCQs for both transition number and type. The authors reached an 87% inter-rater reliability (counting matches for agreement on transition number and agreement on type). Finally, one of the authors coded the remaining 60% of the MCQs from the CS0 course.

5.4.2   Results

The transition levels used to categorize questions are defined in Table 5.4. The types (why and how) used to further categorize questions are defined in Table 5.5. Together, Tables 5.4 and 5.5 define the Abstraction Transition Taxonomy. Table 5.3 below shows the final results of applying the classification scheme to our in-class Peer Instruction MCQs. Due to lack of space, we don't show the breakdown of how and why questions in each category, but overall 21% of questions were why and the rest were *how*.

To read the chart, look in the row for the level of the question stem, and then move to the column that indicates the option level. This shows that 9% of the questions asked in lecture through MCQs had a question stem in English and the answers were in Code,

Table 5.2. CSE in-class MCQ Distribution. Overall 21% were why questions.

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	6%	9%
CS Speak: 2	2%	8%(A) / 28%(D)	6%
Code: 3	12%	15%	9%

making 9% of the questions an AT level 13. Similarly, 12% of questions were AT level 31, 6% were 23 and 15% 32.

5.5   Applying the AT Taxonomy

5.5.1   Methodology

Returning to our original goal, we used the AT Taxonomy to help both overview and tease out the manners in which this course sought to help students develop computational thinking practices - or to reach a basic level of understanding of the programming community. We noticed that our CS0 course was fairly distributed amongst all the transitions, with perhaps a surprising focus on use (2A) and understanding (2D) of CS Speak.

With this enlightened view of the course, we explored how our summative assessment (final exam) differed from or matched our formative assessment (in-class MCQs). We were also interested to explore other introductory CS courses' (CS0, CS1 and CS2) summative exams to see a) if our taxonomy would be applicable to others' assessments and b) if our course differed from theirs (as measured by summative assessment).

We analyzed 7 exam sets taken from 4 sources: recent assessment literature, a standardized exam from the Advanced Placement series in the US, a complete exam with 8 questions from a large mid-west US Institution found on the web, and our own CS0 exam. Of note, the CS2-DCER data represents 3 complete exams from the year 2009 (the most recent available) accessed through the DCER project [127]. Table 5.3 summarizes the datasets.

Table 5.3. CSE in-class MCQ Distribution. Overall 21% were why questions.

Dataset Name	Language	Number of Quesitons	Complete or partial set
CS0 - Our Exam	Alice	37	Complete
CS0 - Meerbaum-Salant	Scratch	5	Partial
CS1 - Lister	Java	12	Partial
CS1 - Lopez	Java	24	Complete
CS1 - APCS	Java	22	Partial
CS1 - R1	Java	8	Complete
CS1 - DCER	Java	143	Complete

5.5.2   Results

Transition Deistribution

As noted in Section 5.5.1, the in-class clicker questions from our CS0 course are fairly distributed across the taxonomy categories. How were these skills reflected in summative assessments? Tables 5.6 through 5.12 show the AT Taxonomy applied to the exams.

CS0. It is thought provoking that our exam question distribution was not reflective of the kinds of questions asked in class. Rather, the exam is almost "all about code" - all but four (related) questions involve an AT level 3. This suggests that even if an educator is fully aware of the need for more diverse cognitive apprenticeship in programming culture and tasks, the norms of examinations may send a very different message to students about what is important.

Of the set of assessment questions reported in Meerbaum-Salant's Scratch paper [107] 60% of them fall in the 23 or 3 AT level. The course is therefore assessing the students' coding ability - can students write or trace code, possibly given a storyboard or pseudocode description. The focus of the course is to introduce computing concepts, but it is clear that

Table 5.4. Definition of Transition Levels. The numerical coding in the leftmost column represents the particular transition where 1 is English, 2 is CS Speak and 3 is Code. AT levels that do not follow this scheme are: CS Speak Apply (2), CS Speak Define (2D), and Code (3).

Transition Level	Description
12	**English-CS Speak:** Given an English description of a scenario or goal, choose a technical description in "CS Speak" of the process to achieve the goal in the form of an algorithm or storyboard.
23	**CS Speak-Code:** Given a technical description (CS Speak) of how to achieve a goal, choose code that will accomplish that goal.
13	**English-Code:** Given an English description of scenario or goal, choose the code that will accomplish that goal.
32	**Code-CS Speak:** Given code, choose either a description in CS Speak of the goal of the code, or choose which coding constructs are used within the code.
21	**CS Speak-English:** Given a description in CS Speak or coding construct, choose an English description that describes what the CS Speak does or the goal that it accomplishes.
31	**Code-English:** Given some code, choose an English description that describes the goal of the code, not the step-by-step process, but the overall goal.
3 (Apply)	**Code:** Given code and conditions, choose a result from executing the code. This is "code tracing" and does not imply overall goal, but simply the execution of the code.
2 (Apply)	**CS Speak:** Given a CS Speak description, choose the coding constructs that are present (or vice versa).
2D Define	**CS Speak:** Provided coding constructs choose a technical description of their purpose or how they work.

Table 5.5. Definition of Types.

Type	Description
Why	Choose a **rationale** for why a statement of answer choice is correct or incorrect
How	Choose an unambiguious answer that depends on **mechanism or definition.**

Table 5.6. CS0 Final Exam

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	0%	27%
CS Speak: 2	0%	11%(A) / 0%(D)	11%
Code: 3	8%	27%	16%

Table 5.7. Meerbaum-Salant

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	0%	0%
CS Speak: 2	0%	0%(A) / 20%(D)	40%
Code: 3	20%	0%	20%

Table 5.8. Lister

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	0%	0%
CS Speak: 2	0%	0%(A) / 0%(D)	42%
Code: 3	0%	0%	58%

Table 5.9. Lopez

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	0%	13%
CS Speak: 2	4%	8%(A) / 8%(D)	8%
Code: 3	13%	17%	29%

Table 5.10. AP CS A

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	5%	9%
CS Speak: 2	0%	0%(A) / 0%(D)	14%
Code: 3	0%	23%	50%

Table 5.11. Midwest R1 Institution

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	0%	13%
CS Speak: 2	0%	25%(A) / 0%(D)	50%
Code: 3	0%	0%	13%

the assessments are also focused on code.

CS1. Both the Lister [93] and Lopez [94] datasets reflect efforts to study student abilities on commonly desired programming skills across institutions and countries. Lopez was a part of the BRACElet [159] project where assessment of Lister's questions led to the development of Explain in Plain English (EiPE) questions. The Lopez dataset is the only set that reports assessment of a range of AT levels. This is likely because he sought to explore a new type of question within the framework of existing exam questions. Lopez's identified types of questions are: EiPE (AT31), Code Writing (AT13), Tracing 1 (AT3) and Tracing 2 (AT3).

Both the APCSA sample and the R1 exam show strong emphasis on level 3 activities.

Table 5.12. DCER

Q/A	English: 1	CS Speak: 2	Code: 3
English: 1	—	2%	0%
CS Speak: 2	0%	46%(A) / 29%(D)	3%
Code: 3	0%	5%	14%

50% of the sample questions for the APCSA exam are AT 3 level requiring code tracing with no abstraction transition and no engagement with contextualization of the problem. In the R1 exam, we also see the emphasis on transitions focusing on the code - 75% of questions fall into AT 13, 23, or 3.

CS2. Quite strikingly, different skills seem to be valued in CS2 exams. These exams represent 3 complete exams from the DCER international dataset. Nonetheless, we find a switch in emphasis from code (and transitions to/from it) to CS Speak; asking students to apply appropriate computing concepts or indicate understanding of the purpose of use of those concepts.

Type Distribution

Of the questions asked on our CS0 final exam, 11% of them were *why questions* (89% were *how questions*). Of these why questions, 75% of them were 32 and 25% of them were 23, meaning some kind of CS Speak was involved. Of the remaining sets, there are only three exams that have any *why questions*; CS1-Lopez, CS1- R1 and CS2-DCER. The distribution of *why questions* across these exams also never exceeds 15%. The remaining sets, CS0- Meerbaum-Salant, CS1-Lister and CS1-APCSA, have no *why questions* whatsoever. Although why questions might be considered challenging or subjective to grade, we hope instructors will consider the value of asking students to be able to explain their rationales and therefore situate their abilities and use of tools in the programming culture.

5.6 Discussion

5.6.1 Culturally-Informed Learning Outcomes: *Why* Questions are Critical

Reflecting on the AT Taxonomy in light of situated cognition theory - the amalgamation of activities, tools and culture that define the programming community - we are struck by the relative scarcity of *why questions* among both our clicker and exam questions. Surely

the integration of culture with activity and tools means that "just" being able to apply a tool or perform and activity is not sufficient. We wouldn't claim one was a vetted member of our community who couldn't explain *why* – for any question in any of our 9 AT categories.

But *every clicker question is a why question* - regardless of whether the question's stem or options make that why explicit. It's inherent in the Peer Instruction pedagogy that questions are developed based on the kinds of discussions one wants the students to have (through the vote, discuss in small groups, revote procedure). Quite specifically, in Peer Instruction, the instructor frequently exhorts students to discuss not only why the right answers are right, but also why the wrong answers are wrong. Students aren't just "doing" an AT transition, but they are rationalizing their thought process and describing what they did and thought. In their groups, the class-wide discussion where they hear the explanations of other groups, and the instructor wrap-up, they are apprenticed in how programmers think about problems and rationalize actions.

We claim this means that we have identified 18 learning goals for development of programming students: All 9 categories have both how and why goals. So for example, by the end of the course students should be able to:

- In the context of a computational problem, read an English description and write code to solve the problem (13 how)

- In the context of a computational problem, read an English description and write code and describe why that code solves the problem in the English description. (13 why)

The truth is, as much as we do care about 13 how, in the absence of demonstrated 13 why ability, we cannot claim a student has, from a situated cognition perspective, become proficient as a practitioner in the community. Similarly though rarely a featured part of instruction, we need goals that students should be able to:

- Read code and give an English description of what it does (31 how)

- Read code and explain why their English description of what it does is correct (and possibly why another is not) (31 why)

These goals contribute to defining the community practitioner's ability to read the code of others, perhaps for code maintenance, modification, or debugging.

5.6.2 Can Summative Assessments Measure Computational Thinking?

We began this work with the goal of summarizing and elucidating our in-class learning materials. We hoped this would help us understand how a CS0 course which, at surface level, is focused on programming was achieving the positive changes in student confidence and abilities previous reported [44]. This paper has shown that a new view on how we want to think about, teach, and assess introductory programming courses can be found by stepping back and focusing on cognitive apprenticeship of computational thinking skills.

Interestingly, most of our identified 18 learning outcomes are not assessed on summative assessments - based on a sample selected primarily from recent research literature. Notably, *why questions* represent less that 15% of any exam - with three of the exams evidencing no *why questions* at all. Does this bother us as a community? We hope this work spurs discussion of that question.

Through our recent experiences in supporting high school teachers in the CS Principles project, the need to assess *why* ability has sharpened. We are beginning to explore multiple-choice questions (of the *how* variety) and asking students to explain in written English form how they analyzed it and why the wrong answers were wrong. These responses are more challenging to grade, but seem to be both more valuable and easier to create than multiple-choice *why questions* where the options are various explanations or rationalizations. In future work, we hope to further explore the potential and validity of these questions.

5.7   Conclusion

We propose that situated cognition theory, with its focus on learning for application in the "real world" and its focus on developing expertise within the context of a community is a useful lens for reconsidering programming instruction and perhaps computer science instruction more generally. A number of factors support this idea. First, our community has norms that seem particularly challenging for outsiders to understand. The common stereotype of a computer programmer makes it clear that we are to be considered other and our actions incomprehensible. It may be that part of our collective recruitment and retention issues (at least in the US) stems from the lack of attention to acculturation of new members. The AT Taxonomy defines 18 learning outcomes that explicitly address the required culture, activity, and tools required for members of the programming community which we believe needs to extend, at least minimally, to embrace everyone in modern, digital society.

5.8   Acknowledgments

Chapter 6

Using Exploratory Homeworks to Create an Active Learning Tool for Textbook Reading

Chapters 4 and 5 focus on the enculturation of students during lecture, however students are also spending many hours and hundreds of dollars engaging with textbooks outside of lecture. In this chapter we introduce *Exploratory Homeworks* which is an iteration of textbook design to engage students in a more active reading experience. Specifically, this is critical for a Peer Instruction course, since it requires students to complete pre-lecture reading to be prepared to discuss during lecture.

6.1 Activating Constructionist Learning in Pre-Lecture Reading

Active learning is known to be a best practice in classroom pedagogy. Research shows that people don't learn very effectively from being "told" what to do. They gain a much deeper and meaningful understanding by constructing their own learning through engaging with a concrete problem [14]. Various classroom techniques such as Peer Instruction, POGIL (Process Oriented, Guided Inquiry Learning), PLTL (Peer Led Team Learning), studio-based instruction and lab-centric instruction replace or significantly augment lecture in order to accomplish this [105, 51, 77]. However, the classroom is only one learning

resource common to undergraduate education.

The next most obvious resource (at least to students, since they pay for it) is the textbook. Textbooks are generally expository in nature and textbook reading by students is often very passive. A report from a genetics course finds students' attitudes and descriptions of how they read textbooks to be quite novice - very much out of line with how faculty would want them to read it [12].

Teaching a programming language has a unique aspect that we seek to take advantage of - one can interact with the computer to get feedback on behavior of programming constructs. In fact, anecdotally, we posit that experts rarely learn a new programming language by sitting with a static, passive, textual or expository resource. Instead they combine explanatory sources with guided or self-defined micro-experimentation - isolating a specific construct or concept of interest and developing the simplest possible code to enable them to explore its behavior.

In this discussion paper we introduce a new "active learning" tool for supporting students in successful, and more expert-like, engagement with a traditionally passive resource - the textbook. *Exploratory homeworks* are written guides that engage students in their first exposure to and experimentation with a programming construct. In our experience, they augment the textbook by outlining small learning goals (at the subsection level or smaller), and provide specific guidance on how to use the book.

First, students are asked to both open the textbook and launch their programming language development environment. They are then asked to "follow along" building up an example code, and are stopped frequently and confronted with questions to pose to themselves, experiments in modifying the code, and explanations of what they should have just experienced. Afterwards students are provided with example questions that indicate the level of understanding they should have achieved by doing the homework.

In our context (and what we recommend), this homework is done individually by

students before each lecture period, covering introductory material for that lecture. Students are encouraged to do the homework, which is not itself graded, by a 3-4 question quiz for credit given at the beginning of every lecture. This both supports and flows naturally into a lecture that also utilizes an active learning-based pedagogy. In such lectures, since students are involved in constructing their own understanding, not just listening, there is usually less time for standard expository lecture. Exploratory homeworks provide students the opportunity to become familiar with basic material before class, allowing the instructor to spend their valuable time with students working with more advanced, nuanced, and/or in-depth applications and explanations of the material.

Next we provide background and related work for the theoretical underpinnings supporting the efficacy of exploratory homeworks. In Section 6.3 we describe the setting of the course and in Section 6.4 we outline the flowchart guiding design of exploratory homework and provide an example from a section on looping. In Section 6.5 we provide qualitative and quantitative data on students' experiences with and perceptions of the value of the homeworks. In Section 6.6 we discuss our experiences and provide recommendations for instructors seeking to develop exploratory homeworks in their programming courses.

6.2   Related Work

Lecture. Various forms of active learning are much studied both in STEM higher education and in computing education. Labcentric instruction replaces all or most of lecture with extensive guided lab work, featuring frequent opportunities for feedback - something designed to help improve students' self-assessment skills [91]. Peer Instruction, POGIL, PLTL, and studio-based instruction all fall into the realm of collaborative pedagogies, and are designed for lecture or discussion section-type settings [105, 51, 77]. Modifying the lecture period to be more engaging and collaborative is an important pedagogical change to help students learn, but the lecture period is typically only a couple hours a week. Students

spend the majority of their time outside of lecture, where they often do not have access to an expert (the lecturer) to clarify or ask questions.

Closed-Labs. Frequently computer science lectures are accompanied with a lab-section. This provides students with an additional few hours a week where they have access to an expert to help guide them or explain complex concepts. Lab manuals and tutorials [157] offer students guidance when in closed-lab settings, though they are still distinct from exploratory homeworks in that 1) The environment in which they are used is a completely guided, "safe" environment and 2) The goal of a lab manual or tutorial is typically to help the students understand a concept and actually write a program. Though exploratory homeworks guide students through complex concepts and writing programs, they are also focused around teaching students *how* to read a computer science textbook or text resource. Lab manuals or tutorials are separate from traditional textbooks or online resources in that they are completely contained. Exploratory homeworks ask students to go outside of the homework to seek information in order to progress (which is what is typically done when experts are learning a new language or concept).

Worked Examples. Casperson also realized that the textbook is a relatively mal-used resource in computer science education. He understood "A static program example presented in a textbook reveals nothing about the process of developing the program." [7] Casperson therefore developed Worked Examples in computing; problems that actively engage the students in a step-by-step process of learning the concept while attempting to minimize cognitive load. This was significant (though perhaps still in need of development) because it encouraged educators to engage their students in process rather than just results. Exploratory homeworks also focus the student on process. However, we specifically focus on directing students in understanding more "standard" static computing instructional text and translating it into problem solving and programs.

Our Work. Exploratory homeworks are uniquely designed to enhance at-home

textbook reading. This setting is very different from both lecture and closed-lab. Typically reading is a sedentary action students engage in because "they are required to". Though almost every university has some form of student academic support unit that recommends various approaches for effective textbook reading, they are typically not discipline specific [80]. And we believe many of these suggestions do not encourage computing students to engage in thinking through problems and following or analyzing code. Kolb [85] describes a learning process (the Kolb Learning Cycle) that involves combinations of perceiving and processing information. The Kolb Learning Cycle involves four learning styles: *concrete experience, reflective observation, abstract conceptualization* and *active experimentation*. Typically, educators apply the Kolb Learning Cycle to a lecture or lab component. Exploratory homeworks seek to bring Kolb-like styles to the less-structured reading experience.

Finally, the efficacy of exploratory homework can be argued from another aspect of social constructivist theory - authentic practice. In [19] they argue that all too often "school-based" instruction ignores the situated nature of knowledge and suggest that development of expertise be guided through cognitive apprenticeship. That is, we must engage students not just in abstracted conceptual knowledge, but also in the authentic practices that experts in the practice use. We must engage students in "doing as we do" and support them in developing the ability to "think as we think".

6.3   Setting

6.3.1   The Course

Exploratory homeworks were introduced in a CS0 course that was taught using the active learning pedagogy Peer Instruction (PI) [105]. The course taught 440 mostly non-major, undergraduate students at a large R1 institution in the western USA. Students learned the Alice programming language and Excel. The course was designed to help

students develop a general understanding of computing concepts and constructs.

6.3.2 Incorporation of Exploratory Homeworks

Exploratory homeworks were incorporated into the course to enhance pre-lecture reading. The course structure was as follows:

- Students were assigned pre-lecture exploratory homeworks.

- At the beginning of the 90-minute lecture, a quiz was given to assess what students should have learned from the homeworks. A subset of the quiz questions came from questions provided in the homework.

- Lecture then continued in a typical PI fashion. Furthermore, the instructor could delve deeper into a problem by extending examples from the homework.

- The exploratory homework, quiz and lecture cycle took place twice per week.

- The week following both lectures, students were required to demonstrate understanding through a directed, closed lab involving code writing.

- Midterm and final exams also included questions of the style introduced in the exploratory homeworks.

Students were able to spend as much or as little time as they wanted or could manage on the exploratory homeworks. The homeworks were not graded, but every lecture students were directly rewarded for completing the homeworks via a quiz. Additionally, the homeworks served to help students engage in the kind of analysis processes they would be called to perform during peer discussions during lecture.
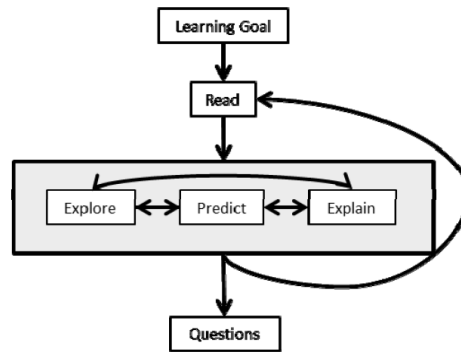
Figure 6.1. Exploratory Homework Structure

6.4 Example Homeworks

The course used the book "Learning to Program with Alice" [46]. This book is written in a fashion that easily supports active learning. Each chapter is divided into sections that ask the student to create (or use an existing) Alice program and then enhance or modify it based on instruction and guidance from the book. The exploratory homeworks were developed to focus on the led examples in the book, enhancing them with details and encouraging the students to further explore the code, make predictions and "play around". Homeworks were typically broken up into 3-4 sections. Figure 6.1 shows the structure of the exploratory homeworks.

The purpose of this structure is to actively engage students in learning the basics of concepts and constructs.

- **Learning Goal** - One sentence description that tells students what they will learn and serves as a reference for study.

- **Read** - Asks students to read (and follow along coding) a specific section of the book.

- **Predict** - Focuses students on understanding a misconception through code changes, predictions, tests and reflections.

- **Explore** - Allows the students to gain confidence in manipulating, predictin-g and

Learning Goal

LEARNING GOAL 1: Reading and Predicting What Loops Do

Start Alice and open your book to section 7.1.

Read

Load up the world called HW9_bunnyWorld that can be found here. **When you download it, make sure it's not saving as zip.**

Follow the book to page 211 and fill in the method myFirstMethod, but using the world we have provided for you, **not** the one in the textbook folder of Alice. (If you are confused on how to write the method with the loop, please be sure to seek help from a tutor.)

Explore

The section ends with a suggestion that you can have more complicated statements within the loop body. Let's try to make this bunny a little more interesting.
- In the HW9_bunnyWorld that is provided to you for this assignment, there is a bunny.dance method. Read this method, try modifying it to make him do a specialized dance, not just the generic one everyone is doing. After you have played with the bunny.dance method and made your bunny do an awesome dance, go back to your myFirstMethod.

Predict

- We want to make the bunny do a "hop-once-dance-once" combo, which he then repeats maybe 10 times. Add a DoInOrder inside of the loop. Put the call to bunny.hop followed by a call to bunny.dance inside of theDoInOrder.
- How many times will the bunny hop? How many times will the bunny dance? Is it 10 hops followed by 10 dances?
- Now Press Play. Did it do what you thought it would?

Explain

One of the most common problem people have with loops is knowing what goes "inside" the loop tile and what goes "outside" (before or after) the loop tile. Let's try moving instructions around in and out of the loop and predicting what it does:

Predict

- Drag the call to the hop method out of the loop tile to just above the blue loop tile. What will your world look like when you hit play? **PREDICT FOR YOURSELF** (because something like this would be a great exam question). Play and see if you are right.
- Next Drag the hop tile from before the loop to AFTER (below) the blue loop tile. What will happen this time? Try it out.

Predict

- Try copying the hop method call (right click on the tile, select copy) and paste a copy of it ABOVE the loop as well. (There should now be 2 tiles with a hop call on them). Predict...

Predict

Delete the hop tile before the loop. Drag the hop tile from below the loop back into the loop, but put it below the dance method call in the DoTogether tile. How will this be different that the first one we started with? When you are done, drag the dance tile below hop to get back where you started.

Predict

It makes more sense that when a bunny hops closer to the camera, it should get bigger. So inside the DoInOrder (still inside the loop) add a DoTogether and move the bunny.hop call into the DoTogether followed by a bunny.resize(1.05). Leave the bunny.dance in the DoInOrder, right after the DoTogether.

Press Play, did the bunny get bigger every time it hopped closer to the camera? How many times did the bunny hop, get bigger and dance?

Explore

Now try dragging other methods/actions into and around the loop to do a more complex dance routine. Any old bunny can hop then dance then hop then dance...make your bunny do something special. Maybe have him hop-dance-spin, or have a "grand finale" that only happens once after he hops-dances.

Questions:

Questions:

1. Suppose I have a program that makes a guy wave after he walks up to a girl. How would you change it to make the guy wave repeatedly AS he walks up to the girl?

2. Which loop construct(s) is(are) incorrect?, Why?

- Loop 5 times
  Do Nothing
- Loop Infinity times
  Do Nothing
- Loop Alice.DistranceTo(Rabbit) times
  Do Nothing
- Loop Alive.MoveTo(Rabbit) time
  Do Nothing

Note:
Original
Exploratory
Homework
includes
screenshots
from Alice.

3. Why is Loop also called "Counted Loop"

Figure 6.2. Example Section from Exploratory Homeworks

reflecting on changes in code.

- **Explain** - Points out common misconceptions or validates students' learning through logical reasoning.

- **Repeat** - The students will do the above four sections until the required reading is complete (order can vary).

- **Questions** - Allows students to self-assess their understanding and repeat sections if they find they are confused.

Figure 6.2 provides an example of one section of an exploratory homework, as well as the questions asked of them to check their understanding. The full texts of all 15 homeworks are available at www.peerinstruction4cs.org under CS Principles.

6.4.1 Learning Goal and Reading

Referring to Figure 6.2, the first step is a high-level learning goal: **Reading and Predicting What Loops Do**. The explicit goal gives students context for the homework and the chance to find examples quickly in the future.

The first two paragraphs then ask the student to read and follow along in the book until a certain point. This step involves Kolb's *concrete experience and reflective observation* because students are asked to program what is in the book (passive) but also to ensure understanding (review). Students are discouraged from reading the entire text before attempting to follow the guided examples. Instead, they read small portions and then stop to reflect, predict and "play around" with the code to satisfy their own interests and curiosities. This encourages the students to work with small, doable goals versus large, daunting problems.

6.4.2 Explore

We choose to stop student reading where the book suggests that the student "try something different" for the bunny's dance. We enhance the book with some guidance for this task. The first bullet says:

- **In the bunnyWorld that is provided to you for this assignment, there is a bunny. dance method. Read this method, try modifying it to make him do a specialized dance, not just the generic one everyone is doing. After you have played with the bunny.dance method and made your bunny do an awesome dance, go back to your myFirstMethod.**

This bullet emphasizes to the student how to actively engage in learning a new programming language using three techniques: 1) Understand the current implementation, 2) Modify the code, 3) Test to make sure the result is what was expected. This step is Kolb's *active experimentation and abstract conceptualization.* This also encourages students to read more authentically (Kolb's *concrete experience*).

Next, we ask them to follow instructions on creating code, but before running it, to predict what will happen based on the code changes. This step is a form of *reflective observation* in the Kolb Cycle. This active learning skill is especially useful in debugging and will help students recognize errors based on prediction and reflection about code changes. If the prediction does not match, it can either be an error in the code change, or an error in their prediction, but either way the student gains more insight into the problem.

- **We want to make the bunny do a "hop-once-dance-once" combo, which he then repeats maybe 10 times. Add a DoInOrder inside of the loop. Put the call to bunny.hop followed by a call to bunny.dance inside of the DoInOrder.**

- **How many times will the bunny hop? How many times will the bunny dance? Is it 10 hops followed by 10 dances?**

- **Now Press Play. Did it do what you thought it would?**

These steps walk the students through the process:

Code change – Predict – Test

They also encourage them to re-visit the code and their hypothesis should there be a discrepancy.

6.4.3   Explain

The next section points out a common mistake made by novices:

One of the most common problems people have with loops is knowing what goes "inside" the loop tile and what goes "outside" (before or after) the loop tile. Let's try moving instructions around in and out of the loop and predicting what it does:

By pointing out the struggles novice programmers have with nested loops, the student is able to focus their attention on the intricacies of the loops that they will be working with in the next **Predict** section. Students engage in Kolb's *abstract conceptualization* and *active experimentation*. By presenting a difficult concept, the students gain confidence when they know that they are progressing and are prepared for lecture.

6.4.4   Questions

At the end of the entire exploratory homework there are 3-5 conceptual questions that the students should be prepared to answer. This is Kolb's *reflective experience*. The goal is to support students coming in to lecture prepared so the instructor can delve into more complex concepts.

1. Suppose I have a program that makes a guy wave after he walks up to a girl. How would you change it to make the guy wave repeatedly AS he walks up to the girl?

This is asking them to essentially do **Explore** and **Predict**, where they have to explore what already exists, then predict the outcome of changes they would make and then test to make
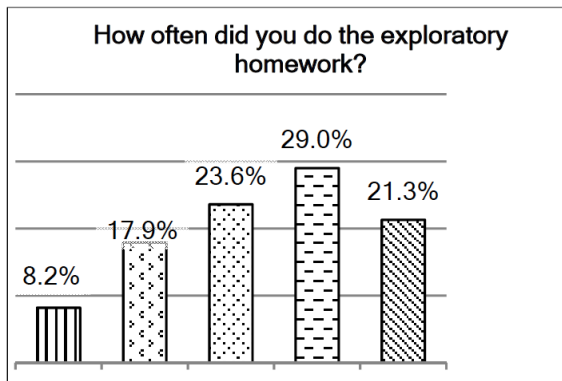
Figure 6.3. Student Survey Results for "How Often"

sure the actual outcome matches the scenario, or expected outcome.

The questions in this section are not meant to be tested in the Alice programming environment; rather the students are expected to be able to do them on paper. At the beginning of lecture, the instructor will give a quiz as described in Section 6.3.2. The students will not have access to any computer or notes; therefore it would behoove them to practice the questions in a similar environment.

6.5   Student Experience

From both an optional midterm survey (N=156) and a required end of term reflection[1] we can provide information on the experiences students had with exploratory homeworks[2].

6.5.1   Usage

Although we assigned exploratory homeworks for every lecture and had a quiz covering the material at the beginning of each lecture, we're not so naive as to think students always did the homework, or did it in the manner we asked them to. When we asked students how often they did the exploratory homework, we were pleased that just over 50% answered most or all of the time. Additionally only 1/4 of the class said they rarely or never did it.

[1] All data is from the end-of-term survey unless otherwise noted.

[2] Students were told "effortful and honest answers, both positive and negative, will receive full credit."

Table 6.1. Students Approach to HW. (5% of students did not do the homework at all.)

Approach (recommended approach in italics)	%
Read the textbook but don't do the code in Alice	33%
Create the code in Alice as I read the textbook	*30%*
Just create hte code in Alice, skimming the textbook	15%
Create the code in Alice after reading the textbook	9%
Just read over the homework	8%

Why is this pleasing? We also asked students about their homework habits in other classes. 45% stated that they almost never do non-required homework in their other classes (though 26% thought they should, 19% said they didn't need to). Since our homeworks weren't physically turned in, but motivated by a quiz, we were comparatively pleased.

We also asked students to indicate an approach that most correctly described the approach they took to their homework. Table 6.1 shows that only 30% of students followed the core portion of our approach: to work with the textbook and program simultaneously.

Perhaps to be expected, the largest number of students said they only read the textbook - negating our true intent (though we don't know if they also read the homework prompts (prediction prompts, explanations) as well). Based on student open-ended comments we expect that lack of easy access to a computer with Alice installed is a key factor. Traditional study locations in the dorm or library would be a barrier unless the student had their own laptop with Alice installed. In doing a post-hoc analysis of grades we were pleased to discover that students who created the code in Alice both in following the exploratory homework correctly and incorrectly (e.g. reading the textbook first or skimming it) did do significantly better (4.6 points) in the course than those who did not do the exploratory homeworks.

Finally, 76% students reported spending between 30 minutes and 2 hours on exploratory homework per week (averaging 40 minutes per homework). 10% reported spending

Table 6.2. Homework Survey Results

	Always	**Sometimes**	**Never**	**Total**
Lecture	58%	41%	1%	156
Labs	42%	54%	4%	156

more than 2 hours per week. We were expecting students to spend a somewhat greater amount of time on the exploratory homeworks. However a) 33% weren't actually doing the coding and b) student self-report of study time is a ¡100% reliable measure. On the other hand, the fact that most students *reported* spending less than 2 hours a week is positive (from the strategic view of teaching evaluations for a non-majors course). If they really spent more time, but did not realize it, that's educationally desirable.

6.5.2   Valuation

During week 5 of 10 in the quarter, we asked the students to take an *optional*, anonymous survey asking 3 questions:

- Has the homework helped you understand lecture?

- Has the homework helped you with labs?

- Explain how the homework as helped you or why it has not helped you.

On questions 1 and 2 students choose between **Always, Sometimes**, and **Never**. Question 3 was open ended.

There were five main points students brought up in their openended responses (all grammar and typing is original):

- **Step-by-step guidance** allowed them to understand the material easier and learn **skills and techniques**.

    Sometimes it is not clear which examples will help us with the lab, since the instructions depend on us to understand exactly how to utilize

what we learned through the homework and lectures. Most times it helps because it teaches us techniques and skills.

- Gave them **prior-knowledge** and therefore confidence in lecture and lab to discuss and ask questions because they felt they knew what was going on.

  I feel like I can help myself get ahead by doing homeworks before class. I don't have to depend on my team mates to have the answers and I get to voice my opinion if I think they're wrong.

- Gave them a chance to **practice** more than once, which helped them **clarify confusions** they had the first time through.

  The hw allowed me to experiment and try out new things on my own. If there were any difficult concepts, it would stick a lot better if I struggled and learned it on my own rather just have someone tell me.

- Showed them **new ways** of accomplishing what they want to do and to be more **inventive**.

  The homework gives me an idea of how to do new things. I get to experiment with the ideas before doing the labs for them and that's good preparation.

- Pointed out **debugging mistakes** that they could make and give them a **reference** when they make similar mistakes in lab.

  Every homework assignment has helped me by explaining all the coding and understanding what the program does. It's really helpful and I advise on people to do the homework because it gives a better understanding when we have to solve our debugging mistakes.

One student in particular summed up his view of the role of exploratory homeworks in reference to the rest of the course:

Homework teaches me the basics, class teaches me the details of how it works, and lab allows me to apply what I've learned to real situations.

This student has identified that the homeworks have allowed him to gain a familiarity with the concepts, more so than passively reading the text. Since the course was taught using the PI pedagogy, students were actively learning in lecture as well. The fact that the student was able to continue learning more complex concepts during lecture reinforces that they were ready to move on having done the exploratory homework. Finally, the fact that after two homeworks and two lectures the student was able to apply what they learned in a "real situation" (lab) indicates that the student had enough experience via similar problems to be able to tackle new, complex problems on their own.

At the end of the term we asked students, retrospectively, if the times they didn't (or couldn't) take the exploratory homework seriously had an impact on their learning in class. 62% said that it did hinder their learning in some way. 11% said it absolutely hindered their learning. 51% said it sometimes did, it depended on the material. The others claimed that they could usually figure out the material in class.

Finally, we wondered if the "spirit" of the homework would infect the students. That is, we were modeling for students the kinds of activities that "real programmers" do, and encouraging them to think about and explore concepts in the way we do. We asked:

> How often did the exploratory homework prompt you to further experiment with Alice? - e.g. by writing and testing fragments of code not explicitly asked for in the homework, just to see what they would do?

24% of students said they did this all or most of the time and 34% did it "once or twice". To the extent that this informed "playing" somewhat reflects computational thinking, this is pleasing.

6.6 Discussion

The role of the exploratory homeworks was to introduce and model for students "active learning", authentic approaches by which they can explore computing concepts and constructs and therefore acculturate themselves in technical reading. We have also explored

how the components of exploratory homeworks can be assessed via the Kolb Learning Cycle - hopefully engaging students in an effective learning process. We hope both the methods and confidence (through practice) that students gained will serve them as citizens who will continue to learn and engage with new software and technologies throughout their lives.

Most notably, exploratory homeworks are a low fidelity way of introducing students to technical reading strategies employed by experts in the field. They are designed such that the students never get stuck and therefore can engage in authentic practice without an expert present (outside of lecture or lab). Students are taught to read technical material with their programming environment open, this is typical with experts in computing, but is discipline specific. Students learning physics do not typically read a section on gravity and then throw a watermelon off the roof to test the theories in the text. As described above, programming involves a very unique experience in that students can interact with the computer directly, getting feedback on behaviors and concepts they are learning. Exploratory homeworks engage students in cognitive apprenticeship without requiring in-person expert guidance.

6.7 Future Work

Having developed the structure of the exploratory homeworks and seen indications of students improved textbook reading habits and increased learning, we plan to develop more exploratory homeworks for CS1 courses. During this second phase we will explore the effectiveness of using the Kolb Learning Cycle as we do now (not necessarily in order) against a more rigid structure.

We also plan to observe some students completing the exploratory homeworks and ask them questions about their normal study habits as they compare to using exploratory homeworks. Finally, we plan to follow up with students who have been working through exploratory homeworks and see if they have transferred the textbook reading skills outside of the CS0 course.

We are eager to have a discussion with the community on how we might improve exploratory homeworks, what other educators would need to support them in creation of their own exploratory homeworks and advice they might have on measuring the effectiveness of exploratory homeworks when they are often coupled with effective pedagogies in lecture and tutorials in labs.

6.8   Conclusion

Exploratory homeworks are tools that encourage students to actively engage in pre-lecture reading; this includes writing and debugging code as well as learning how to read technical, static texts. Active learning is a technique that is often successfully used in lecture and lab based settings, but rarely are students asked to actively learn on their own, before lecture. Exploratory homeworks can enhance already active textbooks, like the Alice textbook used here, as well as more expository textbooks. The biggest benefit from students actively reading before lecture is they are able to come to lecture prepared to learn more complex concepts and constructs enabling instructors to delve more deeply in lecture. Students are also developing authentic computing habits by actively participating in trial and error, predictions and reflections when learning a new programming language. We encourage the community to create exploratory homeworks for CS0 and CS1 courses using a variety of textbooks and report back on both effectiveness and challenges.

6.9   Acknowledgments

Chapter 7

"Sparking the Fire" in Children to Engage in Computer Science

Chapters 3 through 6 show effective ways of enculturating undergraduate students in a large university classroom, the following chapters will focus on how to similarly enculturate *children* through a video game. Specifically, this chapter categorizes expert programmers' "origin stories" and designs a video game to engage children in similar experiences.

7.1 Designing Authentic Learning Experiences Based on Origin Stories

That minds are "fires to be kindled rather than vessels to be filled" was first expressed by Plutarch and has become one of the most oft employed quotations within pedagogical discourse. Still, there would appear to be some magic and mystery to the sparking of flames: it is much easier to develop curricula and textbooks that seek to fill up minds with material rather than to spark a love of learning. Not surprisingly, it is a rare student indeed that catches fire in a classroom. Furthermore, our "factory" model of education has come under such heavy criticism in recent years, [105, 77, 97] that one might begin to suspect that some forms of institutional instruction may have a dampening effect on the sparking of flames. This suspicion (further supported anecdotally by several decades of the authors'

personal experiences in academia), prompted us to study the nature of flame-sparking by investigating how such flames manifest themselves "in the wild" - i.e. outside of classrooms.

Indeed, the field of computer science is rife with stories of children (many strikingly young) who discovered computers and taught themselves to program without the benefit of formal instruction. We collected various such accounts from a variety of sources. The following section gives a grounded-theoretical analysis of these accounts and begins to set forth a list of five qualities that appear to correlate with the eventual sparking of a lifelong love of programming.

We then sought to recreate these five qualities in a laboratory study with 40 girls, ages 10 to 12, exposing them to programming for the first time using a video game called *CodeSpells*, which supports many of the aforementioned qualities by default. Our goal was to study these qualities "up close", in order to refine our understanding of them. We present the results of this second study in Section 7.4. Ultimately, using the insights gained from our groundedtheoretical analysis and our laboratory study, we are able to further illuminate how a passion for programming spark "in the wild". Although our studies are both of non-institutional learning environments, we believe (as in [132]) that the study of these environments can help shed new light on the process of learning in general. Thus, in Section 7.5, we discuss various contemporary software tools for teaching programming - looking at them in the new light offered by our analysis of non-institutional learning.

7.2 *Formal* vs. *Information* Learning

In related literature, a distinction is generally made between "formal" and "informal" learning. However, we prefer the terms "institutional" and "non-institutional" because, as pointed out by Sefton-Green [132], the terms "formal" and "informal" are somewhat over-loaded. "Formal" can refer either to "organized" (as opposed to "disorganized") knowledge acquisition or to "institutional" (as opposed to "non-institutional") settings. This can lead to

confusions: for example, one could imagine a disorganized (or student-driven) acquisition of material within the confines of an academic classroom - which is difficult to classify as a "formal" or "informal" experience. Thus, we use "institutional" (and "non-institutional") to make clear that we are discussing settings that fall within (or out side of) traditional academic institutions.

There has been wide research on informal learning spaces, but relatively little that pertains directly to non-institutional *computer science education*. The only two studies that we know of both involve ethnographic research on how Scratch (a novice programming environment) has been employed voluntarily by minority teenagers in a Los Angeles-based Computer Clubhouse [114, 102]. Although, the clubhouse environment is technically an institution, we classify the setting as non-institutional because it is not a traditional academic classroom environment. Students have free rein over their activities and can even play video games if they wish.

Thus, this paper adds to the small amount of prior knowledge on non-institutionalized computer science education by performing two new studies, using two new methodologies, and offering novel analysis of non-institutional learning to better illuminate the nature of learning in general.

7.3   Origin Stories Study

Since our research interest lies in the notion of sparking flames that lead to *lifelong* passions, we elected not to begin by studying existing non-institutional learning spaces - e.g. the clubhouse-like settings mentioned above, or after-school programs. After all, the children partaking in these spaces are (at best) only just *beginning* a lifelong computer science career. Instead, we collected firsthand accounts from contemporary computer science professionals, whose lifelong passions began in non-institutional settings. In this way, we hoped to be better equipped to draw conclusions about the non-institutional learning qualities that correlate

with long-term excitement toward programming.

Ultimately, we accumulated 30 of these "autobiographical origin stories" (as we call them) from 3 different sources. We first drew 12 such stories from: 1) the Computing Educators Oral History Project [32], and 2) a qualitative study performed by Hewner and Guzdial [74]. To obtain more data, we also obtained 18 origin stories from 3) a survey we conducted ourselves, wherein we instructed participants to respond to the simple prompt: "Describe your first experiences with programming." We sent out this survey to the faculty and graduate student population at our university as well as to industry professionals via email and LinkedIn. In all cases, we selected only first-hand accounts where it was clear that the participants were describing their first experience as a programmer within a non-institutional setting.

The demographics of the autobiographers were intentionally quite diverse: ranging in age from graduating college seniors to industry professionals to retired. There were 11 males and 14 females. The genders of the 5 authors acquired from the study performed by Hewner and Guzdial is not directly reported. Some accounts involved programming on ancient Teletype machines, whereas others used modern hardware and environments like Alice. Since our intention was to begin constructing a very general theory of effective non-institutional learning environments, we felt that diversity was key.

7.3.1 Methodology for Origin Stories Study

Following the well-accepted procedures of grounded theory as set forth by Strauss and Corbin [145], we first engaged in open coding, where we coded each sentence and often time partial sentences. In the open coding, we found 13 subcategories: selfdrive, enjoyable/emotional connection, "flow state" while programming, confidence/belief that you can succeed, investment in the results of the code, empowerment, creation of "meaningful" artifacts, lots of hours/re-visitation, access to immediate feedback, wrongness is on a spectrum and is therefore not binary, access to support, feeling addicted, losing track of

Table 7.1. Results of Grounded Theory coding of Origin Stories

Axial Coding Results	Open Coding Results
Learner-Structured Activities	Self-driven, Access to immediate feedback, Acces to support
Exploration/Creativity /Play	Creation of "meaningful" artifacts, Investment in the results of the code
Programming as Empowerment	Enjoyable/emotional connection, confidence/belief that you can succeed, wrongness is on a spectrum; not binary
Difficulty Stopping	"Flow-state" while programming, Feeling addicted
"Countless" Hours	Lots of hours/re-visitation, Losing track of time

time.

We followed open coding with axial coding [145], which allowed us to derive 5 distinct categories based on the open coding. Table 7.1 shows how the open coding results were connected to the 5 categories found in the axial coding and in the next section we define and give examples to our 5 categories.

7.3.2 Results of Origin Stories Study

All 30 origin stories exhibited at least 1 of the 5 qualities. 20 of the origin stories exhibited 3 or more.

Learner-Structured Activities: An activity is learner-structured when the learner engages in activities that are not at all (or not entirely) structured by some outside force:

- "...when the Commodore PET came along and **I took it home one summer and taught myself BASIC**",

- "I was **by myself** with no assignments - **just me and the computer** to play with."

- "I... bought books and wrote programs instead of taking courses in high school."

Exploration / Creativity / Play: The act of creation through experimentation and play was common - a mentality of "What would happen if...?"

- "I'd also **try to add things**, so if it was print a 'hello world' string, I would make it ask for your name, and print 'hello ¡name¿', and if your name was a friend's name, **I'd add an inside joke or something**."

- "I would **purposely try changing some of the parameters to see what would happen**"

- "And we had gotten a Radio Shack computer and I had **played around** with that."

Programming as Empowerment: Programming was commonly viewed as a means of increasing one's personal efficacy, selfesteem, sense of purpose, or social standing.

- "... so I was able to do it, and that **gave me confidence** that **'I knew computers'**"

- "To me, **computers are freedom**, they are entertainment and above all they are a **symbol of power and adulthood.**"

- "Soon enough the computer was asking me what my name was and then asking me how I was and addressing me by name. **I'd made a new friend!** I think my most impressive program was when I got the computer to make different sounds."

Difficulty Stopping: We expected to discover that the autobiographers found programming enjoyable, but the origin stories included many accounts of how frustrating the process was. The common thread, however, is that the act of programming was perceived as an engaging and often addictive one - with descriptions liked being "hooked".

- "That was a moment where **I got hooked** because just the sense that you could program a loop that could do an array of any number of any size, any number of numbers, just seemed to be such a fantastic thing."

- "When that actually worked, I thought it was the coolest thing ever, and after that **I was hooked** =P"

- "[I was] becoming **so closely involved** with technology... [that] I was a little afraid of **being sucked into** the CS major stereotype..."

- "I went for Mechanical Engineering. But **my liking for computers never died**... and I took some private computer courses on the side"

"Countless" Hours: This is closely related, but distinct from, difficulty stopping. (Some autobiographers did not exhibit signs of addiction but still invested large amounts of time for other reasons.) Many origin stories describe spending so much time that the autobiographers could not readily quantify the amount, tending to use abstract generalizations like "countless" or "dozens" or metaphors like "journey" (which imply a large timeinvestment).

- "I wrote **countless Pascal programs** on my old PC. (I still have some of the old code, printed out, somewhere!)"

- **"My journey** has been full of excitements. **Since the beginning** I have enjoyed computers."

- "So back then it was really **very labor intensive**, was **not instantaneous by any stretch of the imagination..**"

- "I read the guide to my TI-89 calculator a **dozen times**, learning to make simple games or programs in it."

We stress that this study was intended to be a preliminary theorybuilding endeavor, not a means by which a definitive, overarching theory was to be achieved. Indeed, the main intent was to inform and structure the laboratory study that followed it.

7.4   Lab Study

Our laboratory study was intended to "enhance theoretical sensitivity" [145] through the use of questioning - moving a step beyond the brief, straightforward nature of the origin stories.

We wanted to be able to determine if 1) the above five qualities could be recreated under laboratory conditions, and if so 2) to study them "up close" in order to enrich our theoretical understanding of them. Furthermore, we contend that performing laboratory studies on non-institutional learning is extremely important if one's ultimate goal is to learn from non-institutional learning in order to help inform institutional learning environments. After all, a laboratory lies somewhere between a fully institutional setting and a fully non-institutional setting. This means that lessons learned in a laboratory may be more easily transferable to a classroom than lessons learned in the wild.

7.4.1   Methodology for Lab Study

Software and Experimental Setup

CodeSpells is an educational video game - more specifically, an immersive fantasy role playing game designed to teach Java programming by immersing the player inside a 3D virtual world and a first-person storyline wherein she plays the part of an apprentice wizard [26]. *CodeSpells* teaches Java by giving players access to a novice-friendly API for crafting novel magic spells.

We chose to use *CodeSpells* because its magic metaphor has been shown to be compelling and exciting to novice learners [ibid]. We felt that the game would be an engaging environment for girls in the 10- to 12-year-old age group.

Normally, learning in *CodeSpells* is encouraged by way of a series of quests that must be completed with the use of Java-based spell crafting. However, we particularly wished to study **learner-structured activities** and **creative exploration**. So we employed a version

of *CodeSpells* that did not have explicit quests to complete. (The *CodeSpells* platform is quite extensible). Rather, players could walk up to in-game gnome-like characters who would give various spells to the player, along with simple explanations. Our hope was that these spells would serve as starting points for code exploration.

We recruited forty girls (ages 10 to 12) who had no prior programming experience in any language or programming environment. We gave them a short overview of the *CodeSpells* game mechanics - including how to write and edit code with the in-game IDE. We divided them into 12 groups of three and 2 groups of two, and encouraged them to explore the 3D world and to see if they could "do interesting things". We were purposefully vague, as we hoped to encourage a largely unstructured learning environment.

We then allowed the subjects to play *CodeSpells* for one hour while we observed. Data collection techniques during this time involved video and audio recordings of 6 groups; video was of the computer monitor so that we could record actual gameplay. We also assigned one undergraduate computer science major to each group; each undergraduate took notes on what the girls struggled with. Our main research team also took observational notes as they walked around the lab.

After one hour, we split the girls into groups of 12 to 15 members and engaged each group in a semi-structured group interview. The interview involved questions such as: "Describe what you have been doing for the past hour.", "Can anyone share something interesting they did in *CodeSpells*?", "Can anyone share something they were trying to do, what did you do to try to make it happen?", "Did anyone get stuck while playing, what did you do when you got stuck?", "If you had more time, what would you want to do next?", and "Can anyone describe to me what spells are and how you use them?". Data collection during the group interview was videotaped and the interviewer took brief notes.

7.4.2 Results of Lab Study

Our subjects played *CodeSpells* for the entire hour before we had to ask them to stop. Students expressed disappointment that it was "over so soon". 25 of the subjects showed interest in playing *CodeSpells* at home and wanted to know when it would be available for them to play. We consider this to be evidence that students experienced some **difficulty stopping**. And the lingering excitement is a rough indicator that novices might indeed play such a game for **"countless" hours** - though we would need to allow our users to play *CodeSpells* at home to measure this further.

Of the 6 groups of girls we recorded, roughly 90% of their time was spent exploring the 3D world and/or editing code (as opposed to, say, chatting amongst themselves). Also interesting is that the quest-less version of *CodeSpells* provides considerably less structure than most video games. In spite of this, subjects did not ask "What am I supposed to be doing?", nor did they seem at a loss for activities to engage in. The simple directive to "do interesting things" was sufficient for inspiring subjects to give structure and shape to their own activities. This strongly suggests that a **learner-structured mentality** arose within the lab study.

Some subjects tried changing method calls like thing.levitate(3) to thing.hop() or to thing.blowup() (which are not available in the API but are surprisingly correct syntactically). Even though these attempts failed to evoke the desired effect, students did not appear to become discouraged. Subjects also discovered quite valid changes in this way too - e.g. thing.levitate(300000). This suggests a drive to **explore, play, and create**. And furthermore, this drive appeared to be fueled (rather than dampened) by syntax errors. Of the 6 groups video taped, 4 of them encountered some syntax error that they resolved either by undoing the error they introduced (55% of the time) or asking the undergraduate student that was near them (45% of the time). In all cases, acts of **self-structured activities** followed these interludes.

A particularly interesting phenomenon occurred with regard to what one might call "logic errors". One group of girls made the mistake of levitating an object so high into the air that it could not be reached. They were able to retrieve the object, however, by jumping onto *another* object and levitating it (and thus themselves) sufficiently high enough to reach the original object. This emergent use of code to surmount challenges of one's own making is an act that fits our definition of **exploratory play**. Logic errors were seen in all 6 groups recorded, but we note again that the exploration seemed to be *fueled* (rather than dampened) by things going awry.

During the lab study and more so in the group interviews we were encouraged to hear that the girls felt **empowered**. When changes to the code didn't accomplish what they wanted, they kept working towards their goal, trying different spells or different code changes until they eventually reached it. When asked about how they reached their goals, they conveyed that they "knew it could be done" and that they "just needed to figure out how". They described code as a "way to accomplish anything" within the 3D environment. At no point did they describe programming as a barrier to the **self-structured activities** and **creative exploration** in which they chose to engage.

7.5   Discussion

7.5.1   Refining the Theory

With the benefit of the laboratory study, we were able to study 4 of the aforementioned qualities (with the exception of **"countless" hours**) up close. Our observations allow us to add a quality that seems to go hand in hand with **exploratory play**. Namely, **A positive attitude toward "failure"**: An institutional setting often provides an objective and authoritarian definition of what "success" and "failure" are. It was striking to observe that, when left to their own devices in a *non*-institutional setting, subjects approached what would traditionally be labeled as "failures" (syntax and logic errors) without the slightest

apparent awareness that they had done something "wrong". No one had ever told them that syntax errors were bad or that code ought to do what one expects it to.

Thus far, we have presented two studies, 1) a study of origin stories in computer science and 2) a laboratory study of 40 novice 10- to 12-year-old pre-programmers. In the first study, we were able to tentatively identify 5 qualities that correlate with noninstitutional learning in such a way that leads to the sparking of a lifelong passion for computer science. In the second study, we showed that at least 4 of these qualities (all but **"countless" hours**) could be recreated in a laboratory setting. We further observed a striking relationship between a **positive attitude toward "failure"** and the tendency to engage in **exploratory play** - allowing us to further enrich our theoretical framework. We believe that further study of non-institutional learning is critical if we, as educators, wish to better understand how the sparking of young minds happens in the wild. We feel that a clearer understanding of the conditions under which such a seemingly mysterious phenomenon naturally occurs can serve to shed light on how to recreate this phenomenon more readily within institutional settings. Moreover, though, the theory we have developed so far can help enrich modern pedagogical discourses - for example, the ongoing discourse about tools for teaching novice programmers

7.5.2   Applying the Theory

Even though a comprehensive theory of non-institutional learning is perhaps a long way off, the aforementioned theoretical framework is already sufficient to allow for existing tools to be discussed with a new vocabulary, leading to new insights. Consider, for example, some common tools for teaching introductory programming.

Novice IDEs

Scratch is a visual programming environment designed to allow novice programmers, particularly young programmers, to create media-rich results by dragging programming

blocks into place to create programs [101]. The 3D *Alice* environment is a friendly IDE serving as a "stepping stone to computer science careers" [34]. *Alice* provides an experience that allows users to make their own movies or video games.

Such environments allow the production of visually stimulating effects - which certainly gives young pre-programmers more avenues for **exploration, play, and creativity**.

On the other hand, it is not clear whether such environments encourage the other four qualities. For example, these environments do not *overtly* seek to motivate **learner-structured activities**. Likewise, there is no overt effort to induce **"countless" hours** of practice. On the other hand, nothing in Alice or Scratch overtly *prevents* learners from spending **"countless" hours** on **learner-structured activities**. Then again, nothing overtly *prevents* a student from opening up Eclipse and playing around with Java for hours on end. It is no doubt a rare occurrence though.

When the above are embedded within a classroom environment, **"countless" hours** can easily be required of students - but at the risk of sacrificing **learner-structured activities** and **exploratory play**. A natural place to look for solutions to this problem is in educational video games.

Educational Video Games

There exists a long history of educational video games, some of which are intended to teach programming [48, 86, 4, 123]. Although programming-related educational games to date have not gained widespread popularity, one can still examine video games (in general) in light of the framework.

Learner-Structured Activities: Video gaming tends to be a selfdriven and self-structured activity - even for individuals who are too young to apply the same kind of self-motivation to academic subjects.

Exploration / Play / Creativity: Games can be used to create rich 3D worlds that facilitate exploration. In *CodeSpells*, for example, the interplay between objects, physical

laws, and magic makes for emergent properties that drive play and exploration.

Programming as Empowerment: In *CodeSpells*, for example, the code a player can write directly correlates with the efficacy of that player within the 3D world. What one can code and what one is empowered to *do* go hand-in-hand.

Positive attitude toward "failure": Games are not always enjoyable. Players can lose. Characters can die. Tetris blocks don't always fit. Frustration is as common as euphoria. Yet it doesn't seem to phase gamers and is indeed an integral part of the experience.

Difficulty Stopping: Indeed, while not always strictly an enjoyable experience, video games inspire an unprecedented level of active engagement that (for some) borders on addiction.

"Countless" Hours: The gaming industry is a multi-billion dollar industry. At least one study shows that children spend 10,000 hours playing video games throughout childhood [122]. Clearly non-educational video games *do* spark lifelong passions... for playing video games.

One might then ask, if video games have all the requisite qualities, why aren't educational video games routinely sparking lifelong passions for all sorts of academic disciplines? Our response would be to point out that many explicitly *educational* video games actually lack some of the above qualities.

It is difficult for many educational games to support **creative exploration**. Consider, for example, the classic *Math Blaster*. In order to provide positive and negative feedback to the player, the game must be able check the player's answers to mathematical problems. This means that the game can only serve-up a series of checkable problems, each with an objective right or wrong answer. There is no room for creativity. The game cannot say to the player: "Do something mathematically creative. You'll get more points the more creative it is."

Similarly, many educational games have been criticized and called "chocolate-

covered broccoli" [20] because the educational material interrupts what would otherwise be smooth gameplay. In other words, the education interrupts the fun stuff. Or, to put it the other way around, the fun stuff (chocolate) is intended to thinly disguise the education (broccoli). No doubt this has a detrimental effect on the **difficulty stopping** and the tendency to engage in **"countless" hours**.

In conclusion, it would appear that both novice-friendly IDEs and educational games alike leave something to be desired from the standpoint of sparking lifelong passions for programming. The gamification of programing environments like Alice or Scratch might be one solution. *CodeSpells*, for example, seeks to combine the best aspects of educational games with the best aspects of novice-friendly IDEs.

7.6   Future Work

Our main goal is to use lessons learned from highly successful non-institutional learning to shape and inform institutional learning. To this end, we are currently conducting a 6-week-long class in which unstructured play time in *CodeSpells* is supplemented with in-class instruction. Also, we have developed a multi-player, competitive version of *CodeSpells*, which we are utilizing to examine the non-institutional learning environment that has sprung up around a competitive *CodeSpells* team that practices three times a week while we observe.

Our grounded theory study on origin stories was somewhat limited because we reviewed static, written text and were not engaging in interviews. We would like to further define our 6 categories that classify non-institutionalized learning spaces through a series of interviews where we could employ the grounded theory technique more rigorously. This study has introduced us to the theory, but we would like to define this theory more detailed so that other education researchers may apply it when examining and creating tools and environments for novice programmers.

7.7   Conclusion

We take the position that it is highly relevant to study non-institutional learning in which young people teach themselves to program. We analyzed the origin stories of 30 individuals who eventually became successful in computer science. We identified five qualities that tended to occur across multiple origin stories and that can be tentatively posited to correlate with the sparking of lifelong passions for computer science.

To more deeply study these qualities, we performed a laboratory study with 40 girls (ages 10 to 12) and analyzed their experiences according to the aforementioned five qualities. This allowed us to refine our understanding of these qualities and even to observe a sizth quality - further deepening the theoretical framework.

Our contributions are three-fold:

- We give the first theoretical framework for understanding the conditions under which the sparks of lifelong learning are sparked in non-institutional computer science learning environments.

- We demonstrate two novel methodologies for further examining non-institutional computer science learning. (The small amount of prior work that exists uses only *in situ* ethnographic methodologies.)

- We contribute a new vocabulary for discussing and designing tools for novices, using our theoretical framework to point out opportunityes for improvement in both novice-friendly IDEs and educational games.

Ultimately, we take the position put forth by Maloney et al. - namely, that studying non-institutional learning is a means by which to understand learning in general [101]. After all, non-institutional learning has a unique character. It is unstructured. It is creative. It is play. Also, as we saw in our laboratory study, the absence of values implicitly instilled in

institutional settings (e.g. that syntax errors are bad) can lead to differences in behavior. As such, the study of non-institutional spaces may serve as an untapped resource for computer science educators a resource that can help inform how we structure our institutional spaces. We believe the study of these space can yield surprising new insights about what were doing right as well as what we may be doing wrong. Ultimately, it is a line of research that can give us new ways to light fires and to keep them lit.

7.8   Acknowledgments

Chapter 8

Embodying the Metaphor of Wizardry to Support Enculturation

Chapter 7 introduced the authentic design of CodeSpells based on real experiences of expert programmers. In this chapter, we study the effects of *wizardry and magic* on the enculturation of children. Though CodeSpells was designed to engage children in authentic experiences, we show that we leverage that it is a video game to further enhance that experience through the embodiment of the metaphor.

8.1   Embodiment in Novice Programming Tools

Novices have often struggled with learning to program, often times due to their inability to connect abstract programs with their effects on the environment. Novices can also struggle with low self-efficacy due to their dis-immersion with their programming environment. In this paper we present features of CodeSpells [55] that make this environment particularly immersive and therefore an environment where novice programmers are encouraged to try hard and not to become discouraged.

Though there has been much improvement in novice programming environments, with visual programming such as Alice or Scratch [46, 101], gamified environments such as Kahn Academy and Codecademy, and applied environments such as Media Computation [68], we believe that none has immersed the novice into an environment that not only

encourages a deep understanding of the concepts, but also to instills a sense of determination.

In the next section we discuss how other languages and environments compare with CodeSpells. We then go deep into the API of CodeSpells, discussing the immersive and applied properties that we believe create a unique novice experience. We then present findings from an exploratory study and discuss the implications and future work.

8.2 Related Work

8.2.1 Programming Languages

Languages such as Pascal and BASIC were early efforts to make programming concepts easier for novices [83] by sacrificing some degree of authenticity. Today, tools such as Alice [46], Scratch [101], and Lego Mindstorms [90] take this trend to its logical conclusion, providing a syntax-free programming interface that involves dragging and dropping blocks of a program into their appropriate places. Similarly, the Toque [147] and Tern [76] systems allow programs to be constructed by acting out cooking actions (in front of a Wii) and by assembling physical blocks.

Yet, there has been a counter-trend toward teaching industrystandard languages in academia, at least to computer science majors. This has resulted in an Optimal Novice Language Debate over which industry-standard language strikes the best balance between palatability and authenticity [47, 11, 78, 103, 142, 75].

CodeSpells avoids these either/or tradeoffs. It relies significantly on rich gameplay to inspire players to engage with challenging programming concepts and syntax in ways they might never voluntarily do otherwise. Also, the implementation of CodeSpells is factored into both game engine and programming engine, enabling the use of whatever programming language is preferred.

One of the design goals for CodeSpells was to immerse novice programmers in the effects of programs, specifically by focusing on video game features. Guzdial has

penetrated the CS1 curriculum with his Media Programming pedagogy [68], which adds novice-centered libraries to Java and Python in order to assist programmatic manipulations of photographs, audio clips, and other rich media. Although students are not necessarily immersed with Media Computation, they are introduced to the complex concepts through an approachable (known) application: photo manipulation. Even if students do not have experience with programs such as Photoshop, they understand the basic effects of photography (e.g. greyscale, green-screen effect).

On the other hand, environments such as Scratch [101], Alice [46], the Logo family [95], and Lego Mindstorms [90] engage the user by providing a scene that can be watched and explored. This allows students to fully control the environment they are effecting and therefore better understand implications of changes they make in the code. Though these environments offer a scene where students can watch coding effects in real time, they tend to sacrifice authenticity for this "immersion". CodeSpells, on the other hand, increases engagement and therefore authenticity.

8.2.2 Context of Programming

By giving the novice programmer a new domain in which to work, programs can take on a new significance, enhancing the learner's motivation and persistence. Recent enhancements to the "How to Design Programs" methodology [59] provide a scheme environment for programming 2D animations. As programming environments, both require external learning support and lack the motivational hooks provided by a first-person video game.

Logo programs allow the user to mobilize an on-screen "turtle"; Alice and Scratch programs allow the user to create 3D and 2D animations (respectively); and Lego Mindstorms programs allow users to affect microcontrollers embedded in a Lego apparatus.

The completeness of CodeSpells' contextualizing metaphor means that the game requires neither supplemental lectures nor textbooks, like the systems above.

All learning resources are seamlessly interwoven into gameplay, making the game into a self-contained learning environment that can be used even where formal instruction is unavailable. In addition, the storyline and 3D environment provide added engagement to sustain motivation, say, in the absence of an external motivator. This significantly increases the sense of immersion amongst students, not ever being taken away from the programming environment to "learn a new concept" and instead finding them throughout gameplay.

The quests provide novices with pedagogically relevant tasks. Coding constructs are introduced via the spellbook. Reasons to persist come from the first-person storyline, badges, and visually stimulating 3D environment.

Likewise, the CodeSpells approach is also distinct from the many popular game building environments, many of which are used in classrooms: e.g. Greenfoot [86], NetLogo [48], Unreal Scripting [52], the RPG creator [126], the Starcraft map editor, Kodu [100], the Spring RTS Engine [143], and ToonTalk [81]. CodeSpells is an RPG (Role Playing Game) that more fully immerses the students in the gameplay, but also in the learning. Most games used in the classroom require the students to be the "builders" or "engineers", but not actually be affected by the code. We wanted to focus on the students "feeling" the effects of their modifications and programs, not just "witnessing" them.

8.3   A New Programming Context

8.3.1   An Immersive Video Game

The Metaphor

Magic. Magic is a metaphor that crosses both programming and fantasy RPGs. In programming, what an expert programmer can do is often referred to as "magical". As Brooks states "The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be" [18]. It can then be considered that expert programmers have

a suite of "magical spells" that they can use at any time to solve various problems. These spells might be small programs they have written, methods, or simply concepts they now understand (e.g. the "Iterate through an array spell").

Similarly, in most fantasy RPGs, players often have magic spells that they can use at any time to solve various problems. Comparable to the "spells" of expert programmers, RPG players may start with simple spells, and have to earn and learn how to use more complex ones. CodeSpells embodies the idea of "spells" as both an artifact that players can see, read and write and also an executable object that players can use to interact with the world and solve problems.

Figure 8.1 is an example of the spellbook provided to players as an in-game resource for game-play and learning. To fully immerse the players in CodeSpells we designed a similar resource that they might find in any other fantasy RPG, a spellbook. This spellbook provides them with information about how spells work and when they might be useful. Players can use the spellbook, along with the Non-Player Character's advice, to navigate through quests.

Most importantly though is the spellbooks dual purpose as a learning resource. Players are encouraged to read through the spellbook, but also to execute spells to test them out. Being in a non-competitive environment allows them to take their time in fully understanding the effects and limitations of each spell. Modifications can easily be made in the in-game IDE shown in Figure 8.2. This particular resource is the gate to programming within CodeSpells and is essential to immersion and embodiment.

8.3.2 Designing for Embodiment

In this section, will explore the API of CodeSpells, paying particular attention to how we designed classes and methods to encourage players to understand the effects of spells (programs) from a first person perspective.
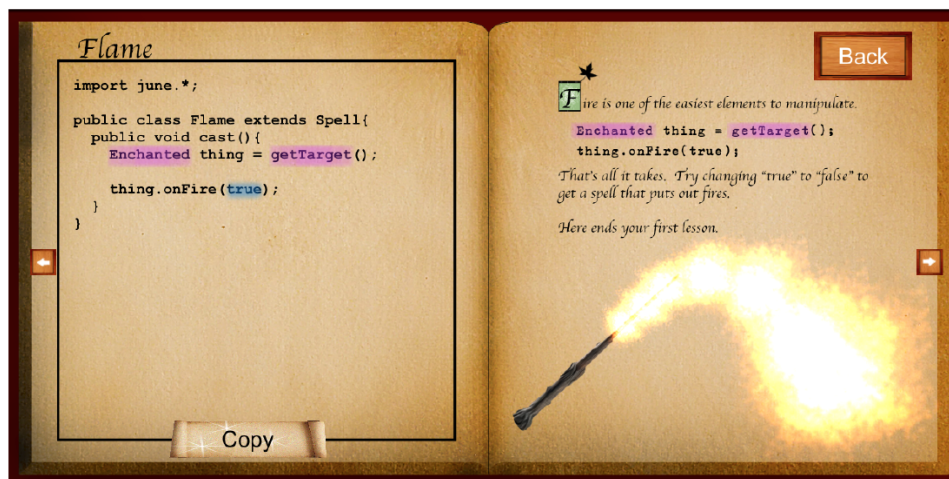
Figure 8.1. Spellbook for in-game resource (solving challenges and quests) and learning (understanding Java concepts).
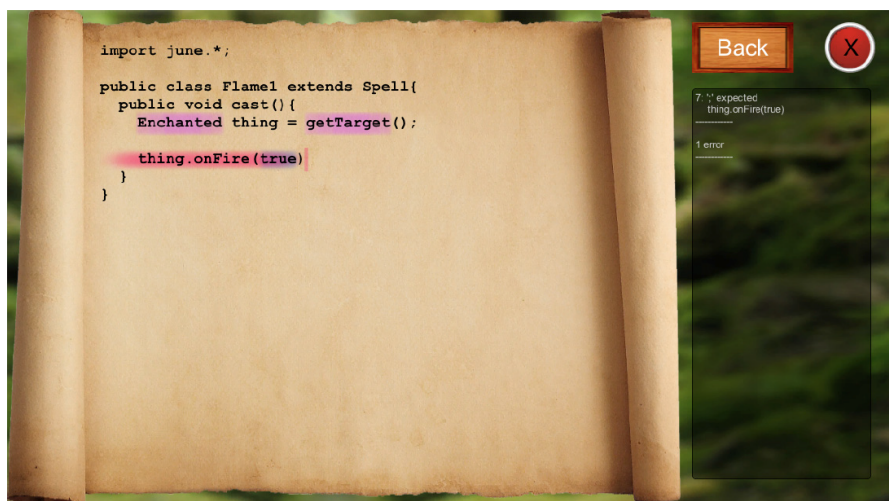
Figure 8.2. In-Game IDE that allows players to make modifications to the code very easily to test effects and limitations of spells.

An Innovative API

We designed our API to be similar to that of Java web applets where Applet is subclassed and the run() method is implemented, by having a superclass Spell and subclasses that implement the cast() method.

The first spell that is provided to the player in their spellbook is the Flame spell. The player casts this spell onto an item that is flammable, and then the item is set on fire.

```java
public class Flame extends Spell{
  public void cast(){
    Enchanted thing = getTarget();
    thing.onFire(true);
  }
}
```

Our first challenge was making the spells powerful enough such that players could interact with the environment relatively easy, but authentic enough such that players could engage with the typical concepts presented in a CS1 course. We studied the balance of power and authenticity in Guzdial's Media Computation and within Alice and Scratch and decided on a threshold close to reality, allowing players to easily accomplish tasks that are easily done in the real world (setting something on fire, moving forward), but having difficult tasks be represented as complex spells that combine the simple API calls and typical CS1 level concepts such as loops, conditionals and parameters.

In addition to making complex magic difficult to attain, we wanted other, intuitively mundane tasks, to be easy. Our rule of thumb was that if it a task is typically performed without much conscious thought by someone in the real world, then it should require a minimal amount of code in the fantasy world. One such intuitive task is the ability to specify and talk about things in the world around us. We do this with various combinations of

pointing, gesturing, and naming – all of which can be done with little conscious thought. We felt that if such intuitive tasks required a non-trivial amount of code to be written then players would quickly become frustrated at the API's lack of expressiveness.

We tested the limitations of our API with 3 intermediate programmers (undergraduate students in the computer science major) making as many complex spells as they could in 3 months. We observed them during this process, improving upon the API when they struggled due to a minimal API. Though further refinement might be done once a larger scale study of CodeSpells and it's learning effects is complete, we are confident that the level of specificity in our spells will promote the understanding of typical CS1 learning goals.

Two key outcomes of this design process are intuitive mechanisms for (a) expressing entity reference and (b) specifying spatial manipulations. As will be seen, the basic design rule that emerged was to make the API work in ways that are analogous to the way humans think and work when performing everyday tasks. That is, not only should the interface make mundane tasks easy, but the interface should be easy to understand by analogy to everyday experience. As a result, there is a degree of embodiment designed into many of the operations, via reference to the avatar. In the next section we will explore how embodiment played a role in the effects of CodeSpells on novice programmers.

8.4   Embodiment

8.4.1   Referencing Objects

Entity Reference

Obtaining a Java reference to a particular entity in the 3D world is something we wanted to make as easy as possible. We handled the trivial case using the aforementioned getTarget() function to reference any individual entity that the apprentice wizard is able to see and point to – i.e. entities in close proximity to the player's embodied location in the virtual 3D world.

However, during the course of working with our intermediate programmers, it quickly became necessary to refer to more than one game entity, for example to move one entity to the location of another entity. Thus, we added the ability to reference certain special entities by name. For example, the player may obtain a reference to their own in-game avatar by writing Enchanted e = getByName("Me"). To make this practical, we gave permanent names to certain fixtures in our environment – e.g., "Me" and "Rain", which the wizard can learn by reading her spellbook. The following spell can be used to make the local rain cloud named "Rain" follow the player around:

```
public class SummonRain extends Spell{
  public void cast(){
    Enchanted rain = getByName("Rain");
    Enchanted myself = getByName("Me");
    while(true){
      Direction toward_me =
      Direction.between(rain,myself);
      rain.move(toward_me, 5);
    }
  }
}
```

Not all entities are so special, of course. If a player wants to refer to a particular rock in a spell, she can give the rock a name and then later use getByName() to obtain a reference to the rock.

Referencing Groups of Entities

When building a bridge out of 20 rocks, however, our interns found it tedious to give a name to every rock and to declare 20 variables of type Enchanted. We handled this by

adding API support for defining variably-sized lists of entities.

Our first attempt was to provide support for querying a radius around an entity (identified by getTarget()) using a getWithinRadius(float) method. However, it was too easy to accidentally obtain too many rocks, too few, etc. The root of the problem is that a radius of size X is difficult to visualize before casting one's spell.

The solution to this problem was to bring the senses into play. We designed a magic staff that can be used to designate a static area of effect. Putting the staff down on the ground *displays* a radius within which all entities are selected, and a textual name for the area is displayed above it. The area's radius can be resized to suit the player's needs. Supposing a staff has been used to create an area, and it has been renamed to "Fire Trap Area'", the following spell lays a permanent trap for one's enemies:

```java
public class FireTrap extends Spell{
  public void cast(){
    Enchanted area = getByName("Fire Trap Area");
    while(true) {
      EnchantedList list = area.getWithin();
      for(Enchanted e : list)
        e.onFire(true);
    }
  }
}
```

Vector Math

Another challenge was to allow novices to manipulate entities in 3D space without expressing their ideas in the language of linear algebra, trigonometry, or geometry. That is, our API had to provide intuitive ways of performing spatial manipulations without using mathematical syntax. After all, human beings are able to perform a wide variety of

spatial tasks (driving, building with blocks, throwing balls, etc.) without formal math skills. Because CodeSpells is a first-person game, we were able to simplify the API by leveraging the player's existing embodied intuition. That is: the ability to easily move and rotate one's avatar, and to write code that is dependent on the avatar's location and rotation, gives the player a surprising amount of expressivity without the need for higher-order mathematics.

In CodeSpells there are 4 avatar-referenced directions – forward, backward, left, and right and 6 world-referenced directions – up, down, north, south, east, and west. The following spell uses the "forward" direction to allow the player to fly when standing on top of the spell's target entity (e.g., a magic carpet). Because "forward" is dependent on the direction the player is currently looking, the behavior of the spell is affected at runtime by the player's movements and head rotations, flying in whichever direction the player's embodied avatar is currently looking.

```
public class Flight extends Spell{
  public void cast(){
    Enchanted target = getTarget();
    while(true){
      target.move(Direction.forward(), 0.2);
    }
  }
}
```

The flight spell is a good illustration of our general approach: Just as human beings are trivially able to push physical objects along their "forward vector'" in the real world – we wanted players to be able to easily employ magic to move objects forward in the fantasy world. Also, just as human beings can trivially comprehend how their own bodily rotations change what "forward" means, we wanted spells to be able to make dynamic adjustments when the player's avatar rotates. Aside from this use of the API's powerful and intuitive

spatial affordances, the only "magical"' API call used above is the basic move() operation. But it has been wrapped within a non-terminating Java control structure, so that it will run until the player manually stops the enchantment. Until then, the player can fly freely throughout her virtual 3D environment without knowledge of the underlying vectors and quaternions that make this possible.

Emergent Pedagogical Properties of Embodiment

The experience of embodiment instilled by immersive RPGs gives a CodeSpells program (spell) the potential to take on more personal significance than, say, a program written in Alice or Scratch. Rather than making movies or games (as in Alice or Scratch), a CodeSpells player is able to live out fantasies that are essentially embodied. By jumping on top of an inanimate object and casting the flight spell, a player can live out the fantasy of flight. Many of our users have expressed that this is enjoyable.

Thus, the emotional payoff for learning how to program in CodeSpells can be high. Furthermore, because we have designed the APIs for simplicity, a spell of just a few lines stands in the way of the novice and these moments of accomplishment and joy. The learner only needs to meet a handful of learning goals before receiving positive feedback.

8.5   Evaluation

After validating that the CodeSpells API and environment would make it possible for programmers to design and write spells (programs) that were using typical CS1 learning goals, we decided to conduct and exploratory study to understand how novices responded to the API. Specifically we were interested in the ability of the novice programmer to:

- Use the spells correctly without modification,

- Modify the spells correctly to accomplish a desired goal and

- Create their own spells using the API that currently exists.
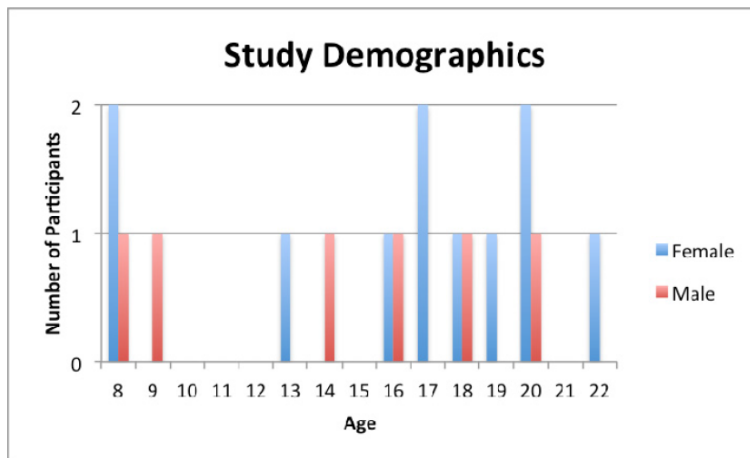
Figure 8.3. Study Demographics.

8.5.1   Methodology

We recruited 17 subjects to play CodeSpells (see Figure 8.3). None of the subjects had prior programming experience in any industry standard language but all knew what "programming" meant.

Each subject completed a background questionnaire before attending the study. Before the study began each subject was walked through the game mechanics by one of our research team. We then encouraged them to attempt to solve the quests in the game, but did not provide any other guidance or assistance unless they requested it. We recorded each session with standard-video equipment. Finally, we conducted a 15-minute semi-structured exit interview where we queried their immersive experience.

8.6   Results

Due to the explorative nature of this introductory study we were able to receive very intriguing results on the game-play behavior of our subjects. We will first discuss their ability to interact with the programs then discuss their sense of immersion in the game.

8.6.1 Using Spells

It is very encouraging that each one of our subjects was able to successfully cast at least 6 of the 8 spells that were in their spellbook (meaning they were able to successfully execute 6 of the 8 small Java programs). The documentation provided in the spellbook was not as detailed as a textbook or a tutorial, therefore to read and test spells which led to a correct understanding of those spells within only a 45 minute period with no prior experience with the CodeSpells API or any Java was astounding.

8.6.2 Modifying Spells

We also witnessed an average of 20 code edits in each 45-minute session. A code edit is defined as a successful change in a method argument, number of loop iterations, an entire API call or moving/adding/deleting entire lines of code. This level of interaction with code was surprising to us, especially since each code edit was a positive change towards a desired goal.

8.6.3 Creating Spells

Finally, 2 of our subjects were able to successfully write their own spells. Only 2 of our subjects wrote their own spells because in the short session only 2 were able to successfully complete all of the 8 challenges presented to them. Once they had completed the challenges, they were able to come up with their own challenge and design their own solution for it. One of these subjects designed a particularly interesting spell: a portal spell that used the staff (introduced in Section 8.4.1 that was meant for referencing many objects) to create a portal on one side of the river and send anything that stepped inside the portal to the other side. This innovation in such a short period was very positive.

8.6.4   Physical Immersion and its Value System

As is the case for perhaps all game designers, our intention was to architect a new world – to fully recontextualize computer programming. In our case, we wanted a world where magic exists and where wizards can break the laws of physics. But more than a world with new *mechanics*, we wanted a world with new *values* – one where the protagonist feels entwined with the fate of the world, one where one's education is meaningful and valued, one where executable code is an exciting arcane mystery rather than a "thing for nerds", and one where anyone can safely learn those mysteries. We focused on the exit interviews to gather information about the effects of the immersive properties.

Physical Immersion. In each of the exit interviews, subjects talked about how their programs affected *their* world. When asked about times when they might have struggled a bit to accomplish a task, they typically described the situation as "I was trying to levitate the rock that I was standing on, but then I accidentally clicked on it and I fell off". It was also interesting to hear how these subjects would talk about this environment with others. When asked if they would play CodeSpells with peers, one subject responded "Yeah, that would be cool because we could work together to do harder things. Like I could levitate a rock while they caught it on fire." In our informal experience with students using other interactive environments, such as Alice, these speech acts differ in that our users seem to have an embodied experience with CodeSpells, whereas Alice users typically talk about the objects in the scene in the third-person.

Values Immersion. Along the same lines, using the environment of a video game was also very interesting as it applied to immersion. Our players seemed to not give up easily when attempting a difficult quest or challenge. This was evident by their exit interviews, when asked about their experience with particularly difficult challenges, one subject described her approach by saying "[I] never gave up – because it's a video game!" This is not a typical sentiment held by novice programmers, especially with girls we often

see a feeling a failure or discouragement when the program doesn't work properly. In this case, the immersive features seemed to have made the difficult challenges *fun*.

It seems that through the CodeSpells API, environment and metaphor, our subjects became immersed in their programming, something we hope all novices can do in an introductory setting.

8.7 Discussion and Future Work

This study is particularly interesting to the computer science education community because it has begun to address two major concerns: 1. Students' inability to understand how abstract programs effect the environment and 2. Students' determination to succeed during difficult programming challenges. We are excited by these initial results and are encouraged to continue this research with more subjects over longer periods of time.

In the next three months we will be running a large CodeSpells study with 35 9-10 year old students in a near-by elementary school. During this study we plan to fully examine and explore the immersion properties of CodeSpells and how each aspect (the API, the video game features, the 3D environment) effect student motivation, learning and efficacy. We are also interested to see how CodeSpells compares with programming environments such as Media Computation and Alice. We believe that CodeSpells will offer more motivation for students due to it's immersive qualities.

8.8 Conclusion

Our goal was to get novice programmers immersed in a programming environment that would lead to their ability to read, execute, understand, modify and create spells (programs) with determination and a positive view of their ability. We believe we have begun to show how CodeSpells is a viable solution for encouraging novice programmers to invest in their programming as they do with video games.

We plan to run a larger study where we can compare how novice programmers engage with CodeSpells and Media Computation (which is used in our institutions CS1 course), as well as other environments such as Alice and Scratch that also offer immersive qualities. We plan to improve on our API as we measure learning gains in our future studies, but posit that the level of authenticity versus power, along with the metaphor of magic, engages novice programmers such that they can feel positive about their struggles when learning their first industry-standard language.

Our contributions are two-fold: We have presented here an API and environment that has:

- Allowed novice programmers to read, execute, understand, modify and write CS1-level Java programs within a 45- minute period.

- Immersed novice programmers into their programming environment so much so that they have developed a determination to solve problems and a positive outlook on programming challenges.

8.9   Acknowledgments

Chapter 9

Designing Quests to Teach Programming Skills

Chapters 7 and 8 focus on the enculturation of children through CodeSpells. In this chapter we evaluate the programming tasks required of the students during gameplay and contribute best practices in engaging children in the development of programming *skills*.

9.1   The Role of Quests in Serious Games

In serious games, quests are often employed to reduce complications. Here, quests are defined as **the requirement posed to the player to engage in some set of activities leading to some learning goal**. CodeSpells brings the quests to the forefront of the game, making them the primary motivating factor and rewarding students with badges as they complete each quest. CodeSpells is a fairly new, novice programming environment that has been shown to engage students learning to program by providing an accessible metaphor: magic [54]. CodeSpells was designed to teach basic Java programming concepts to novice programmers without necessitating the presence of an expert programmer. Since one of the most critical components of CodeSpells is a quest, this paper focuses on students' interaction with quests and the effects of these interactions on their learning.

Before conducting the study, an exploratory study looking at how students engaged with the CodeSpells environment was run. Students' engagement in the quests during

gameplay was analyzed and it was found that students did not explore the code as expected. As a result, an additional ten quests, requiring significant code edits, were added to the environment. A second group of students' game-play during the first six quests was then observed. Although students had not yet been required to engage in the more challenging quests, they were made aware of the more complex quests. It is clear that their behavior in the game and their relationship with the spells/code was aligned with the intentions of the game. These results are supported with quotations from students throughout game-play, showing that students recognized that they were meant to understand, edit, and create spells. This paper also presents lessons learned in quest design.

9.2   Related Work

Novice programming environments, such as Alice, Logo, and Scratch, have been shown effective in teaching novice programmers [34, 89, 95]. The issue remains, however, that these environments require some external curriculum to scaffold the students' learning; they lack a built-in structure to guide students through exploring a particular set of concepts. Research has been done on effective assignments that engage the students in cognitive apprenticeship, helping them understand that they need to write, test, modify, *explore*, and code. Much like the quests in CodeSpells, the scaffolding provided by the Alice textbook and exploratory homeworks guide students through exploring particular concepts [46, 56].

Additional contexts for textual based languages have also been explored. For example, Media Computation for Python and Java has had huge success in reducing the barrier for students in editing and creating code [68]. Such approaches to introducing computer science try to make it more appealing by replacing traditional, contrived programming assignments with more practical, engaging ones that require students to explore computing concepts through media creation and manipulation. Providing students with a more practical and engaging context has also been previously studied in regards to CodeSpells [55], though the

specific benefits of the quests were not previously addressed, they will be addressed here.

Although programming contexts, assignments, and textbooks have been improving; the equivalent is still new in serious games. Serious games have had success in STEM education and, in particular, teaching computer science [71]. They have been found to increase student motivation and more actively engage students in interacting with and exploring concepts. The design of serious games is a very difficult challenge [84], as it requires good game design, to ensure engagement, and the development of an effective learning environment.

This paper focuses on the issue of the in-game quest design to provide scaffolding and expectations to the students. Gidget is another serious game that aims to teach programming and has also begun to explore this space. Recently, Lee et. al. have published work on in-game assessments and how they affect students [89]. Lee describes two versions of Gidget, one where students were assessed throughout gameplay and one where they were not. Surprisingly, students who were given the version with in-game assessments were able to solve more quests/levels in the same amount of time as students who did not receive the assessments. The assessments are similar to the additional of ten quests presented in this paper, which positively affect students' engagement in complex coding tasks farther than the "critical qualities of non-institutional computer science learning spaces," specifically, learner-structured activities, exploration /creativity, programming as empowerment, difficulty stopping, and "countless" hours [54].

9.3   Exploratory Learning Study

Twenty-four students in a local 4th grade classroom were observed playing Code-Spells for 4 hours to explore how students engage in the quests. Of the twenty-four students, eleven were female and thirteen male, all aged 9 and 10. Students were presented with a talk about why learning computer science is important, were given a brief interactive

group demonstration introducing game mechanics and features of CodeSpells, and then began playing CodeSpells in assigned pairs. Note that during the demonstration of the game mechanics, students were introduced to the SpellBook (which contains starter spells), how to read it, copy spells from it, and cast the spells that they copied. Students were also shown the procedure for opening and modifying a spell. These actions are required for engaging with the code. Students then began playing CodeSpells in self-selected groups of 1-3 students. They had access to a tutor throughout game-play if they had questions, but the tutor limited responses to encouragements to explore the code and its effects.

9.4   CodeSpells Base Quests

CodeSpells originally had six quests where students had to complete some specific task for gnomes in the 3D world. Quests were the main source of scaffolding for students and encouraged code exploration after completion. Each quest introduced a working spell, provided a scenario for testing that spell, and suggested modifications to the spell. Quests are described in Table 9.1.

9.5   Quest Engagement Analysis

The six initial quests presented in Table 9.1 were designed to familiarize students with the relationship between code and its effects in the 3D world. Students were meant to engage in a quest (by talking to a gnome), go through the SpellBook reading the spells and accompanying descriptions, choose the appropriate spell, and try it in the world. After completing the quests students were highly encouraged to create their own spells. Although this was our intention, students did not engage with the spells as we expected them to: they did not appear to engage with the code after completing a quest. We attempted to rectify this issue with additional quests that were more open-ended and required more code-modification.

Table 9.1. Description of the Initial 6 Quests

Picking Up Stuff	This was a very simple quest that only required clicking on a crate in the world and transferring it to the inventory. This quest did not involve any spells.
Cross River	This quest required the use of the Teleportation spell to teleport oneself across the river. It did not require any spell modifications, but suggested that the student try teleporting other objects as well.
New Heights	This quest required the levitation of a crate to the top of a building. The original spell only levitated the crate 10% of the necessary height. Students should change the conditional in a while from 10 to 100: `while(counter < 10) --> while(counter < 100)`

Out of Reach	Students were expected to fly around the rooftops to collect floating bread. They cast the Flight spell on the crate used in New Heights. Students are encouraged to modify the code to fly faster. Students should change a parameter from 0.2 to 0.4:
	
	`target.move(Direction.forward(), 0.2); -->`
	`target.move(Direction.forward(), 0.4);`

Firefighter	This quest expected students to summon the rain cloud across the world to a flaming crate. Students were then encouraged to try to summon other objects to themselves.
Light Fire	This quest required students to set two crates on fire using the Flame spell. They were then asked to modify the spell to extinguish the fires. Students should change a Boolean parameter from true to false: `target.onFire(true); --> target.onFire(false);`

9.6   Quest Modification Study

9.6.1   The Study Setup

Based on what we observed in the exploratory study, we ran a second study during a 7-day summer program. Eight female and eight male students aged 8-12 participated. Two of the students reported having minor Scratch programming experience and two other students reported having watched a parent program. The remaining twelve students had no prior exposure to programming. Students were given the chance to play CodeSpells for seven sessions, totaling approximately 10 hours. As with the exploratory study, students were given an introductory talk about why learning computer science is important and the game mechanics of CodeSpells. Students then began playing CodeSpells in self-selected groups of 1-3 students. Discussion and collaboration within and among groups was both encouraged and observed. Students had access to a tutor throughout gameplay, though the tutor limited responses to help deciphering syntax errors and encouragement to explore the code and its effects.

9.6.2   New Quests

In addition to the quests presented in Table 9.1, ten new quests were added, all of which required code editing. Two of the quests were automatically checked by the in-game program analysis. The remaining eight were manually checked during game-play due to their complicated nature. Each of these new quests required students to modify or create spells.

9.7   Quest Engagement Data Analysis

Interesting patterns in students' engagement with the code required to complete quests were found in this study. The students focused their problem solving strategies around the code, specifically their ability to modify and create spells. This was evident in

Table 9.2. Additional 10 complex quests

Unlevitate	In this quest, students were asked to lower 3 crates to the ground. As in the New Heights quest, students were expected to modify the Levitation spell to loop 100 versus 10 times and change from up to down.

Summon Object	Students were expected to modify the Summoning spell to summon any target object to them. The modified Summon spell is the following:
	
	```Enchanted target = getTarget();```
	```Enchanted me = getByName('‘Me");```
	```while(target.distanceTo(me) > 5) {```
	``` Direction to_me = Direction.between(target,me);```
	``` target.move(to_me, 5);```
	```}```

Square Dance	Students were expected to modify the Sentry spell so that it would move North - West - East - South, instead of simply West and East. The correct spell is the following:
	
	```Enchanted target = getTarget();```
	```while(true) {```
	``` moveNorth(target);```
	``` moveEast(target);```
	``` moveSouth(target);```
	``` moveWest(target);```
	```}```
	
	With four helped methods structured as follows:
	
	```public void moveEast(Enchanted target) {```
	``` for(int i = 0; i <100; i++) {```
	``` target.move(Direction.east(), 0.2);```
	``` }```
	```}```
	
	The three other methods replaced east with north, south or west.

My Dance	This quest allowed students to create their own spell for moving a crate using Square Dance as a model.

Table 9.2. (Cont'd) Additional 10 complex quests

Massive Levitate	This quest required students to levitate all objects within a specific area. To do this, students created an area by dropping a magical staff from the inventory to the world. They then modified the Massive Fire spell to levitate all objects in the area instead of catching them on fire. The Massive Fire spell:
	
	``
	Enchanted area = getByName(''Area 0");
	EnchantedList list = area.findWithin();
	area.grow(10);
	for(Enchanted target : list)
	  target.move(Direction.up(), 5);
	``

Massive Dance	This quest combines the previous two quests. Students replace the bold line in Massive Levitate with their calls to their creative dance. They also copy over their helper methods into the Massive Dance class.
Follow the Leader	Students were expected to modify the Summon spell for this quest. Students added a while(true) around the conditional while of their Summon Object spell.

Portal	In this quest, students were expected to change the Teleportation spell to teleport anything in one area to another area. The Portal spell is as follows:
	
	``
	Enchanted area1 = getByName(''Area 1");
	Enchanted area2 = getByName(''Area 2");
	Location temp = area1.getLocation();
	Location dest = temp.adjust(Direction.up(), 10);
	while(true) {
	  EnchantedList list = area2.findWithin();
	    for(Enchanted target : list)
	      target.setLocation(dest);
	}
	``

Table 9.2. (Cont'd) Additional 10 complex quests

Umbrella	In this quest, students were expected to change the Follow the Leader spell so that it would have a specific crate always follow a specific gnome named Alex. The solution:
| | ```
Enchanted target = getTarget();
Enchanted alex = getByName(''Alex");
Location loc = alex.getLocation();
Location above = loc.adjust(Direction.up(), 5);
target.setLocation(above);
while(true){
  loc = alex.getLocation();
  above = loc.adjust(Direction.up(), 5);
  target.setLocation(above);
}
``` |

the logs and video taken of student game-play. The students were made aware of the more complex quests (described in Table 9.2) they would need to complete, but were required to first complete the earlier quests.

After casting a spell, the hope was that students edited the spell they had just cast. The intention was always for students to be able to cast a working spell to understand its effects and then to experiment with changing that spell to do something slightly different, either out of curiosity or for the sake of a quest. Per session, students edited a spell approximately twelve times after casting. This means they edited about 18% of their spells. Further analysis of these edits made it clear that students were fully engaged with their code. Students made 1054 edits on average, adding 610 characters and removing 444 characters each session. In reviewing the video data collected on students while editing spells, students were making complex edits: copying lines of code, combining spells, and creating spells rather than simply change parameters. If students modified a working spell and received a syntax error, they looked through correct code in their SpellBook or asked help from a tutor to resolve the error, rather than copying a new working spell and starting over. This

behavior is the intention of the additional ten complex quests.

9.7.1 Analyzing Student Speech Acts

Audio and video data collected while students were playing CodeSpells was analyzed to improve the understanding of student experiences. This paper focuses on the analysis of the 80-minute session in which students completed the first six quests described in Table 1. Based on speech acts of the students during game play, it seemed that they were approaching quests knowing that changing and understanding spells is the expectation.

For example, one pair attempted to complete the Out of Reach quest: *"I just want to read the spells."* The pair commenced reading the page on Flight in the SpellBook for at least 30 seconds. They then opened the Flight spell in the IDE, and said, *"This while means repeat over and over and over."* They then made a code edit. Another student also read the Flight spell and prepared for the changes she would make once she copied it out of the SpellBook. While reading Flight in the SpellBook (without having used it yet), *"I'm going to fly 1000 feet high. I figured out how."* While saying this, she highlighted the value in the levitation portion of the spell that could be changed to make this happen.

One student wanted to make his levitate spell levitate down at a faster rate. *"Oh my god! I know how to make levitate go faster..."* He then changes 0.1 to 0.2, then 0.4, then 0.9 *"I'm doing 0.9 now. Whoa, 0.9 is awesome! It goes down a lot."* Another student didn't wait to be presented with undefined spells, and instead began experimenting with spells of his own creation. One student even got frustrated when he saw that another student was earning badges too quickly. Talking about spells, he said, *"You actually should use it. Don't just like do it, use it. Have fun! It's a video game!"* He then proceeds to copy a spell out of the SpellBook, test it and modify it to do something slightly different.

9.8   Lessons Learned

Though one benefit of novice environments and serious games is that the students can perform complex actions with relatively less effort, based on these results, the expectation for more effort needs to be made clear from the beginning. Lowering the barrier for exploring code is critical, but students need to recognize that they need to understand the code they are using. We therefore provide three key lessons to consider when creating in-game quests for serious games:

1. Providing working code that can be used to build on (starter code) or build from (requiring the students to write something similar) allows students to test the code and understand its effect on the environment.

2. Providing the expectation that they will have to edit code from the beginning is essential. This could be a reward system that rewards only code-edits and not just code-usage.

3. Providing a more directed exploration through the quests give them a sense of the complexity ordering. This allows them to choose the challenge they want to overcome.

9.9   Conclusion

Based on a study that shows students engagement with quests within CodeSpells, three key lessons to consider when creating in-game quests have been presented. To develop these guidelines, students were first observed during an exploratory study to understand the issues with the initial quests within CodeSpells. Additional quests that required more complex code edits were added and the expectation for the students to be able to complete them was given. A second set of students was then observed and their game-play was analyzed. It was observed that the expectation for significant code changes encouraged

students to engage with code in a more suitable way. Future work will explore the likelihood of similar results in the absence of tutors.

9.10 Acknowledgments

Chapter 9, in full, is a reprint of the material as it appears in CCSC-SW 2014. Esper, Sarah; Wood, Samantha R.; Foster, Stephen R.; Lerner, Sorin; Griswold, William G., Consortium for Computing Sciences in Colleges, 2014. The dissertation author was the primary investigator and author of this paper.

Chapter 10

Bridging Educational Language Features: Industry-Standard Languages Can Support Enculturation and Skills Development

This chapter evaluates the changes made to CodeSpells in both encultration and skills development for children playing the game. We introduce the new features of the CodeSpells software and the additional CodeSpells curriculum that was created to take advantage of best practices of educational programming languages while teaching an industry-standard language. These features and curriculum were tested during an 8-week study where 55 9-10 year old students played CodeSpells 1-hour per week and were also required to work through the guided workbook each day.

Section 10.1 of this paper will give a background on CodeSpells and previous research done on other programming languages and environments that led to the changes made for this study. Section 10.2 describes the motivation for the feature changes to the CodeSpells software. Section 10.3 describes the curriculum changes, including the guided workbook. Section 10.4 describes the study. Sections 10.5 and 10.6 describes the results of the study; the students ability to write Java code on paper. Section 10.7 discusses how these results might impact the computer science education research community.

160

10.1 Previous Work

Significant background research on the importance of CodeSpells has been presented in previous work that will be described here [55, 54, 57]. This paper will focus on the changes in CodeSpells, and will introduce the concept of Epistemic Frames, which will be described in Section 10.2.

10.1.1 Motivation Behind CodeSpells

CodeSpells was designed to engage novice students in a more authentic computer science educational experience by using a textual programming language while attempting to recreate a non-formal learning environment for novice programmers [55]. Thirty "origin stories" were gathered and analyzed to find a common thread of what sparked a life-long interest in computer science. Specifically, 5 qualities were continuously found: Learner-Structured Activities, Exploration/Creativity/Play, Programming as Empowerment, Difficulty Stopping, and spending "Countless" Hours practicing.

Study

A very basic version of CodeSpells was then tested on 40 middle-school girls in a 1-hour session. The girls were presented with a forest that had three crates spread out amongst the trees. They were also given an inventory with three unique spells; moving side to side, moving up and down and teleporting. They were encouraged to cast the three spells they were given onto the crates, to modify the spells, and were explicitly told to "do interesting things." After one hour of game-play, they were broken up into groups of 12-15 students and participated in a focus group style interview.

Results

In such a short study where participants are unable to play outside of the lab environment it is impossible to determine if the girls experienced difficulty stopping or would

spend "countless" hours playing, but it was observed that they engaged in learner-structure activities and explored the environment in similar ways that were described by the expert programmers in the origin stories. One way in which empowerment was observed was their persistence in solving the programming problems. When confronted with a syntactic or semantic error, they would tell a facilitator that they "knew it could be done" and that they "just needed to figure out how". In the focus group interview they described programming as a "way to accomplish anything".

Contributions

From the two studies presented in [55], it was clear that if the proof-of-concept was pursued, it might yield a way to introduce textual-based languages to younger students. More broadly speaking, it was found that engaging novices in the 5 qualities of life-long programmers is beneficial and can be done outside of the CodeSpells environment.

10.1.2 Designing a Game

After the findings in [55], a *game* was designed within the CodeSpells environment [54]. This game involved three specific aspects meant to not only engage the students in programming concepts, but begin to immerse them in the study of *computer science*: 1. The metaphor of magic, 2. An API that encouraged embodiment, and 3. A way to reference objects to encourage immersive behavior.

Magical Metaphor

The magical metaphor was partly inspired by Brooks' description of a programmer. He states "The magic of myth and legend has come true in our time. One types the correct incantation on a keyboard, and a display screen comes to life, showing things that never were nor could be" [18]. This metaphor is often used with computer programmers, but can be applied to learning since popular examples of magic show the effort, practice, and

mentorship it takes to learn magic (e.g. Harry Potter). **CodeSpells adopts the metaphor of *learnable* magic**. In the game version of CodeSpells, students were given an entire Spellbook; rather than only given three spells (programs). Students were also provided with non-player characters whose role was to encourage the exploration of the spells.

API

The API was designed to be similar to a Java applet with "cast()" instead of "run()". Students can cast spells on any object, but depending on the spell and the properties of the object different effects might occur. For example, attempting to the run code to catch an object on fire on a non-flammable object would yield no effect.

The complexity of the API was challenging. Previous novice programming APIs such as Media Computation [68], Alice [46] and Scratch [101] provided a theme: for actions and effects that are simple to do in the real world (e.g. moving objects), the API should be a simple line of code (e.g. target.move()), whereas complex actions (e.g. move multiple objects at once) should be more complex (e.g. requiring understanding of lists and iteration through lists).

Referencing Objects

Finally, an interesting question arose: *how can students engage with entities in the environment easily?* The referencing a single object was already established:

```
Enchanted target = getTarget();
```

To immerse the students even more, additional entity reference was added such that students could reference objects by name:

```
Enchanted myself = getByName(''Me");
```

As described above, having the ability to affect multiple objects with one spell is desired by students, yet difficult to do in a 3D environment. The solution was to introduce

Figure 10.1. Example of an area in the environment.

"Areas" (see Figure 10.1). An area could be placed in the world and the objects contained within an area are put into a list, and can then be iterated through:

```
Enchanted area = getByName(''Area1");
EnchantedList rocks = area.getWithin();
for(Enchanted rock : rocks){
  rock.onFire(true);
}
```

The goal of these changes was to better immerse the students in their learning experience.

Study and Results

To study the changes made in this version of CodeSpells, 17 participants from varying ages and backgrounds were asked to play CodeSpells for approximately 1-hour. Participants were asked to complete all quests and were given as much time as they wanted. Through background questionnaire surveys, observational notes, and exit interviews, it was discovered that participants were able to easily use and modify the spells. Students were also immersed into the environment.

Participants described **physical immersion** by referencing themselves; "I was trying to levitate the rock that I was standing on, but then I accidentally clicked on it and I fell off". Lee and Ko describe a similar phenomena with their programming feedback tool *Gidget* [88]. What Lee and Ko found was the if their tool was more personified, it increased students motivation to learn and therefore understand. Participants described their **values immersion** by describing their experiences; "[I] never gave up – because it's a video game!". Participants seemed to be motivated to learn in ways that aren't typically seen with traditional learning environments.

Contributions

This particular study showed that participants could engage with Java code regardless of age or prior programming experience. It also showed that the gamified approach, the immersive API, and interactions with the environment encouraged participants to be motivated and potentially learn more. What is still missing from this work is an evaluation of the learning activities (see Section 10.1.3) and an evaluation of what is actually being learned, which is the focus of this paper.

10.1.3   How to Properly Design Quests

CodeSpells was shown to be playable and to engage students in Java programming, and in [57] the design of CodeSpells quests, as well as a set of lessons learned on quests for all serious games were explored.

Study

There were two phases to the study in [57]. First, the basic quests that were designed during the previous studies were explored. Twenty-four 4th grade students were asked to play CodeSpells for 4 hours over the course of 4 weeks. There was a total of 6 quests that they were asked to complete, two of which are described in Table 10.1.

Table 10.1. Description of 2 Quests

Quest	Description
Cross River	This quest required the use of the Teleportation spell to teleport oneself across the river. It did not require any spell modifications, but suggested that the student try teleporting to other objects as well.

New Heights	This quest required the levitation of a crate to the top of a building. The original spell only levitated the crate 10% of the necessary height. Students should change the conditional in a while from 10 to 100:
	
	``
	while(counter < 10)
	 --> while(counter < 100)
	``

It was expected that students would engage in a quest (by starting a conversation with a gnome), go through the Spellbook *reading* the spells and descriptions, choose the appropriate spell, try it and then possibly modify it. What was actually observed was that students rarely read the descriptions and rarely modified the spells during or after the quest. In fact, when students were prompted to modify the quests, they seemed resistant.

A second study was then run during a 7-day summer program. Sixteen students between the ages of 8 and 12 little to no programming experience played CodeSpells for a total of 10 hours over 7 days. The students were introduced to an additional 10 complex quests described in [57]. Though the students still had to begin with the original quests, they were aware that the more engaging quests were something that they would have to eventually accomplish. The 10 additional quests required significantly more code edits than the first 6.

Results

After including the additional 10 complex quests to CodeSpells, students were observed to make complex code edits from the beginning of their game play. Debugging habits presented another difference between the students: the second group would debug while the first group would often start over.

What was perhaps most intriguing about this second study was the analysis of the speech acts. Students in the second group would say: "Oh my god! I know how to make levitate go faster..." The student then changed 0.1 to 0.2, then 0.4, then 0.9. "I'm doing 0.9 now. Whoa, 0.9 is awesome! It goes down a lot." The students even maintained the value immersion described in section 2.2.4. This student was describing to his classmates how they should interact with spells, "You actually should use it. Don't just like do it, use it. Have fun! It's a video game!"

Contributions

The key take-aways that should be considered when building educational activities, particularly in serious games were:

- Providing starter code that can be used to build off of (allows students to test understand the code and its effect on the environment.

- Providing the expectation that they will have to edit code from the beginning is essential.

- Providing a more directed exploration through the quests give them a sense of the complexity ordering.

The lessons learned from this study can be applied to a variety of learning environments, not just serious games. It also reinforced themes seen in the previous studies done on CodeSpells, such as values immersion.

10.2 CodeSpells: An Epistemic Game?

The research that was done in [55], [54], and [57] has led to improvements of CodeSpells in engagement, playability, motivation, and immersion/embodiment. Each of these studies touched on a piece of what CodeSpells was designed to do: **teach students how to be a computer scientist, not just how to program.**

It was clear at this point, that CodeSpells was designed to be an Epistemic Game. Epistemic Games, as described by Shaffer, are games in which students use experiences in video games, computer games, and other interactive learning environments to help them deal more effectively with situations outside of the original context of learning; mechanism for what is sometimes referred to as *transfer* [134].

Though CodeSpells requires that the students play the role of a wizard, rather than a computer scientist, it could be argued that if the students are gaining the skills, knowledge, values, epistemology and identity of a computer scientist, through playing CodeSpells, it could therefore be classified as an epistemic game for computer science. Based on the evidence thus far in CodeSpells research, it has been shown that students have begun to develop the epistemic frame of a computer scientists through game-play. However, it must be pointed out that there is no proof that the students will develop the epistemic frame of an *expert* programmer, but rather a novice programmer.

If CodeSpells can, on it's own, help students become enculturated into the community of computer scientists, then it can be a powerful software and experience that can be shared amongst millions. As seen in [55], the beginning of the journey to become a computer scientist is an important one, therefore to carefully craft that beginning and make it accessible is a worthy goal. With the lack of computer science teachers at K-16 levels, **it is the goal of CodeSpells to provide a rich experience of computer science education to students who may not have access to an educator.** Specifically, this paper introduces a study that shows that students can learn basic computer science topics, through demonstrating their knowledge on a final written exam. It also introduces a changes in the software and curriculum, based on the previous studies and research described above.

10.3 Software/Curriculum Changes

There were three major changes to the CodeSpells software and curriculum that will be highlighted in this paper:

- **Spellbook:** The Spellbook should focus on *micro-design patterns*, not spells.

- **Quests:** Quests should involve more programming from the beginning.

- **Physical Spellbook:** A physical Spellbook should be introduced to get students familiar with topics outside of the game.

10.3.1 Spellbook

The Spellbook was changed in two major ways; students could only copy out the first (blank) spell (see Figure 10.2), and each page in the Spellbook introduced a basic *micro-design pattern*. In this paper, *micro-design patterns* are defined as basic programming structures that are evident in nearly all programming languages and often taught in a CS0 or CS1 course. Examples (non-exhaustive) of such *micro-design patterns* are: If-Statements, For-Loops, Method Calls, or Parameter Passing.

By focusing the Spellbook on *micro-design patterns* instead of spells, students will be required to understand the pages more, since they won't be able to just copy the spell out and test it. This attentiveness has always been a goal, but students were never interested in reading the Spellbook. It is clear that just changing what is written in the Spellbook won't automatically make students read the Spellbook, but at least now if they read it is has a useful piece of information, rather than a description of a spell.

Another way that students were required to be more attentive to the Spellbook is by not allowing students to copy any pre-written spells out of the Spellbook. Figures 10.2 through 10.5 show the first four pages of the Spellbook. The first page is the only page they are allowed to copy out directly. This means that from the very beginning the students must

Figure 10.2. Page 1 is the only spell that the students can copy out of the Spellbook.

Figure 10.3. Page 2 describes how students will interact with objects.

Figure 10.4. Page 3 describes method calls.

Figure 10.5. Page 4 holds the API that is available to the students.

Figure 10.6. Page 5 introduces the concept of parameters.

Figure 10.7. Students can still access pages of the Spellbook while working in the IDE.

write all of their spells, even the simplest ones. The next two pages introduce the most basic *micro-design patterns* that the students can use: naming an object and calling a method on that object. Finally, the fourth page lists all the methods that are available to the students to use.

The layout of the Spellbook is much more representative of resources that expert programmers would use; essentially an API with some very simple examples. When the students go back into the environment, they are still able to access the Spellbook through the IDE, as seen in Figure 10.7.

Additional Spellbook pages are unlocked once the student unlocks certain badges. This helps scaffold the introduction of *micro-design patterns*. For example, the next page that gets introduced can be seen in Figure 10.6 . By introducing more complex concepts after the students have had a chance to experiment with the basic spell, the new Spellbook is also following the Inventing to Prepare for Learning (IPL) sequence [130].

Figure 10.8. Quest 1 and 2 require the students to collect the piece of bread by casting a spell on the crate to make it move and hit each piece of bread.

10.3.2   Quests

All of the quests were completely re-written for this study. The environment was setup such that the students were not able to leave a particular area until they had accomplished certain quests. This is similar to how other video games are designed, and would allow for a scaffolded learning experience. It also linked well with the way the Spellbook was scaffolded. Showing students that they needed to earn the right to use more complex magic and to explore the rest of the land.

Area 1

In area 1 students were introduced to the basic method calling (as shown in Figures 10.2 through 10.5). The task was to write a spell that could be cast on the crate to make it collect the one piece of bread. This should be a simple-enough task, considering the third page of the Spellbook contained the exact code the students would need to write (see Figure 10.4).

Once the students wrote and cast this spell, they would be faced with a new challenge;

see Figure 10.8. Though the students had everything they needed on page 4 of the Spellbook (see Figure 10.5), this still posed a challenge because up until now, the concept of writing one spell to do multiple actions wasn't something they had encountered. It is also tricky because the students also had to understand the need to reset the crate for testing purposes. If they wrote code that didn't work the first time, they would have to write code that would reset the create. Therefore the best solution might be:

```
Enchanted target = getTarget();

target.moveSouth();

target.moveUp();

target.moveUp();

target.moveEast();

target.moveDown();

target.moveDown();

target.moveNorth();

target.moveWest();
```

Once the students demonstrated their ability to write a spell that called multiple methods on one object, the next two pages in the Spellbook unlocked (see Figures 10.6). At this point the students hardly had to make any changes:

```
Enchanted target = getTarget();

target.moveSouth();

target.moveUp(2);

target.moveEast();

target.moveDown(2);

target.moveNorth();

target.moveWest();
```

The first area introduced the students to Object Naming, Method Calling, Basic Algorithms, Parameter Passing, and Basic Refactoring. Even though the students are not completely aware of all of these computer science concepts, and even though they haven't begun to master them yet, they have been introduced to them and the language of the game has begun to enculturate them.

At this point it could be said that the change in the quests has carved out a piece of the epistemic frame of a computer scientist. The students demonstrated (or at the very least were exposed to):

- **Skills**: In their ability to write code.

- **Knowledge**: In their ability to write a spell of their own design.

- **Values**: In the need to refactor when it is evident that there is a more efficient way.

- **Identity**: In their act of writing the spell from scratch.

- **Epistemology**: In the reason behind using parameters.

This does not prove that they are forming an epistemic frame of a computer scientist, but it allows for the possibility of such. The contributing factor of CodeSpells is that it represents an entry-point of the field for a majority of people, not that it helps novices *become* computer scientists all within the game.

10.3.3   Physical Spellbook

The final change to CodeSpells that was made for this study was the addition of a physical Spellbook. What was observed in previous CodeSpells studies is that students never wanted to take the time to read the Spellbook. The generally fast-paced nature of games might, therefore, have something to do with the students' unlikelihood to spend the time reading while playing. It has previously been observed, however, that gamers who

Figure 10.9. The first exercise is a code-tracing problem.

Figure 10.10. The second exercise is a code-tracing problem.

Figure 10.11. The third exercise is a code-writing problem.

play complex games often **do** study the game, with a combination of game play and outside resources [60].

Therefore, a physical Spellbook was provided to each student. Activities for the students to complete on paper were also provided with the Spellbook. For example, before the students were introduced to the additional two pages of the Spellbook that introduced parameters, they were asked to engage in three exercises; shown in Figures 10.9 - 10.11. The goal of these problems was to encourage the students to understand the spells that they were writing outside of the exact context for which they were writing them (i.e. applying them to a new problem). Similarly to the changes in quests, this does not necessarily follow that the students will be able to apply this knowledge to *any* problem placed in front of them, but if they have the ability to transfer the knowledge to at least one other problem it demonstrates for them the importance of understanding the concept, and not just memorizing the one solution.

10.4   Study

10.4.1   Participants

Two different schools agreed to participate in the study, one was an affluent private school and the other was a bilingual public school. Each school allowed the research team to attend their 4th grade classroom once a week for a total of 10 weeks. There was a total of 55 4th grade students (ages 9-10), 24 were from the private school, 31 were from the public school. Each of the students were encouraged to participate by their teachers, but no student was required to participate in any part of the study. If they chose to not participate, the teacher provided them with other educational games to play (such as SumDog [146]). In the end, a total of 50 students participated for the entire duration of the study.

Each of the participants were paired up based on their results from the background survey they answered on the first day and with input from their teachers. This allowed each

pair to be as closely matched on experience and confidence.

10.4.2 Methodology

At the beginning of the study, the students were asked to fill out a 15-minute online background survey. In this survey they were asked about their experience with computers, video games and programming. They were also asked a series of growth mindset questions relating to math and programming. Finally, they were asked questions relating to identity and programming (such as "Do you think you are a programmer").

After the students filled out the background survey, a class-wide discussion was led on what programming is and why it might be interesting for them to learn. The Code.org video was shown [25] and students were encouraged to describe why programming is important for everyone to learn.

Each week, the research team would arrive and the students would pair up with their partners. They would begin the game-play in a traditional pair-programming fashion [5]. Every 5 minutes they were asked to switch driver-navigator roles. The students were able to proceed at their own pace. When a pair reached a particular milestone in the game, they were then asked to work on their physical Spellbook. Whenever working in the physical Spellbook, they had to do so quietly and without help from their partner. This was meant to account for a pair where one partner was doing all of the coding.

Throughout the 8-weeks, the CodeSpells research team would help the students when they needed assistance. The CodeSpells research team, however, consisted predominately of psychologists with no programming experience. Therefore the type of help offered to the students was more encouragement, rather than a tutorial on the intricacies of computer science or programming.

At times, throughout the 10-weeks, if the majority of the students had a question relating to a computing concept, a short explanation was given out loud to the entire class. This was then recorded to incorporate into the Spellbook in future iterations.

On the final day, the students were asked to complete a final written exam individually. In this exam, students were asked to trace and write code. They were not provided the Spellbook for reference and had to complete the exam with no help from the research team. Once they completed the exam, they were asked to fill out an exit questionnaire that asked the same growth mindset, identity and perception questions as the background questionnaire.

10.5   Data Collection

10.5.1   Final Written Exam

To evaluate the students skills and knowledge of computer science, they were asked to complete a final, written exam. This exam was designed based on other CS0 level exams used in traditional undergraduate levels [101, 45].

10.5.2   Growth Mindset Questionnaire

Students were also given a Growth Mindset [49] questionnaire at the beginning, middle and end of the study where they were asked to answer eight questions about the fixedness of math and programming intelligence. Students were asked to answer on a scale of 1 - 4 (Disagree, Somewhat Disagree, Somewhat Agree, Agree).

10.6   Results

This study was designed to not only teach the students how to program, but also what it means to be a computer scientist. CodeSpells was designed to enculturate students, therefore, the pre- and post-surveys and the final written exam were designed to better understand students change in their Epistemic Frame.

Figure 10.12. Average score of each question on the final exam.

10.6.1 Skills, Knowledge and Epistemology

The final written exam yields the first results to demonstrate student learning using CodeSpells. Figure 10.12 shows the average score for each question and the total score across all students. The average total score for the exam was a 55%. Though this is a low score, one should remember that the students taking this final exam are 9 and 10 years old and have only been exposed to computer science and programming for a total of 8 hours. Additionally, this exam predominately required the students to write Java code. It is also important to look at the breakdown of scores across exam questions.

Questions 1 and 2

Question 1: Code Writing: Write a spell that collects 4 pieces of bread directly above the crate.

Solution:

```
target.moveUp(4);
```

Question 2: Code Writing: Write a spell that collects 8 pieced of bread in a horizontal rectangle above the crate.

Solution:

```
target.moveUp(2);
target.moveEast(3);
target.moveDown();
target.moveWest(3);
```

Questions 1 and 2 of the exam required the students to essentially write variations of programs that they had previously written within the CodeSpells environment. For example, in the CodeSpells environment, students were asked to write a spell that would collect the bread that was in a vertical rectangle (see Figure 10.8). On the exam, they were asked to write a spell that would collect bread in a horizontal rectangle. This isn't a major change, therefore it is expected that the students should perform well on these questions. The students scored, on average, 86% and 83% on the first two questions, respectively.

Question 3

Question 3: Code Writing: Write a spell where the target crate collects 4 pieces of bread in a square above it and a second crate named "Crate" also collects 4 pieces of bread in a square above it.

Solution:

```
Enchanted target = getTarget();
Enchanted crate = getByName(''Crate");
target.moveUp();
target.moveEast();
target.moveDown();
target.moveWest();
```

```
crate.moveUp();

crate.moveEast();

crate.moveDown();

crate.moveWest();
```

Question 3 asked the students to affect two objects in the world with only one spell. In this question, students scored 55% on average, which is far lower than the first two questions. We expect that this drop in score is because the students were introduced to the concept of defining two objects in one program through their physical Spellbook, but never had to write this code within the CodeSpells environment. This supports the idea that the physical Spellbook is not enough to facilitate students' learning of a concept, but that engaging with it in the immersive environment is critical.

Questions 4 and 5

Question 4: Identify Terminology: Name the parts of the code provided (Value, Type and Variable).

```
Enchanted target = getTarget();
```

Solution:

Enchanted is the **Type**, *target* is the **Variable** and *getTarget()* is the **Value**

Question 5: Identify Terminology: Circle the parameters in the following lines of code (answers surrounded by - -).

```
Enchanted target = getTarget();

Enchanted crate = getByName(--''Crate"--);

target.jump();

crate.jump(--3--);

target.moveUp(--4--);
```

Questions 4 and 5 required to students to have an understanding of four specific vocabulary words: Value, Type, Variable and Parameter. The students had *no* explicit exercise, within CodeSpells or their physical Spellbook, to engage them in this topic. The only way students would have been able to answer this question is through the implicit learning that they engaged in while reading the Spellbook (digital and physical) and asking questions. The students scored, on average, 54% and 61%, respectively, on these two questions. This is a great result, because without having a specific lesson what a parameter is, students were able to have a basic understanding through exposure alone.

Question 6

Question 6: Code Writing: Defining an existing method "jump(int num)".
Solution:

```
public void jump(int num){
    for(int i = 0; i < num; i++){
        target.moveUp();
        target.moveDown();
    }
}
```

Question 6 had three topics that it was testing:

- Can the students decompose a method to smaller method calls (jump() to moveUp() and moveDown()),

- Do the students understand how to define the use of a parameter (public void jump(int num) to where would they put num in their method?), and

- Can the students recognize that to jump a certain number of time requires a For-Loop (rather than moveUp(num) and moveDown(num).

In retrospect, this question is too loaded to be one question that addresses all of these concerns. However the students still, on average, scored 40% on this question. In looking at the student responses, every student was able to recognize the need for moveUp/moveDown, and 50% of the students gave a solution that was moveUp(num)/moveDown(num). Although only 2 students recognized the need for a for-loop, the results of this question still show that students were able to engage in the concept of defining a method. What is even more impressive, is that the students were not exposed to this in the CodeSpells environment *or* in the physical Spellbook at all.

Question 7

Question 7: Code Writing: Write 3 ways to get a target to move up 4 times.
Solution:

```
target.moveUp();

target.moveUp();

target.moveUp();

target.moveUp();

----------------------------

target.moveUp(4);

----------------------------

for(int i = 0; i < 4; i++){

    target.moveUp();

}
```

Question 7 asked the students to produce three different ways of writing a program to have the same effect in the environment. Every student was able to identify at least two ways: call moveUp() four times, and call moveUp(4). 22% of the students came up with a third solution that involved some clever use of parameters (for example, moveUp(2)/ moveUp(2)),

while only 13% of the students recognized the opportunity to use a for-loop in this situation. Since this question didn't specifically ask students to only use parameters in one of their solutions and the students never had to perform this type of activity during the study, it is again impressive that nearly every student was able to come up correct solutions.

Questions 8 and 9

Question 8: Code Tracing: How much will the target move up.

```
Enchanted target = getTarget();
target.moveUp();
target.moveUp(2);
for(int i = 0l i < 4; i++){
    target.moveUp(2);
}
target.moveUp();
```

Solution: 12

Question 9: Code Writing: Re-write the code from Question 8 such that you call moveUp() once.

Solution:

```
Enchanted target = getTarget();
for(int i = 0l i < 12; i++){
    target.moveUp();
}
```

Question 8 required the students to trace a complex program and determine how many times a crate moved up. This is an activity that the students did have to do in the physical Spellbook (see Figures 10.9 and 10.10). However, the students never had to trace

code that involved a for-loop. Only 3 students were able to successfully trace the code, yielding a very low average score for question 8. The positive side of this is that students were able to write an alternative program to have the same effect as question 8 but by using parameters or for-loops. 2 students wrote a solution that used a for-loop and 41% students wrote a solution that used parameters in some way. Considering that, again, students were never engaged in an activity that required them to write a alternative program that yields the same effect, the performance on this question is decently high.

Question 10

Question 10 was open ended and allowed students to simply explore possibilities of what they could do with programming. Students received one point if they wrote code or an explanation and 2 points in they wrote both. The lower score on this question is due to the fact that 11% of students only wrote code or an explanation and 24% of students decided not to write anything at all.

10.6.2   Values and Identity

One of the most important goals of CodeSpells is to model computer science behaviors and values for students, as well as engage them such that they develop a similar identity. As discussed earlier, it is not expected that after playing CodeSpells for ten weeks the students will *become* computer scientists, but it is expected that they will have begun their journey to becoming computer scientists. As in any 12-step program, the first step is to identify who you are trying to be. In the pre- and post-surveys, students were asked "Do you think you are a programmer?" and expected to answer with one of the following responses: **Yes.**, **Yes, but I am still learning.**, **Not yet, but I'm still learning.**, or **No.**.

Figure 10.13 demonstrates the students' self-identification as a programmer. It is evident that the students identified more as a programmer during the post-survey. What is also evident is that they recognized that they still had more to learn. In fact, only two

Figure 10.13. Student's responses to the question "Do you think you are a programmer" on the pre- and post- surveys.

students claimed that they **were** programmers. Though the **values** of computer scientists are not universally categorized, anecdotally a value of a computer scientist is that there is always more to learn, particularly because the field is always changing.

Though further interviews were not conducted, it is possible that the students were sharing the value of *always learning* that computer scientists have. It could be argued that the students were lacking confidence in their abilities, which is why they responded *Yes, but I'm still learning.* over *Yes.*, but the change from pre- to post- survey in favor of *Yes* over *Not Yet* would indicate a rise in confidence.

To better understand how the students defined a "programmer", an open-ended question was also asked on the pre- and post-surveys: "What kinds of people can be good programmers? Why?". All of the responses were 1 sentence long. The research team read through all 108 responses from both surveys and identified five categories that responses fit into:

- **Anyone.** "Everyone", "Anybody can", "Everyone can be good at programing because if they want to do it they should", "people that put effort into it"

Figure 10.14. Student's Open Ended Responses to the question "What kinds of people can be good programmers? Why?"

- **Smart People.** "Smart and talented people", "People with great minds and smartness"

- **Math oriented people.** "Good at math", "people who are good with math and numbers"

- **People who work on computers.** "People who have experience with computers", "People that play and love computers"

- **I don't know.** "I DO NOT KNOW!", "I don't know"

What was found after the categorization of all of the student responses was that the students thought that they had a better understanding of who programmers are; fewer students responded to the question "What kinds of people can be good programmers? Why?" with "I don't know". What is more exciting is that the students tended to respond with "Anyone" in the post-survey, which again supports the claim that they also increased confidence in their own abilities.

An interesting find was that students still believed that people who program should be smart, but their reasoning behind it had changed. For example, in the pre-survey responses

might be: "Smart people", "Smart and talented people", or "Smart intelligent people", whereas in the post-survey the answers categorized under "Smart people." were described **programmers' values**: "Smart learners can be really good programmers because they will pay attention and remember all the codes they need to be a programmer" or "Smart, patient and wise people". Similarly, the responses that were categorized as "Anyone." went from "Anyone" to "I think anyone could be a good programmer if they just never give up."

Furthermore, the students' mindsets did not change in fixedness, however, it does show that the students tend towards a growth mindset. It could therefore be argued that when students reply that "Smart people" can be good programmers, that they are referring to *anyone* who dedicates the time to increasing their intelligence.

10.7   Contributions

What the results have shown is that with the combination of Game-Play and Spell-book Activities, students were enculturated into the Epistemic Frame of novice computer scientists. Though this work does not suggest that other curriculums or softwares could do the same, it brings to light the importance of thinking of all five categories of epistemic frames when designing such curriculum and software. Particularly, it is critical to think of these five categories when designing software meant to be taught by a non-computer scientists. Considering that the education space of the 21st Century is becoming less personalized (with both large classrooms, online educational frameworks) it is important to consider how current technological advances can still promote enculturation, not just information delivery.

The contributions of this work, in particular, are the changes made in both the software and curriculum for CodeSpells. Not only is this a contribution for the work of CodeSpells, but also a contribution to all programming learning environments.

> **It is not critical that each curriculum or software completely encultur-**
> **ate the student, but it is critical, however, that each curriculum and soft-**
> **ware attempt to show the students a slice of the field from all aspects of**

the epistemic frame.

10.8 Children Become Enculturated and Develop Programming Skills While Playing CodeSpells

CodeSpells is a system that has been engaging young students in Java programming, while increasing their confidence and beginning to enculturate them into the discipline of computer science. In this paper we show that students are not only able to engage with Java, but they are able to understand and write basic Java code after only 8 hours of playing the game and working with the offline materials. This work shows potential for future iterations of both CodeSpells, and the curricular framework that it presents. Specifically, this work brings to light the importance of designing to enculturate novices, which can be guided by the Epistemic Frames Framework.

10.9 Conclusion

CodeSpells is a system that has been engaging young students in Java programming, while increasing their confidence and beginnign to enculturate them into the dicipline of computer science. In this paper we show that students are not only able to engage with Java, but they are able to understand and write basic Java code after only 8 hours of playing the game and working with the offline materials. This work shows potential for future iterations of both CodeSpells, and the curricular framework that it presents. Specifically, this work brings to light the importance of designing to enculturate novices, which can be guided by the Epistemic Frames Framework.

10.10 Acknowledgments

Chapter 11

Discussion and Future Work

Thus far I have introduced two major interventions to integrating culture into computer science learning experiences; one for in-person, undergraduate courses and one for children. The question remains about the generalize-ability of these contributions. Based on the results found in Chapters 3 through 6, Section 11.1 begins to measure the effects of using Peer Instruction in a course that focuses around risk management. PI could be very useful in this type of course because students have the opportunity to have instructor-led discussions about risks and benefits, however creating multiple choice questions is more difficult than in an introductory course. CodeSpells was studied and shown to teach children programming skills and become enculturated in Chapters 7 through 10. In Section 11.2 I generalize what was learned from these chapters using Epistemic Frames, but applied to computer science learning environments.

11.1   A Discussion on Adopting Peer Instruction in a Course Focused on Risk Management

Peer Instruction (PI) has been shown to significantly improve student experiences and learning in a range of computer science courses from introductory courses to more advanced topics such as theory of computation and computer architecture. But there has not yet been research on adopting PI in a computer science course that does not teach a

192

specific language, set of algorithms, or mathematical concept. This paper introduces an upper division software engineering course that adopted and modified PI to address risk management.

First the course and the modifications to PI are discussed, then a call to the community is made to discuss how PI might be adopted to courses where pre-defined clicker responses may not address specific concerns of the course. The results are discussed from a student survey where students were asked to reflect on their experiences with this modified PI approach compared to other courses where they did and did not use PI. Finally, a preliminary comparison is made between the questions presented in this course with other PI courses and the student reception of this course is compared with previous versions of the course.

11.1.1   Related Work

Peer Instruction was developed by Erik Mazur in physics education in 1991 [39] and has been shown to improve student performance in computing courses for more than a decade [45]. PI is a pedagogy that requires students to engage in a series of multiple-choice questions during lecture based on pre-lecture reading by answering them individually, and then discussing their answers and coming to a consensus within their group. The instructor then leads a class-wide discussion on the topic. Clicker questions are interspersed with mini-lectures where the instructor is able to build more knowledge around a specific topic now that the student has primed their mind to think about what the teacher is saying, and to use the new information to modify their model of understanding.

Recently, PI has been researched in a large variety of course topics, class sizes and institutions [116]. It has proven to be successful in student engagement, learning and efficacy in large and small computing courses in a range of liberal arts and research institutions. In 2011, Porter et al. show that PI group discussions improve student understanding of concepts in architecture and theory of computation courses [118]. In 2013, Simon et al. show that there is a significant improvement in student performance in a CS0 PI course compared to a

standard instruction course of the same professor, institution and term [139]. Porter et al. has also shown that PI, along with Pair Programming and Media Computation, contributed to a nearly 1/3 increase in retention of computer science majors and a significant reduction of fail rates in four different computing courses (CS1, CS1.5, architecture, and theory of computation) compared to the previous 10 years of each course (16 of those instances using PI) [119].

Though all of these results are encouraging and exciting, there has not yet been an evaluation of PI in a course focused on a topic without a single best answer, such as risk mitigation. Since PI has been shown to work in large classes, one approach to such a course would be to change the kinds of questions that should be asked and therefore what the students should be discussing during class. In 2012, Cutts et al. presented the Abstraction Transition Taxonomy, which is a classification system for identifying the "kinds of knowing and practices we engage our students in as we seek to apprentice them" [45]. Particularly interesting, but not extensively studied in this work, were the rationale questions. These are questions where students are asked to choose an answer that provides the best explanation for why it is correct. Of the seven final exams classified, rationale questions never exceeded 15% of the questions per exam. [45] discusses CS0, CS1 and CS2 courses but lacks in exploration of rationale questions in upper division courses, such as software engineering.

Although software engineering courses have not yet been shown to adopt Peer Instruction, there has been some research on other approaches to engaging students in an authentic experience within the classroom. In 2005, Hadjerrouit presents a pedagogical model that translates constructivist principles in a web-based course in object-oriented software engineering [69]. This course was designed to engage students in an authentic, large, group project that would teach standard software engineering practices. The course presented here similarly had a large project component that served as an authentic learning experience for the nearly 200 students. In addition to this project, the instructor had 3 hours

of lecture each week that could still be used as a valuable time-resource where they could interact with the students and describe the concepts at a meta-level.

11.1.2 Understanding the Courses

11.1.3 The Software Engineering Course

The course, entitled "Software Engineering" was taught in Winter 2013. There were a total of 189 students enrolled in the course; all were undergraduate students majoring in computer science who had completed at least CS2. The course consisted of pre-lecture reading and a reading quiz, followed by a modified PI lecture (described below). Lecture was 1.5 hours twice a week. Once a week there were also a 2-hour lab and a 1-hour discussion section both led by graduate teaching assistants. Pre-lecture reading quizzes were 5% of the grade, clicker participation (must participate in 80% of the questions, not based on correctness) was 5% of the grade, bi-weekly online quizzes were 10% of the grade, closed-labs were 10% of the grade, the project was 35% of the grade, and the final was 35% of the grade.

Course Learning Goals

The instructor of the course gave the students twelve specific learning goals in their syllabus. The seven that are presented here are extremely complex activities that are also situational.

1. How to acquire requirements from customers.

2. How to turn software requirements into executable code.

3. How to perform from-scratch software design, when it's needed.

4. How to productively employ component-oriented software development.

Figure 11.1. Description of how the PI lecture was run.

5. How to reflect on your team's project and work to create customized agile development process.

6. How to work productively with a small team (2-10 people).

7. Understanding how software fits into the world and the consequent responsibilities that software developers carry.

In retrospect, the instructor would have liked to emphasize one major learning goal not on this list: **How to problem solve when there is no right answer.** This goal is only attainable through constant evaluation of risk and modification to the design and program until the program is "complete". Each cycle that evaluates risk is completely dependent on the situation (programmer expertise, deadlines, customer requirements, budget) and therefore there is no right answer.

Modified Peer Instruction

Although the course was generally held in a typical Peer Instruction format [2], the instructor added an additional step that required the students to suggest answer choices for the questions as shown in Figures 11.1 and 11.2.

As an example, Figure 11.2 shows a clicker question before students have suggested

Figure 11.2. Example of a clicker question with no answer choices.

any answer choices. Because there is significantly less information presented in the question, and because of the nature of the questions (relating to risk management), the instructor would often review parts of the reading that were particularly ambiguous at the very beginning of lecture. This provided the students with the instructors interpretation given his expertise in the area, which prompted the students to think critically about the concepts of the course (e.g. productivity) and made them better prepared to suggest answer choices during clicker questions.

While students were suggesting answer choices a mini-discussion would arise between the instructor and the student suggesting an answer. The instructor would not discuss whether it was a good or correct answer (that was left for the group discussions), but rather would often ask students to clarify their answers. By first describing his interpretation of the reading, allowing students to choose the answer choices and requiring that they clarify the answers using software engineering terms, the instructor was engaging students in authentic practice.

Other Courses that use Peer Instruction

Rationale questions can engage students in the tradeoffs between different plausible solutions. Tradeoffs can be focused on broad concepts taught in introductory courses, such

Figure 11.3. Rationale question in a CS0 course.

as in Figure 11.3, or details taught in more advanced upper-division courses, such as in Figure 11.4.

Though traditional PI courses offer rationale questions that encourage students to debate *why* an answer is correct, or the *approach* taken to reach that answer, nearly all of the questions in a software engineering course are fundamentally rationale questions. The set of "correct" answers is vast and correctness completely depends on situation and rationale.

11.1.4 Student Experiences and Feedback

Due to the uniqueness of this course in it's content and pedagogy, it was of particular interest to understand student experiences. At the end of this course students were asked to fill out a survey reflecting on their experiences in the modified PI course. The survey was worth 1% of their final exam grade (only 0.3% of the total grade) and students were assured that the instructor would view none of their responses before grades were submitted. 78% of the students reported previously taking a computer science course that was taught with traditional PI, so the majority of students knew how this pedagogy was typically run.

Suppose you work on an embedded *multi-cycle MIPS processor* and your software team tells you that every program which executes has to go through memory and zero 1k bytes of data fairly often (averages 10% of ET). You realize you could just have a single instruction do this called zero1k (rs) which does:
M[rs] = 0 … M[rs+1020] = 0.
Your coworker thinks you are crazy. You reply?

Remember to ask about single-cycle
Answer - D

Selection	Crazy?	Reason
A	Yes	The complexity of such an instruction combined with no performance gain is silly.
B	Yes	The complexity of such an instruction combined with minimal performance gain (<5%) is silly.
C	No	The minimal performance gains (<5%) rationalize this simple instruction.
D	No	The significant performance gains (>5%) rationalize this complex instruction.
E	Maybe	None of the above.

Figure 11.4. Rationale question in an architecture course.

Clicker Questions

Discussion Topics. In other PI courses students are encouraged to discuss *why* they think an answer is correct and **why** they think the others are incorrect. In this course, however, the discussions are focused around situational context. Therefore the answer choices that students provide are not only meant to spark discussion about *why* they may or may not be correct, but also about the situation presented to them in the question and potentially come up with other answers as well. In a survey question that asked students what they discussed during lecture, the majority of students described discussing tradeoffs, definitions, and other possible answers. Though the instructor would like to see more students discussing real world examples and other situations not directly asked about in the question, the instructor might be able to better support student discussions in the future after having seen survey results.

From the data presented in Table 11.1 it can also be gathered that the students were still focusing much of their effort on coming to a correct answer. As a reminder, clickers were only graded on participation, not correctness, but it seems that students were still trying

Table 11.1. What students reported discussing during lecture.

Discussion Topics	Students
Discussed tradeoffs	52%
Discussed definitions	70%
Discussed other possible answers	60%
Discussed real world examples	39%
Discussed concepts/situations that were not directly asked about in the question	33%
Came to decision on the correct answer	63%
Could not decide what was the correct answer	43%
Discussed ambiguity of the question	3%

to find the correct answer so that they could prepare for the quizzes or final exam. Further interviews with the students would be required to understand what a typical discussion looked like.

Overall Lessons Learned

At the end of the survey students were asked, "Would you recommend this instructor, why or why not". 72% of the students said they would recommend the instructor, over the past 5 years and 28 instances of this course the average rating is 74%. But what are particularly interesting are the comments from those who said they would not recommend the instructor. Of the 28% of students who did not recommend the instructor, 15 students described some reason related to the peer instruction lecture. Comments were split into three main concerns:

1. **No "right" answer.** Eight students were frustrated that during the clicker questions the instructor did not provide a "right" answer. One student described this issue clearly *"There were many instances where it was not clear what the right answer should have been...In computer science, students should be given more clear right and wrong*

answers...Bottom line, things should be spelled out more black and white." Though this is a valid concern for students, one important goal of this course is to encourage the students to realize that designing software for customers on **large teams** is not "black and white". In the future, this concern will hopefully be mitigated by the instructor more clearly describing the goal of "how to problem solve when there is no right answer" at the beginning of the course.

2. **Not the material they expected.** Four students were unclear on what the goals were of the course. They were either frustrated that the instructor was focusing lecture time on discussing software engineering principles rather than Android development or unclear of what they were supposed to get out of lectures. This can also be mitigated with clear goals for each portion of the course and ensuring the students understand why those goals are important.

3. **Unclear clicker questions.** Three students brought up concerns that can be critical to the course's success. One student describes this issue clearly, *"Often time we discuss 4 choices that are basically the same in our eyes while he sees everything differently. It would be nice of him if he could give us a sense of direction before discussion."* Because this was the first time the instructor was teaching this course with this modified PI, he has learned a great deal about what kinds of responses the students will give and how to ensure all of the important points (not necessarily "correct" answers) are clear by the end of a class-wide discussion.

Although these are concerns that will be addressed in the next version of the course, it is reassuring that only 6% of the students found concerns with the modified PI. In retrospect, a modification might have improved student understanding of the clicker questions. Perhaps the instructor should provide the questions online after lecture and each group has to provide an answer (one either discussed during lecture, or another) and an explanation as to why they thought it was the correct answer; essentially a summary of what they did during their

group discussion. One student even suggested something similar in their feedback, which could potentially reduce the concern for a "right" answer as well.

11.1.5 Discussion

Peer Instruction has shown clear benefits in learning and student engagement in the past for many different courses. Though it has been a proven pedagogy, more data regarding Peer Instruction and its effectiveness in varying classrooms is still lacking. A call to the community is made to discuss what courses might be interesting to evaluate using PI, and how might PI need to be modified to best serve that course. Peer Instruction has been shown to be easily adoptable given materials and to improve learning and student engagement in large and small courses. If this pedagogy could serve more courses, it could potentially improve the learning of students over the entire curriculum.

11.1.6 Conclusion

In this paper a modification to Peer Instruction to best support the learning of software engineering practices and skills was explored. This modification was compared to other PI courses where learning goals included reasoning and rationale behind answer choices. It was clear that the software engineering questions are predominately rationale questions and are therefore fundamentally different than the typical "how" clicker questions. This modification was then evaluated using student feedback on an end-of-term, anonymous survey which showed that the majority of students reported enjoying the course. Based on the survey, students also reported a critical take-away: software design and development is a complex and difficult task, but after understanding risk management and mitigation, they should be able to evaluate and solve issues that arise as best as possible.

11.1.7   Acknowledgments

Section 11.1, in full, is a reprint of the material as it appears in CCSC-SW 2014. Esper, Sarah, Consortium for Computing Sciences in Colleges, 2014. The dissertation author was the primary investigator and author of this paper.

11.2   Developing Epistemic Frames: It is not Just About Mass Production of Learning

Learning at Scale is a trend that can reduce educational barriers, particularly in terms of accessibility. An issue with some of the current approaches is that they require the learner to develop their own motivations and discipline to properly engage with the materials. Often the intrinsic motivation of learners is disconnected from the desired learning goals of the course. This is sometimes mitigated with motivational videos throughout lessons, but is often lacking, which may be the cause for poor retention in MOOCs [136]. An effective way to help learners develop a more authentic motivation is to engage them in collaborative activities within a community of their peers. This can pose difficult for a scalable online learning environment, because personal interactions will be limited and the extrinsic motivators are more difficult to capture.

This work proposes two main contributions to the area:

- An immersive environment that provides a virtual community, and

- Suggestions for how to incorporate similar techniques into more traditional environments.

11.2.1   Background

An important feature of in-person education is the ability to form Communities of Practice [158]. Wenger poses the question *what if we adopted a different perspective, one that placed learning in the context of our lived experience of participation in the world?*

This is particularly difficult to do when the learners are not able to meet in person. Virtual Communities of Practice, like forums and wikis, offer an alternative community for learning at scale but are limited because participation is not real-time and is restricted to text. There are some people who are investigating the viability of virtual, video hangouts as part of scaled online courses, but these pose a variety of logistical problems.

The benefit of Communities of Practice is that they support the development of an epistemic frame [135]. An epistemic frame is the mindset of a given community which involves five specific elements: knowledge, skills, values, identity and epistemology. Communities of Practice are able to enculturate the members of the community in these elements through in-person activities and interactions. Traditional learning environment materials, such as lectures, solitary exercises and exams can help learners acquire the knowledge and skills of a discipline, but often are lacking in helping them develop the values, identity and epistemology of experts in the field.

Epistemic games are video games that simulate the training for a professional in a given field [135]. Shaffer suggests that epistemic games are a great approach to enculturating learners in the mindset of a discipline because they "are about *knowledge*, but they are about knowledge in action - about making knowledge, applying knowledge, and sharing knowledge. Epistemic games are rigorous, motivating, and complex because that's what characterizes the practices of innovation upon which they are modeled." [136]

CodeSpells is an epistemic game attempting to incorporate these elements to help learners develop the mindset of expert programmers. It attempts to engage learners not only in authentic programming activities, but also in valuing the importance of code exploration [55].

11.2.2 Developing an Authentic Mindset in CodeSpells

CodeSpells is a 3D interactive video game that is designed to engage novice programmers in Java Programming without the need of an instructor or tutor. It uses the metaphor

Figure 11.5. SpellBook given to players to support code exploration.

of magic by casting the player as a wizard and providing them with a SpellBook, where "spells" are really just Java programs [54].

There are three ways in which CodeSpells addresses the five elements of epistemic frames: in-game resources, non-player character interactions and a unique reward system. In this section, each of the five elements of epistemic frames will be addressed, alongside suggestions for how similar approaches could be taken in more traditional learning at scale environments.

Knowledge Acquisition

The primary in-game resource in CodeSpells is the SpellBook. Figure 11.5 shows an example Java program and explanation of the components of the program. This is essentially an in-game textbook that allows the players to begin to acquire knowledge about programming. The players are encouraged through the SpellBook and interactions with non-player characters to modify the spells and test them in the immersive environment. This also provides players with an opportunity to gain further knowledge about the effects of programs.

A similar approach that could be adopted in more traditional online environments

Figure 11.6. An NPC suggesting a problem for the player to solve with a program.

could be a set of resources that are small enough for the learner to engage with individually, but that can be integrated into larger problems. Media Computation is a great example where learners can modify small parts of the program to make small changes to an image when they are first learning, but soon integrate the different modifications to create amazing artwork as they understand more [68].

Skill Development

Players are presented with quests set forth by the non-player characters. These quests present the player with problems and suggested strategies for solving them. This part of the game truly allows the player to develop the skill of writing programs to solve specific requirements. An example quest can be seen in Figure 11.6.

In a more traditional online environment, students should have the opportunity to engage in more open ended problem sets. This does not necessarily have to be a guided exercise that is monitored and graded, but instead could be similar to exploratory homeworks, where students are guided through an example and encouraged to modify the program in

Figure 11.7. Badges that are earned for experimenting with spells.

authentic ways [56].

Values

CodeSpells uses a badge reward system. Players are encouraged to do a variety of activities within the game to earn different kinds of badges. For example, Figure 11.7 shows a set of badges that can be earned by exploring the code that is provided to the players. Code exploration should be valued for the sole purpose of exploring and understanding the code. A different set of badges that encourage players to solve very specific problems that require small, specific code modifications.

Furthermore, there are even more badges that can be earned for doing something creative with the spells, since this is not a scalable feature we will not go further into these types of badges.

Traditional online learning environments might approach this epistemic frame element by designing innovative reward systems. For example, instead of simply requiring

students to submit code, you could encourage them to design something different with the same basic functions of a program that they have already written. Using techniques such as peer assessment, you can then reward innovation and the ability for students to modify programs for a different purpose. Alternatively, you could have assignments that require students to find test cases that show a bug in the code, and submit the test case that caused the program to fail. This would encourage the students to recognize the value of testing code.

Identity

Embracing the identity of a discipline is a more difficult task than skill and knowledge acquisition when scaling online learning. CodeSpells imparts the identity of programmers through the interactions with the non-player characters. Not only are the NPCs giving the problems for the players to solve, but they are also suggesting other ways in which the players should interact with the code. Often times suggested code interactions do not lead to players earning any rewards, which is by design. Players soon recognize that to be able to solve more complex quests in the future, they must fully engage with the programs of earlier quests.

One way in which a more traditional online environment might attempt to impart the identity of a discipline would be to display motivational videos after major task completion. For example, when a student completes a programming assignment they could then watch how the instructor might modify the assignment slightly to do something completely different. This would encourage the student to view programs not only as a solution to a specific problem, but also as a new insight into solving other problems.

Epistemology

CodeSpells takes an interesting approach to the final element of epistemic frames; epistemology. In CodeSpells the later quests are presented as problems that need to be

solved. Players are then tasked with deciding how to solve the problem, which program they should start with, which pieces of code they should take from other programs, and how that would solve the problem. For example, in one of the final quests, the players have to write a spell that incorporates three other spells: set a crate on fire, make it move in a particular pattern, and make it teleport when it hits a particular area. Players must then decide which spell they will start with and identify which pieces of other spells they must use to be able to complete the program. Though within CodeSpells players are not asked to explicitly justify their choices, the nature of the game encourages them to make educated choices and does not support guess-and-check.

A more traditional environment might actually require the students to self-reflect on their choices in programming [21]. Self-reflection has been seen to help students engage with the material at a meta-cognitive level.

11.2.3   Conclusion

This work presents preliminary work in an alternative environment with specific suggestions on how they might be integrated into more traditional environments. A call to the community is made to discuss how the theory of epistemic frames might help inform learning at scale course and resource designs.

11.2.4   Acknowledgements

Chapter 12

Conclusions

Computer science is becoming a discipline that everyone, even children, wants exposure to; it's becoming a 21st Century Skill [42, 25, 31, 61]. This improvement in the last decade has caused student-teacher ratios in computer science to quickly grow to massive numbers [98]. Introductory computer science courses often have hundreds of students and only one instructor, extracurricular learning experiences for children have become selective, and massively open online courses have become popular [113].

Since all learning, computer science included, requires novices to learn skills and become enculturated (see Section 1.1), the quickly increasing student-teacher ratio reveals two major challenges addressed in this work:

- University classrooms are getting larger and the gap between novice and expert is rapidly widening, which reduces the effects of the learning experience, and

- Children do not have access to computer science teachers, and scalable options do not support both skills and culture development.

I hypothesized that:

Advances in our understanding of learning, the subject of computer science itself, and computing automation can integrate enculturation into novice learning experiences and scale it to match large student-teacher ratios.

To test this hypothesis, I designed, implemented and evaluated two major interventions; one for large, in-person, undergraduates and one for children.

12.1   Using Peer Instruction and Exploratory Home-works to Enculturate Novice Undergraduates

In Chapters 3 through 6, I presented a transformation of a large university classroom that showed improvements to students' learning and enculturation into computer science. In this course, students were more likely to be interactive in their learning, both in the classroom, and at home doing homeworkm compared to other pedagogies. UCSD's *CSE 3: Fluency in Information Technology* is a CS0-level course targeted at novice students who have no programming experience. Through this course, we contributed three major changes to scalable, in-person, undergraduate education:

1. **Peer Instruction is an effective pedagogy for enculturating students.** Chapter 4 evaluated students perceptions of their role in a PI computing course. Analysis of students self-reported activities demonstrated that the majority of students report interactive activities (arguing, discussing, explaining to peers) in PI classes whereas the majority report active activities (listening, taking notes) in their standard courses. This substantial shift in classroom activities unsurprisingly causes students to also change their perception of their role in a classroom. A grounded theory open-coding investigation of student perceptions found that students were positively affected by the shift in terms of class enjoyment, improved attendance and better attentiveness. They also reported improved meta-cognition skills facilitated by more frequent feedback and class discussions (see Section 4.5). Lastly, a few even reported trying to apply their self-reported improved learning behaviors from PI classes to other classes. This work has potential implications in identifying common desirable characteristics in student-centered learning environments and speaks to the issues raised in the STEM

retention literature.

2. **The Abstraction Transition Taxonomy can be used to categorize formative and summative assesements and find inconsistencies in course design.** We proposed that situated cognition theory [19], with its focus on learning for application in the "real world" and its focus on developing expertise within the context of a community is a useful lens for reconsidering programming instruction and perhaps computer science instruction more generally. A number of factors supported this idea. First, our community has norms that seem particularly challenging for outsiders to understand. It may be that part of our collective recruitment and retention issues (at least in the US) stems from the lack of attention to acculturation of new members. The AT Taxonomy defined 18 learning outcomes that explicitly address the required culture, activity, and tools required for members of the programming community which we believe needs to extend, at least minimally, to embrace everyone in modern, digital society.

3. **Out-of-Lecture activities can also scaffold novice undergraduates in skills and culture development.** Exploratory homeworks are tools that encourage students to actively engage in pre-class reading (see Section 6.4); this includes writing and debugging code as well as learning how to read technical, static texts. Active learning is a technique that has improved student learning in lecture and lab based settings [139, 128], but rarely are students asked to actively learn on their own, before lecture. Exploratory homeworks enhance already active textbooks, like the Alice textbook [46], as well as more expository textbooks. The biggest benefit from students actively reading before lecture is they come to lecture prepared to learn more complex concepts and constructs enabling instructors to delve more deeply in lecture. Students also develop authentic computing habits by actively participating in trial and error, predictions and reflections when learning a new programming language. Novices engaged with textual references in similar ways to how experts engage with them,

which encouraged a cultural shift from their typical textbook reading, to a computer science way of reading textbooks. We encourage the community to create exploratory homeworks for CS0 and CS1 courses using a variety of textbooks and report back on both effectiveness and challenges.

Furthermore, **Section 11.1** described how these contributions could be applied to higher level computing courses.

12.2 Enculturating and Teaching Children through a Scalable Serious Game: CodeSpells

In Chapters 7 through 10 I have presented CodeSpells, a serious game that integrates enculturation and skill in the learning experience for elementary-level students. CodeSpells uses the metaphor of magic and wizardry to scaffold children's learning, while engaging them in the culture of computer science through game-play. Each iteration of the design, implementation and evaluation of the game provided a piece of the overall CodeSpells contribution:

1. **CodeSpells can engage children similar to how current expert programmers were engaged when they first began, in a word, it is authentic**: We take the position that it is highly relevant to study noninstitutional learning in which young people teach themselves to program. We analyzed the origin stories of 30 individuals who eventually became successful in computer science. We identified five qualities that tended to occur across multiple origin stories and that can be tentatively posited to correlate with the sparking of lifelong passions for computer science.

   To more deeply study these qualities, we performed a laboratory study with 40 girls (ages 10 to 12) and analyzed their experiences according to the aforementioned five qualities. In just 1 hour, all 40 girls demonstrated at least three of the five qualities, showing that CodeSpells could enculturate novice programmers at scale. This allowed

us to refine our understanding of these qualities and even to observe a sixth quality further deepening the theoretical framework.

Our contributions are three-fold:

- We give the first theoretical framework for understanding the conditions under which the sparks of lifelong learning are sparked in non-institutional computer science learning environments.

- We demonstrate two novel methodologies for further examining non-institutional computer science learning. (The small amount of prior work that exists uses only in situ ethnographic methodologies.)

- We contribute a new vocabulary for discussing and designing tools for novices, using our theoretical framework to point out opportunities for improvement in both novice-friendly IDEs and educational games.

Ultimately, we take the position of Maloney et al. [101] that studying non-institutional learning is a means by which to understand learning in general. After all, non-institutional learning has a unique character. It is unstructured. It is creative. It is play. Also, as we saw in our laboratory study, the absence of values implicitly instilled in institutional settings (e.g. that syntax errors are bad) can lead to differences in behavior. As such, the study of non-institutional spaces may serve as an untapped resource for computer science educators a resource that can help inform how we structure our institutional spaces. The study of informal space can yield surprising new insights for computer science education interventions. These findings offer new ways to light fires and to keep them lit.

2. **The story and metaphor of a serious game affects the enculturation of children.** Our goal was to get novice programmers immersed in a programming environment that would lead to their ability to read, execute, understand, modify and create spells

(programs) with determination and a positive view of their ability. We believe we have begun to show how CodeSpells is a viable solution for encouraging novice programmers to invest in their programming as they do with video games.

We plan to run a larger study where we can compare how novice programmers engage with CodeSpells and Media Computation (which is used in our institutions CS1 course), as well as other environments such as Alice and Scratch that also offer immersive qualities. We plan to improve on our API as we measure learning gains in our future studies, but posit that the level of authenticity versus power, along with the metaphor of magic, engages novice programmers such that they can feel positive about their struggles when learning their first industry-standard language.

Our contributions are two-fold: We have presented here an API and environment that has:

- Allowed novice programmers to read, execute, understand, modify and write CS1-level Java programs within a 45- minute period.

- Immersed novice programmers into their programming environment so much so that they have developed a determination to solve problems and a positive outlook on programming challenges.

3. **Well-designed quests can effectively scaffold skills development.** Based on a study that shows students engagement with quests within CodeSpells, three key lessons to consider when creating in-game quests have been presented. To develop these guidelines, students were first observed during an exploratory study to understand the issues with the initial quests within CodeSpells. Additional quests that required more complex code edits were added and the expectation for the students to be able to complete them was given. A second set of students was then observed and their game-play was analyzed. It was observed that the expectation for significant code

changes encouraged students to engage with code in a more suitable way.

Chapter 10 shows that CodeSpells both enculturates and teaches children the skills of programming while playing and engaging with workbook-like resources and **Section 11.2** describes how this contribution could be applied to other environments as well.

12.3   Closing Remarks

Advancements in interactive learning have improved novice learning experiences, yet scaling these advancements is not straightforward. In this dissertation I have shown that integrating skills development and enculturation into *scalable* learning environments can effectively teach and enculturate novices in computer science. I have also suggested how my two contributions could be expanded to include different levels of student-expertise and different learning environments. It is an exciting time, and with these contributions we can see a positive progression towards making computer science, truly, the 4th 'R'.

Bibliography

[1] L.W. Anderson, D.R. Krathwohl, R.W. Airasian, K.A. Cruikshank, R.E. Mayer, P.R. Pintrich, J. Raths, and M.C. Wittrock. *A taxonomy for learning and teaching and assessing: A revision of Bloom's taxonomy of educational objectives*. Addioson Wesley Longman, 2001.

[2] Arduino. http://arduino.cc/.

[3] R. Bareiss and M. Radley. Coachin via cognitive apprenticeship. *SIGCSE*, 2010.

[4] William H. Bares, Luke S. Zettlemoyer, and James C. Lester. Habitable 3d learning environments for situated learning. *ITS*, pages 76–85, 1998.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 1999.

[6] T. Bell, J. Alexander, I. Freeman, and M. Grimley. Computer science unplugged: School students doing real computing without computers. *New Zealand Journal of Applied Computing and Information Technology*, 13(1):20–29, 2009.

[7] J. Bennedsen, M.E. Caspersen, and M. Kolling. *Reflections on the Teaching of Programming: Methods and Implementations*. Springer-Verlag, 2008.

[8] S. Bennett, K. Maton, and K. Kervin. The 'digital natives' debate: A critical review of the evidence. *British Journal of Educational Technology*, 39(5):775–786, 2008.

[9] J.B. Biggs and K.F. Collis. *Evaluating the quality of learning: The SOLO taxonomy (Structure of the Observed Learning Outcome)*. Academic Press, 1982.

[10] R.A. Bjork. Information-processing analysis of college teaching. *Educational Psychologist*, 1979.

[11] Jonathan D. Blake. Language considerations in the first year cs curriculum. *Journal of Computing Sciences in Colleges*, 26(6):124–129, 2011.

[12] J. Bonner and W. Holliday. How college science students engage in note-taking strategies. *Journal of Research in Science Teaching*, 43(8):786–818, 2006.

[13] M. Borrego, S. Cutler, M. Prince, C. Henderson, and J. E. Froyd. Fidelity of implementation of research-based instructional strategies (rbis) in engineering science courses. *Journal of Engineering Education*, 2013.

[14] J.D. Bransford. *How People Learn: Brain, Mind, Experience, and School*. Academy Press, 2000.

[15] John D. Bransford, Ann L. Brown, and Rodney R. Cocking. *How People Learn*. National Academy Press, 2000.

[16] G. Braught, L. Martin Eby, and T. Wahls. The effects of pair-programming on individual programming skill. *SIGCSE*, 2008.

[17] G. Braught, T. Wahls, and L. Martin Eby. The case for pair programming in the computer science classroom. *Transactions of Computing Education*, 11(1), 2011.

[18] Frederick P. Brooks. *The Mythical Man-Month (Anniversay Edt)*. Addison-Wesley Publishing Company, Boston, Ma, USA, 1995.

[19] J.S. Brown, A. Collins, and P. Duguid. Situated cognition and the culture of learning. *Educational Researcher*, 18(1):32–42, 1989.

[20] Amy Bruckman. "can educational be fun?". *Game Developer's Conference*, 199.

[21] R. Catrambone and Y. Mashiho. Acquisition of procedures: The effects of example elaborations and active learning exercises. *Learning and Instruction*, 16(2):139–153, 2006.

[22] M.T. Chi. Active-constructive-interactive: A conceptual framework for differentiating learning activities. *Topics in Cognitive Science*, 1(1), 2009.

[23] Codecademy. http://www.codecademy.com/learn.

[24] Codecademy for Schools. http://www.codecademy.com/schools/curriculum.

[25] Code.org. http://code.org/.

[26] CodeSpells. codespells.org.

[27] A. Collins, J.S. Brown, and S.E. Newman. *Knowing, learning, and instruction: Essays in honor of Robert Glaser*, chapter Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics, pages 453–494. Lawrence Erlbaum, 1990.

[28] Allan Collins, John Seely Brown, Susan E. Newman, and Lauren B. Resnick. Cognitive apprenticeship: Teaching the crafts of reading, writing, and mathematics. *Knowing, learning, and instruction: Essays in honor of Robert Glaser*, pages 453–494, 1989.

[29] Computer Science as a General Education Requirement. http://www.sixth.ucsd.edu/advising/requirements/index.html.

[30] Computer Science Education Statistics. http://www.exploringcs.org/resources/cs-statistics.

[31] Computer Science Teaching Association. www.csta.acm.org/Curriculum/sub/K12Standards.html.

[32] Computing Educators Oral History Project. www.ceohp.org.

[33] S. Cooper. Personal communication, 2010.

[34] Steve Cooper, Wanda Dann, and Randy Pausch. Alice: a 3d tool for introductory programming concepts. *Journal of Computing Sciences in Colleges*, 15(5):107–116, 2000.

[35] J. Corbin and A. Strauss. *Basics of Qualitative Analysis: Techniques and Procedures for Developing Grounded Theory*. Sage Publishing, 2007.

[36] M.W. Corney, D.M. Teague, A. Ahadi, and R. Lister. Some empirical results for neo-piagetian reasoning in novice programmers and the relationship to code explanation questions. *CRPIT*, 2012.

[37] S. Craig, M. Chi, and K. VanLehn. Improving classroom learning by collaboratively observing human tutoring videos while problem solving. *Journal of Educational Psychology*, 101(4):779–789, 2009.

[38] Tom Crick. Computing: The science of nearly everything.

[39] C. H. Crouch and E. Mazur. Peer instruction: Ten years of experience and results. *American Journal of Physics*, 69(9):970–977.

[40] C.H. Crouch, A.P. Fagen, J.P. Callan, and E. Mazur. Classroom demosntrations: Learning tools or entertainment? *American Journal of Physics*, 72, 2004.

[41] CS Principles Website. http://csprinciples.org.

[42] CSEd Week. http://csedweek.org/.

[43] Q. Cutts, M. Brown, L. Kemp, and C. Matheson. Enthusing and informing potential computer science students and their teachers. *SIGCSE Bulletin*, 39(3):196–200, 2007.

[44] Quintin Cutts, Sarah Esper, and Beth Simon. Computing as the 4th "r"; a general education approach to computing education. *ICER*, 2011.

[45] Quintin Cutts, Sarah Esper, and Beth Simon. The abstraction transition taxonomy: Developing desired learning outcomes through the lens of situated cognition. *ICER*, 2012.

[46] Wanda P. Dann, Steve Cooper, and Randy Pausch. *Learning to Program with Alice, Brief Edition*. Prentice Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[47] Michael de Raadt, Richard Watson, and Mark Toleman. Language of tug-od-war: industry demand and academic choice. *ACE*, 20:137–142, 2003.

[48] Matthew Dickerson. Multi-agent simulation and netlogo in the introductory programming concepts. *Journal of Computing Sciences in Colleges*, 2011.

[49] Carol Dweck. *Mindset: The New Psychology of Success*. Ballantine Books, 2007.

[50] C.S. Dweck. *Self-Theories - Their role in Motivation, Personality and Development*. Philedelphia: Taylor and Francis / Psychology Press, 1999.

[51] T. Eberlein, J. Kampmeier, V. Minderhout, R.S. Moog, T. Platt, P. Varma-Nelson, and H.B. White. Pedagogies of engagement in science. *Biochemistry and Molecular Biology Education*, 36:262–273, 2008.

[52] Magy Seif El-Nasr and Brian K. Smith. Learning through game modding. *Computing Entertatinment*, 4(1), 2006.

[53] K.A. Ericsson. *Cambridge handbook of expertise and expert performance*, chapter The influence of experience and deliberate practice on the development of superior expert performance, pages 685–706. Cambridge University Press, 2006.

[54] Sarah Esper, Stephen R. Foster, and William G. Griswold. Codespells: Embodying the metaphor of wizardy for programming. *ITiCSE*, 2013.

[55] Sarah Esper, Stephen R. Foster, and William G. Griswold. On the nature of fires and how to spark them when you're not there. *SIGCSE*, 2013.

[56] Sarah Esper, Beth Simon, and Quintin Cutts. Exploratory homeworks: An active learning tool for textbook reading. *ICER*, 2012.

[57] Sarah Esper, Samantha R. Wood, Stephen R. Foster, Sorin Lerner, and William G. Griswold. Codespells: How to design quests to teach java concepts. *Consortium for Computing Sciences in Colleges*, 2014.

[58] K. Falkner and E. Palmer. Developing authentic problem solving skills in introductory computing classes. *SIGCSE*, 2009.

[59] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2001.

[60] Stephen R. Foster, Sarah Esper, and William G. Griswold. From competition to metacognition: Designing diverse, sustainable educational games. *CHI*, 2013.

[61] National Science Foundation. Stem-c partnerships: Computing education for the 21st century (stem-cp: Ce21). Program Solicitation, 2014.

[62] U. Fuller, C.G. Johnson, T. Ahoniemi, D. Cukierman, I. Hernan-Losada, J. Jacksova, E. Lahtinen, T.L. Lewis, D. Thompson, C. Riedesel, and E. Thompson. Developing a computer science-specific learning taxonomy. *SIGCSE Bulletin*, 39(4):152–170, 2007.

[63] J.D. Gaffney, A.L.H Gaffney, and R.J. Beichner. Do they see it coming? using expectancy violation to gauge the success of pedagogical reforms. *Physical Reciew Special Topics - Physics Education Research*, 6(1), 2010.

[64] E. Galliot and E. Bache. Coding Dojo - http://codingdojo.org.

[65] Daniel D. Garcia, Brian Harvey, and Luke Segars. Cs principles pilot at university of california, berkeley. *Inroads*, 2012.

[66] Google Practicum. http://www.google.com/about/careers/students/.

[67] National Guidelines. Information and communicatiosn technology: 5-14, 2000.

[68] Mark Guzdial. A media computation course for non-majors. *SIGCSE-Bull*, 35(3):104–108, 2003.

[69] S. Hadjerrouit. Constructivism as guiding philosophy for software engineering education. *SIGCSE Bulletin*, 37(4):45–49, 2005.

[70] R. R. Hake. Interactive-engagement versus traditional methods: A six-thousand-student survey of mechanics test data for introductory physics courses. *American Journal of Physics*, 66(1):64–74.

[71] L. Hakulinen. Using serious games in computer science education. *Koli Calling*, pages 69–78, 2011.

[72] P. Henriksen and M. Kolling. Greenfoot: Combining object visualisation with interaction. *Companion to 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, pages 78–82, 2004.

[73] Z. Herpic, D. Zollman, and N.S. Rebello. Comparing students' and experts' under-standingof the content of a lecture. *Journal of Science Education and Technology*, 16(3), 2007.

[74] Michael Hewner and Mark Guzdial. Attitudes about computing in postsecondary graduates. *ICER*, 2008.

[75] Jason Hong. The use of java as an introductory programming language. *Crossroads*, 4(4):8–13, 1998.

[76] Michael S. Horn, Erin Treacy Solovey, R. Jordan Crouser, and Robert J.K. Jacob. Comparing the use of tangible and graphical programming languages for informal science education. *CHI*, pages 975–984, 2009.

[77] C.D. Hundhausen, N.H. Narayanan, and M.E. Crosby. Exploring studio-based instructional models for computing education. *SIGCSE*, pages 393–396, 2008.

[78] Janusz Jablonowski. A case study in introductory programming. *CompSysTech*, 2007.

[79] C.G. Johnson and U. Fuller. Is bloom's taxonomy appropriate for computer science? *Baltic Sea*, pages 120–123, 2006.

[80] R. Lorch Jr., E. Puglez Lorch, and M. Klusewitz. College students' conditional knowledge about reading. *Journal of Educational Psychology*, 85(2):239–252, 2009.

[81] Ken Kahn. Drawings on napkins, video-game animation, and other ways to program computers. *Communications of the ACM*, 39(8):49–59, 1996.

[82] J. Karpicke and J. Blunt. Retrieval practice produces more learning than elaborative studying with concept mapping. *Science*, 331(6018):772–775, 2011.

[83] Caitlin Kelleher and Randy Pausch. Lowering barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, 2005.

[84] R. Khaled and G. Ingram. Tales from the front lines of a large-scale serious game project. *CHI*, pages 69–78, 2012.

[85] D. Kolb. *Experiential learning: experience as the source of learning and development*. Prentice Hall, Inc., 1984.

[86] Michael Kolling. The greenfoot programming environment. *Transactions of Computing Education*, 10(4):14:1–14:21, 2010.

[87] Jean Lave and Etienne Wenger. *Situated Learning: Legitimate peripheral participation*. Cambridge University Press, 1991.

[88] Michael J. Lee and Andrew J. Ko. Personifying programming tool feedback improves novice programmers' learning. *ICER*, 2011.

[89] M.J. Lee, A.J. Ko, and I. Kwan. In-game assessments increase novice programmers' engagement and level completion speed. *ICER*, pages 153–160, 2013.

[90] Lego Mindstorms. http://mindstorms.lego.com.

[91] C. Lewis, N. Titterton, and M. Clancy. Developing students' self-assessment skills using lab-centric instruction. *Journal of Computing Sciences in Colleges*, 26(4):173–180, 2011.

[92] Colleen M. Lewis and Niral Shah. Building upon and enriching grade four mathematics standards with programming curriculum. *SIGCSE*, 2012.

[93] R. Lister, B. Simon, E. Thompson, J. Whalley, and C. Prasad. Not seeing the forest for the trees: novice programmers and the solo taxonomy. *ITiCSE*, 2006.

[94] M. Lopez, J. Whalley, P. Robbins, and R. Lister. Relationships between reading, tracing and writing skills in introductory programming. *ICER*, 2008.

[95] George Lukas. Uses of the logo programming language in undergraduate instruction. *ACM*, 2:1130–1136, 1972.

[96] Mary Margaret Lusk and Robert K. Atkinson. Animated pedagogical agents: Does their degree of embodiment impact learning from static or animated worked examples? *Applied Cognitive Psychology*, 2007.

[97] Andrew Luxten-Reilly and Paul Denny. A simple framework for interactive games in cs1. *SIGCSE Bulletin*, 41(1):216–220, 2009.

[98] Ryan Lytle. Computer science continues growth on college campuses. *US News*, 2012.

[99] John Van Maanen. *Breaking in: Socialization to work*, pages 67–130. Rand-McNally College Publishing, 1976.

[100] Matthew B. MacLaurin. The design of kodu: a tiny visual programming language for children on the xbox. *SIGPLAN*, 46(1):241–246, 2011.

[101] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *TCE*, page 15, 2010.

[102] John H. Maloney, Kylie Peppler, Yasmin B. Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. *SIGCSE*, pages 367–371, 2008.

[103] Linda Mannila and Michael de Raadt. An objective comparison of languages for teaching introductory programming. *Baltic Sea*, pages 32–37, 2006.

[104] E. Mazur. *Peer Instruction: A User's Manual.* Prentice Hall, Inc., 1997.

[105] Eric Mazur. Farewell, lecture? *Science*, 323(5910):50–51, 2009.

[106] C. McDowell, L. Werner, H. Bullock, and J. Fernald. The impact of pair programming on student performance of computer science related majors. *ICSE*, 2003.

[107] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Learning computer science concepts with scratch. *ICER*, 2010.

[108] Microsoft Explorers. http://careers.microsoft.com/careers/en/us/university-programs.aspx.

[109] A. Mitchell, H. C. Purchase, and J. Hamer. Computing science: What do pupils think? *ITiCSE*, 2009.

[110] Briana B. Morrison, Brian Dorn, and Mark Guzdial. Measuring cognitive load in introductory cs: Adaptation of an instrument. *ICER*, 2014.

[111] B. Moskal, D. Lurie, and S. Cooper. Evaluating the effectiveness of a new instructional approach. *SIGCSE*, pages 75–79, 2004.

[112] S. Papert. *Mindstorms: Children, Computers and Powerful Ideas*. Basic Books, 1980.

[113] Laura Pappano. The year of the mooc. *New York Times*, 2012.

[114] Kylie A. Peppler and Yasmin B. Kafai. From supergoo to scratch: exploring creative digital media production in informal learning. *Learning, Media and Technology*, 32(2):149–166, 2007.

[115] L. Porter, S. Garcia, J. Glick, A Matusiewicz, and C. Taylor. Peer instruction in computer science at small liberal arts colleges. *ITiCSE*, 2013.

[116] Leo Porter, Cynthia Bailey-Lee, and Beth Simon. Halving fail rates using peer instruction: a study of four computer science courses. *SIGCSE*, 2013.

[117] Leo Porter, Cynthia Bailey-Lee, Beth Simon, Quintin Cutts, and Daniel Zingaro. Experience report: A multi-classroom report on the value of peer instruction. *ITiCSE*, pages 138–142, 2011.

[118] Leo Porter, Cynthia Bailey-Lee, Beth Simon, and Daniel Zingaro. Peer instruction: do students really learn from peer discussion? *ICER*, 2011.

[119] Leo Porter and Beth Simon. Retaining one-third more majors with a trio of instructional best practices in cs1. *SIGCSE*, 2013.

[120] M. Prenksy. Digital natives, digital immigrants. *On the Horizon*, 9(5):1–6, 2001.

[121] Mitchel Resnick, John Maloney, Andres Monroy-Hernandez, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. Scratch: programming for all. *Communications of the ACM*, 2009.

[122] Clare Richards. Teach the world to twitch: An interview with marc prensky, ceo and founder games2train.com. *FutureLab*, 2003.

[123] RoboCode. http://robocode.sourceforge.net.

[124] J.D. Karpickeand H.L. Roediger. The critical importance of retrieval for learning. *Science*, 319(5865), 2008.

[125] J.K. Rowling. Harry potter franchise.

[126] RPG Creator. http://rpgcreator.net.

[127] K. Sanders, B. Richards, J. Mostrom, V. Almstrum, S. Edwards, S. Fincher, K. Gunion, M. Hall, B. Hanks, S. Lonergan, R. McCarney, B. Morrison, J. Spacco, and L. Thomas. Dcer:sharing empirical computer science education data. *ICER*, 2008.

[128] Nicolai Scheele, Anja Wessels, Wolfgang Effelsberg, Manfred Hofer, and Stefan Fried. Experiences with interactive lectures - considerations from the perspective of educational psychology and computer science. *CSCL*, 2005.

[129] Donald A Schon. *The Reflective Practitioner: How Professionals Think in Action*. Ashgate Publishing Limited, 1995.

[130] Daniel L. Schwartz and Taylor Martin. Inventing to prepare for future learning: The hidden efficiency of encouraging original student production in statistics instruction. *Cognition and Instruction*, 22(2):129–184, 2004.

[131] Scratch Website. http://scratch.mit.edu/.

[132] Julian Sefton-Green. Literature review in informal learning with technology outside school. *NESTA Future Lab*, 7, 2004.

[133] E. Seymour and N.M. Hewitt. *Talking about leaving: Why undergraduates leave the sciences*. Westview Press, 1997.

[134] David W. Shaffer. Epistemic frames for epistemic games. *Computers and Education*, 46:223–234, 2006.

[135] D.W. Shaffer. Epistemic frames and islands of expertise: Learning from infusion experiences. *ICLS*, pages 473–480, 2004.

[136] D.W. Shaffer and J.P. Gee. Before every child is left behind: How epistemic games can solve the coming crisis in education. *Academic ADL Co-Lab*, 2005.

[137] B. Simon, M. Kohanfars, J. Lee, K. Tamayo, and Q. Cutts. Experience report: peer instruction in introductory computing. *SIGCSE*, 2010.

[138] Beth Simon, Sarah Esper, and Quintin Cutts. Experience report: an ap cs principles university pilot. Technical Report CS2011-0965, University of California, San Diego, 2011.

[139] Beth Simon, Julian Parris, and Jaime Spacco. How we teach impacts student learning: peer instruction vs. lecture in cs0. *SIGCSE*, pages 41–46, 2013.

[140] M. Smith, W. Wood, W. Adams, C. Wieman, J. Knight, N. Guild, and T. Su. Why peer discussion improves student performance on in-class concept questions. *Science*, 323, 2009.

[141] Royal Society. Current ict and computer science schools - damaging to uk's future economic prospects? *Press Release*.

[142] N. Solntseff. Programming languages for introductory computing courses: a position paper. *SIGCSE*, pages 119–124, 1978.

[143] Spring Real-Time Strategy Engine. http://springrts.com.

[144] Constance A. Steinkuehler. Learning in massively multiplayer online games. *ICLS*, 2004.

[145] Anselm Strauss and Juliet Corbin. *Basics of qualitative research: grounded theory procedures and techniques*. Sage Publishing, 1990.

[146] SumDog. www.sumdog.com, 2014.

[147] Sureyya Tarkan, Vibha Sazawal, Allison Druin, Evan Golub, Elizabeth M. Bonsignore, Greg Walsh, and Zeina Atrash. Toque: designing a cookie-based programming language for and with children. *CHI*, pages 2417–2426, 2010.

[148] S. Teasley. *Talking About Reasoning: How Important is the Peer in Peer Collaboration?*, pages 361–384. Springer-Verlag, 1997.

[149] Allison Elliott Tew, W. Michael McCracken, and Mark Guzdial. Impact of alternative introductory courses on programming concept understanding. *ICER*, 2005.

[150] The College Board. AP computer science: Principles project overview.

[151] Nathaniel Titterton, Colleen M. Lewis, and Michael J. Clancy. Experiences with lab-centric instruction. *Computer Science Education*, 20(2):79–102, 2010.

[152] S. Tobias. *They're not dumb, they're different: stalking the second tier*. Research Corporation, 1990.

[153] S. Turkle. From powerful ideas to powerpoint. *Journal of Research into New Media Technologies*, 9(2):19–28, 2003.

[154] C. Turpen and N.D. Finkelstein. Not all interactice engagement is the same: Variations in physics professors' implementation of peer instruction. *Physical Review Special Topics - Physics Education Research*, 5(2), 2009.

[155] UCSD AP CS Principles Pilot. http://www.csprinciples.org/home/pilot-sites/ucsd.

[156] UCSD CSE PhD Program. http://cse.ucsd.edu/node/34.

[157] W.A. Waller. A framework for cs1 and cs2 laboratories. *SIGCSE*, 1994.

[158] Etienne Wenger. *Communities of Practice: Learning, Meaning, and Identity*. Cambridge University Press, 1998.

[159] J. L. Whalley and R. Lister. The bracelet. *ACE*, 2009.

[160] Cameron Wilson, Leigh Ann Sudol, Chris Stephenson, and Mark Stehlik. Running on empty: The failure to teach k-12 computer science in the digital age. *CSTA*, 2010.

[161] J. Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

[162] Xenophon. *Conversations of Socrates*. Penguin Classics, 1990.

[163] Stuart Zweben. Computing degree and enrollment trends. Technical report, Computing Research Association, 2013.