**The Dissertation Committee for Timothy Tung-Ming Yuen Certifies that this is the approved version of the following dissertation:**

**Using a game template as a multimedia-based cognitive tool to facilitate novices' conceptual understanding of object-oriented programming**

**Committee:**

Min Liu, Supervisor

Edmund Emmer

Guadalupe Carmona-Dominguez

Mary Lee Webeck

Roger Priebe

# Using a game template as a multimedia-based cognitive tool to facilitate novices’ conceptual understanding of object-oriented programming

**by**

**Timothy Tung-Ming Yuen, BS; MS**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

**December 2008**

## Dedication

這本論文要獻給我的父母,

阮文偉和阮陳惠琴,

以及他們遺留下的無止盡犧牲.

For my parents,

Man Wai and Wing Hung Yuen,

and their never-ending sacrifices

that will continue to live on.

## Acknowledgements

I, Timothy T. Yuen, humbly admit that I could not have succeeded in this PhD program without the help and guidance from a vast support network of mentors, colleagues, friends, and family; and the countless and amazing opportunities afforded to me during my time at UT. Just writing a "brief" description on how each person or entity contributed to this achievement would require another five chapters, at least. Therefore, only their names are listed below in alphabetical order.

*Association for Computing Machinery*
*Linda L.G. Brown*
*Arnold Cano*
*Dr. Lupita Carmona*
*Edmund Danyal*
*Sonia Soto Dominguez*
*Dr. Ed Emmer*
*Dottie Hall*
*Enspire Learning*
*iD Tech Camps*
*Linda Guardia Jackson*
*Alana Lee*
*Dorothy Lee*
*Dr. Min Liu*
*Joy Meserve*
*Dr. Mark Luetzelschwab*
*Dr. Pam Bell Morris*
*Trevor Anthony Pickering*
*Dr. Roger Priebe*
*Linda Prieto*
*Dr. Todd C. Reimer*
*Mike Scott*
*SIGCSE Doctoral Consortium 2007*
*SIGCSE Doctoral Consortium 2008*
*Haley Steele*
*Dr. La Vergne Lestermeringolo Thatch*
*Dr. Paul Toprac*
*Carlos Tovar*
*Vaughn Gross Center for Reading and Language Arts*
*Dr. Mary Lee Webeck*

Lastly, I wish to express my appreciation and gratitude to my friends in Austin who became my family away from home, and to my family and friends in Southern California for making every visit home a homecoming celebration.

**Using a game template as a multimedia-based cognitive tool to facilitate novices' conceptual understanding of object-oriented programming**

Publication No._____________

Timothy Tung-Ming Yuen, PhD
The University of Texas at Austin, 2008

Supervisor: Min Liu

This study examined how a multimedia-based cognitive tool (MCT) facilitates novices' conceptual understanding of object-oriented programming (OOP). The tool used in this study was CSNüb, a game template created in Adobe Flash. The MCT design framework guided CSNüb's design. The MCT design framework was synthesized from literature on constructivist, multimedia, and motivation learning theories and computer-based cognitive tool design principles. Students worked with CSNüb to develop a simple role-playing game (RPG). Through clinical interviews and process tracing methods, it was found that CSNüb affected novice computer science students' conceptual understanding of OOP through five cognitive processes and factors: cognitive *disequilibrium* evoked through multimedia-based feedback, *exploring* for resources that *scaffold* understanding, changing the level of *awareness* of the "bigger picture" and ability for higher-level thinking, and consistent *refinement* of solutions and mental models within the problem space. The five cognitive processes and factors were found to be the result of three levels of interaction with CSNüb. At the Tool Level, students received conflicting information, generally through multimedia-based feedback from the CSNüb, which placed students in states of disequilibrium. At the Interaction Level, students interacted with the CSNüb to resolve their disequilibrium through exploring resources within the tool and refining their solution. They were able to experiment and test out their understanding on OOP. At the Cognitive Level, students used the resources as cognitive scaffolds found through exploration, which in turn, increased the degree of awareness and influenced the level at which they understood the object-oriented system. The five cognitive processes and factors through the three levels of interaction were formed into one model—the MCT Interaction Model (MCT-IM)—as a general explanation for how MCTs, such as CSNüb, affects novice students' conceptual understanding.

## Table of Contents

## List of Tables

## List of Figures

## List of Illustrations

# Chapter 1: Object-Oriented Programming and Computer Science Education

## MY FIRST COMPUTER SCIENCE CLASS: A PERSONAL VIGNETTE

My college career began as an Information and Computer Science major at the University of California at Irvine in the fall quarter of 1996. The first class in the sequence was the notorious ICS 21 – Introduction to Computer Science I, where the curriculum was infamously demanding, the workload heavy, and the pass-rate low, according to all ICS majors. It was a 6-unit class, whereas most other courses were only 4 units. ICS 21 adopted an "objects-first" approach to programming in which object-oriented programming (OOP) is taught at the beginning of the computer science curriculum. Unfortunately, the object-oriented programming paradigm was foreign to me, even though I had been programming in BASIC since elementary school, since BASIC programming is a procedural language in which each line of code is executed in a systematic and orderly manner.

In ICS 21, the instructor taught us how to think in terms of objects. He provided various examples of how to write a "class": code that defines the properties and behaviors of an object. To decrease the abstractness of the concept, the instructor used real-world examples to explain the design of a "class." I specifically remember him using a car and its mechanical parts as a metaphor for a class and its properties, respectively. I left the lecture hall having some understanding about classes and objects, but I remained confused as to how that applied to what I already knew about programming and when I would use this newly acquired concept. Later in the quarter, when my study group and I were attempting to decipher our first OOP assignment, we struggled to connect our prior computer science knowledge with the new programming information and to apply it to the task at hand; that is, how the conceptual aspect of OOP applied to the programming. I was not used to thinking of the problems or solutions in terms of objects.

My struggle with learning OOP was not a unique experience, as many other students have also struggled with it. OOP has also been problematic for instructors who have to teach this new way of thinking and problem-solving (Dale, 2006; Eckerdal, 2006; Hadjerrouit, 1999; Kölling, 1999; Sicilia, 2006). This has led to the debate in the CS education community on whether or not to teach "objects first" (Bruce, 2005; Lister et al., 2006). Before delving into the problems of teaching and learning OOP, however, it is necessary to explore what object-oriented programming means.

## OBJECT-ORIENTED PROGRAMMING

Object-oriented programming (OOP) is a paradigm in which solutions follow a specific design and structure. Code is written and organized into separate, interactive, reusable, and manageable parts known as objects. Organizations depend on the functionality of real-world objects that the code must emulate. "In fact, the main difference between object-oriented design and 'ordinary' programming is that in ordinary programming we are limited to predefined abstractions, while in object-oriented design we can define our own abstraction" (Ben-Ari, 1996, p. 265). OOP has three characteristics that distinguish it from other paradigms: encapsulation, inheritance, and polymorphism (Ben-Ari, 1996; Sebesta, 1999; Smith, 1991).

*Encapsulation* is also known as data or information hiding (Smith, 1991). An object's state is defined by its properties. Objects have their own behaviors, which can change their state. The pieces of code that affect objects' behaviors are organized into methods (also known as functions). The properties and methods are encapsulated together into a single object. Encapsulation also implies that an object's properties and methods are hidden from other objects. However, objects can provide interface methods through which other objects can access and modify their hidden parts when these objects interact (Ben-Ari, 1996).

*Inheritance* allows for objects to be extended and become more specified. For example, consider a Rectangle object. The properties include length and width. The object also includes behaviors that act on its properties: these methods can calculate the object's area and perimeter. Suppose another programmer wanted to write code for another object: a Parallelogram. Since it is very similar to a Rectangle, that programmer could reuse the Rectangle object and add extensions to the code. For instance, a Parallelogram might require a height property and the area method would need to be reformulated to compute the correct area. In this situation, the programmer need not write code for a Parallelogram object from scratch. Inheriting from other objects results in a hierarchical relationship between specific objects. Inheritance is a powerful addition to the OOP paradigm (Pratt & Zelkowitz, 1996).

In the previous example of a Parallelogram object inheriting from a Rectangle object, the former inherited all the properties and methods of the latter. As mentioned, when extending an object, the programmer can add or ignore properties and methods from the original object. *Polymorphism* allows for the programmer to override the methods as well and rewrite them for that particular object. For instance, the Parallelogram does not—and should not—use the same area method as the Rectangle's area method since calculating the area for both shapes differ. Instead, the programmer overrides Rectangle's original area method. When the area method is called, the compiler is able to determine which area method is appropriate (the new Rectangle area method or the original Parallelogram area method): the method definition bound to the actual class being used is called. This process, referred to as dynamic binding, is associated with polymorphism (Sebesta, 1999).

These contrived examples serve to illustrate the concept of OOP in a simple form, but when presented to a non-CS audience, it can be very confusing and abstract. As a current trend in programming, many undergraduate CS curricula opt for an objects-first approach in which OOP is taught at the very beginning. “The principal advantage in the objects-first strategy is the early exposure to object-oriented programming, which has become increasingly important in both academia and industry” (SIGCSE, 2001, §7.6.2). Within the context of a single course, Bruce (2005) defined the “traditional route” as teaching procedural coding in the first two-thirds of the course with OO concepts towards the end. The “objects-early” route would deemphasize classical procedural constructs, such as control structures and loops, and focus on the fundamental concepts of OOP. In 2001, the Special Interest Group for Computer Science Education (SIGCSE) of the Association for Computing Machinery (ACM) established guidelines for undergraduate computer science courses. SIGCSE’s (2001) guidelines for an introductory computer science course taking an objects-first approach include the following: the history of computing, ethics, computer systems, the object-oriented paradigm, fundamental programming constructs, data structures, algorithms, problem solving, and programming languages.

## Teaching OOP

Dale (2006) surveyed computer science educators on the ACM SIGCSE listserv asking what they felt were the most difficult topics to teach. Four categories emerged from her data: problem-solving and design skills, general programming topics, object-oriented constructs, and student maturity and self-discipline issues. The three categories pertaining to programming/computing are discussed here: problem solving and design, object-oriented constructs, and general programming topics.

The first category, problem solving and design, includes algorithms and abstraction. In order for students to write object-oriented code (as well as non object-oriented code), they must have a clear understanding of abstraction, design, and problem-solving skills. The second category, object-oriented constructs, includes any topics within the OOP paradigm (e.g., encapsulation, polymorphism, inheritance). The third category, general programming topics, includes topics such as parameter passing, arrays, recursion, pointers, loops, files, conditionals, and testing. Most of these are fundamental programming constructs also necessary in algorithmic design. As noted by the SIGCSE (2001), these topics should be covered in an introductory computer science course, which may be made more difficult when presented with another layer of complexity with OOP.

When I surveyed the same listserv in 2006 as to what computer science educators felt were the most difficult concepts for their students to understand, the top four concepts reported were similar: object-oriented programming, algorithms, functions and parameter passing, and recursion. Again, OOP was found to be a difficult concept for students to understand. Lahtinen, Ala-Murka, and Järvinen's (2005) survey of university students and teachers found similar difficult concepts that include recursion, pointers and references, abstract data types, error handling, and using libraries. An abstract data type (ADT is comprised of a set of data and a set of operations that can be enacted upon that data (Ben-Ari, 1996; Pratt & Zelkowitz, 1996). An important relationship to OOP is that the data set within ADTs, generally written as classes, are encapsulated such that the values cannot be changed except through the methods provided (Pratt & Zelkowitz, 1996).

If the most difficult topics for instructors to teach are also challenging for students to understand, then instructional assistance must be provided to both the instructors and students. This problem illustrates a need to improve instruction. My position is that instruction can always be changed and improved to include all students interested in computing. They may not all have the desire to become software engineers or developers, but they should be given the opportunity and necessary scaffolding to learn.

Kölling (1999) discussed several problems with teaching OOP. First, when moving from a procedural language to an object-oriented language, it can take a programmer 6 to 18 months to make the shift. Bruce (2005) echoed the suggestion of Borne Stroustrup, a designer of C++, that it can take 18 months to transition from the procedural C to its OOP counterpart, C++. Lister et al. (2006) found no additional evidence for this claim, but some transition time is needed. Next, since OOP is a way of thinking, Kölling believed it is necessary to start teaching with an object-oriented approach if OOP is to be taught at all. Some courses adopt a procedural style of programming first and then teach OOP later, which Kölling regarded as a "serious mistake." If anything, he suggests that teaching OOP should precede procedural programming since the latter is easily incorporated.

Another issue that Kölling found relevant dealt with the availability of the teaching tools:

> In our view, it is not object-orientation in principle that causes the problems, but the tools available to teach it. […] In short: in our view the reason for all the trouble is that the wrong languages and environments are being used. (1999, p. 2)

The languages used and the development environments can be too complex, especially for novices. According to Kölling, some of the widely used languages present some problems in teaching. He listed C++, the first language I learned in an objects-first course, as one of the worst choices for an OOP language:

> C++ fails to meet almost all requirements on our list. It is a hybrid language that does not support the concepts in a clean way, has a highly redundant set of constructs, an unsafe type system and a highly complex execution model. (p. 9)

In addition, C++ gives programmers the option to write code that can manipulate data on the bit level (e.g., bit-shifting), which is considered too low level. OOP requires higher-level thinking: problem solving requires students to think in terms of objects. Moreover, the programmer has to deal with unnecessary maintenance such as allocation and deallocation of memory when that should be the operating system's job. Lahtinen, Ala-Mutka, and Jarvinen (2005) found that "the biggest problem of novice programmers does not seem to be the understanding of basic concepts but rather learning to apply them" (p. 17). If the implementation of knowledge was a factor in learning as Lahtinen et al. suggest, then the tool in which that implementation is facilitated must not impede the learning process. With respect to development environments, Microsoft Visual Studio is a popular integrated development environment used both in academia and industry. Thus, beginning students will only use a small fraction of what this product actually offers. It is also easy for students to find in this environment too many choices in terms of tools, settings, messages, and so forth. Such an environment may overwhelm students. Finding an appropriate program language and development environment for teaching is one step toward improving the teaching of OOP.

## Students' Perception of OOP

As mentioned before, OOP requires a specific way of thinking and planning in terms of "objects." In addressing challenges to teaching and learning, it is important to have an overview of the different ways students understand and learn OOP. Eckerdal (2006) conducted a phenomenographic study to find the different ways novices experience object-oriented programming. She interviewed 45 students who were taking a required introductory programming course in Java in a Swedish university on their understanding of programming and OOP concepts.

Eckerdal focused on the concepts of objects and class, which are central to the OOP paradigm. An object is the abstract representation and instantiation of a class, which provides a defined template in code. Eckerdal categorized three perspectives that students assume in understanding objects and classes. The "code perspective" refers to the structuring, organizing, and modularizing of code that is aligned with the notion of encapsulation. The "user and result perspective" implies that OOP simplifies the programmer's work due to the affordances of code reusability (e.g., instantiating many objects from a single class template, or inheriting all the properties and methods from another class to avoid writing an entire class definition). OOP also offers guidance for solving problems and designing solutions (e.g., finding an object-oriented solution), and, in fact, Eckerdal and Berglund (2005) found that learning to program is experienced by students as a way of thinking. The "reality perspective" shows a connection between the real world and programming (Eckerdal, 2006). Eckerdal regarded this perspective as reflecting a deep and rich understanding. Not only did using objects and classes structure programming and thinking, students were also able to connect them to real world representations and examples. There was a sense that students have internalized these concepts and were able to explain them in familiar terms.

## CHALLENGES IN COMPUTER SCIENCE EDUCATION

Though OOP is a recurring theme in CS education research (Bennedsen & Caspersen, 2006; Eckerdal, 2006; Eckerdal & Berglund, 2005; Ferguson, 2003; Kölling, 1999; Lawhead et al., 2003), it is also useful to take a broader look at the general challenges facing the computing education field, and not just content specific issues.

A conference hosted by the British Computer Society in 2004 categorized some of the "grand challenges" in computing education. In this case, computing refers to any field that "involves the technical aspects of computing systems" which also encompasses computer science. A "grand challenge" should seek to improve computing education, motivate and engage people in the computing fields, and cause widespread changes to the entire computing community. McGettrick et al. (2004) identified seven grand challenges that computing education faced. This collection of grand challenges can be interpreted as a needs assessment for computing instruction.

1. Perception of Computing: improving the public face of computing by displaying it in a positive light
2. Innovation: finding new ways of teaching that can encompass students from all backgrounds, skill levels, and preparation
3. Competencies: providing students with a foundation of knowledge and skills for lifelong learning which continues to develop well into their professional careers
4. Programming: improving computing instruction by using developmentally appropriate instruction that promotes better learning and transfer
5. Formalism: relating the relevant mathematic theories and formalisms that underlie computing

6. E-Learning: developing computer-based environments that are comparable to traditional instructional environments which can encompass a wider audience while affording students more instructional support
7. Pre-University Issues: accurately informing pre-university students of what the computing field really entails and positively promoting the computing field to these students

Making the computing field interesting and attractive to students was a theme that seemed to transcend all the grand challenges. To do so required motivating students through an innovative and engaging curriculum, showing the full expanse of the computing field, and allowing students to learn at their own pace and level and follow their own interests. Guzdial and Soloway (2002) suggested that CS needs to adapt to the current generation of students, which includes being familiar with popular youth culture and their interests. Describing the current population of students as the "Nintendo generation," they assert that computer science educators are using an outdated view of computing and students replicating the same teaching environments they experienced from their time as students, generally by rote and decontextualized and unauthentic activities (Stein, 1998). Hadjerrouit (1999) referred to this antiquated method of teaching as "objectivist," where learning is passive and fails to engage the learner. This problem may lead to the perpetuation of outdated instruction for future generations of students. One such adaptation to the current generation, which employs a more engaging instructional environment, is to immerse learners in multimedia-rich environments in CS instruction, which are found in television, video games, and other popular technology.

## Rising to the Challenges

Several CS education researchers have offered a variety of solutions that could help students with comprehension and achievement and help instructors with instruction and classroom management. The suggestions and solutions have included the following: 1) the use of interactive, engaging, relevant and real-world projects to motivate students (Chang, Chiao, Chen, & Hsiao, 2000; Guzdial & Soloway, 2002; Hood & Hood, 2005; Lawhead et al., 2003; Moskal, Lurie, & Cooper, 2004); 2) instructional design changes to CS1 to accommodate a wide variety of student skill sets and interests (Forte & Guzdial, 2005; Lemos, 1979; McKinney & Denton, 2004; Stein, 1998); 3) access of on-demand instructional resources and course materials (Boyle, Bradley, Chalk, Jones, & Pickard, 2003; Doube, 2004; Herrmann et al., 2003); 4) using multimedia-based learning tools to provide developmentally appropriate instruction (Doube, 2004; Herrmann et al., 2003; Jehng et al., 1999; Moskal et al., 2004), and 5) code visualization tools (Cooper, Dann, Pausch, 2000; Cross II, Hendrix, Jain, and Barowski, 2007; Guzdial & Soloway, 2002; Kölling & Rosenberg, 1996).

Due to my personal experiences in computer science, I am interested in providing students the necessary support for learning and understanding OOP. The three challenges that fit best with my interests are these: providing suitable e-learning opportunities, innovative instruction, and developmentally appropriate instruction in computer science. According to Jones (2003a), a goal of instruction is to "[p]rovide learning environments that approach the effectiveness of one teacher for every learner. Such systems, properly used, can produce a significantly better-educated populace by combining advances in learning science with advances in information technology" (as cited in McGettrick et al., 2004, p. 17). McGettrick et al. (2004) advocated the use of information and computer technology in the development of e-learning environments that are viable and credible alternatives to traditional face-to-face instruction. Within the objects-first debate, Bruce (2005) sided with the objects-first approach, but advocated the use of learning tools. Such tools can provide every student with on-demand instructional assistance when needed. That was the intent of Boyle et al. (2003) when they chose to put their instructional tools online for students. Similarly, Doube (2004) created CD-ROMs for the same purpose, though on-demand access was limited to those who actually had the CDs. E-learning opportunities are also beneficial to non-traditional students, who may work full time and need the flexibility to take courses on their own schedules.

In addition, such computer-based instruction can be designed to be adaptive to students' instructional needs, and one challenge is to ensure that students can be taught in a manner that is conducive to their own levels of cognitive development. As McGettrick et al. (2004) stated, "A more specific, technically focused challenge is to build a 'one-to-one programmer's assistant': a software aid that would assist the development of individual programming skills, adapting both to different individuals and to the evolving skills of a single individual" (p. 13). As part of the programming challenge, instruction should be as individualized as possible, especially for those students who may need more help or may be deemed "at-risk." This may be difficult to manage in a traditional class where the student-to-instructor ratio is rather large (e.g., 200+ to 1), but instructional tools, such as the software aids McGettrick et al. proposed, can provide such individualized instruction.

In terms of adapting instruction to students' level of development, courses can target affective goals in addition to cognitive/performance goals as in McKinney and Denton's (2004) study. Programming may be a big hurdle for novice computer science students, though "programmer" is not the only job in the computing field. McGettrick et al. (2004) promoted the idea of instruction being sensitive to students' levels of development, personal interests, and career goals. CS1 courses can address these different needs, as in the CS1 course that Forte and Guzdial (2005) studied, where a traditional CS1 course was taught as 3 separate courses: a traditional course for CS majors, a course for engineers, and a third course for non-majors called Media Computation. McGettrick et al. believed that meeting the programming grand challenge would result in decreased student dropout rates due to programming ability. Courses tend to focus on cognitive ability alone, but affective concerns must also be addressed, as McKinney and Denton (2004) noted. It is important to ensure that students are not dropping out from boredom or lack of interest.

Using innovative teaching methods is one of the challenges in computing education especially if affective objectives, such as those in Krathwohl et al.'s (1965) taxonomy, are considered. McGettrick et al. (2004) observed, "We can find new and better ways of teaching (e.g. finding approaches that have greater visual appeal and/or that fire the imagination of current students) and of engaging *all* students in active learning" (p. 9). This is evident in the multimedia format of instruction (Boyle et al., 2003; Doube, 2004), the visual accompaniments to coding (Cross II et al., 2007; Jehng et al., 1999; Kölling & Rosenberg, 1996), interest-specific CS1 (Forte & Guzdial, 2005; Guzdial & Soloway, 2002), and game design/programming to teach computer science (Chen & Cheng, 2007; Frost, 2008; Jones, 2003b)

## A PROPOSED SOLUTION

In the literature on the challenges of e-learning, programming, and innovation in computing instruction, there seemed to be an emphasis on the creation and use of tools to assist in instruction. In rising to the challenges of computer science education, I propose to design an instructional tool that serves to engage and motivate students through imaginative forms of instruction, provides students with individualized and scaffolded instruction, and takes advantage of the e-learning features to accomplish those tasks.

To spark interest in students, the instructional tool should immerse instruction in multimedia-rich environments. Not only does the use of multi-sensory information serve to engage students in all sensory channels, it can provide students with multiple forms of representation, which may prove helpful due to the complex nature of OOP, and of CS in general. Multimedia should also serve as a means of transforming high-level, abstract concepts into viable, workable concepts for novice students.

The tools must be adaptive since the field attracts a wide range of students in terms of ability and skill set. The tools should not only scaffold instruction to the students' levels of development, it should also help students in surpassing those levels. Tools that affect cognition through amplification and reorganization include cognitive tools (Pea, 1985), and the proposed instructional tool is a multimedia-based cognitive tool (MCT) that should support and enhance student learning while engaging students in higher-order thinking and problem solving. Jonassen (1991) believed that efforts should be focused on making cognitive tools more like "thinking technologies" (à la Mindtools) instead of just looking at how multimedia can enhance instruction. In this view, the only role multimedia should have is to deliver the information. It is also important, however, to look at effective ways of using multimedia to deliver instruction and to support student cognition and development. Like cognitive tools, the role of multimedia is to assist students in reorganizing and amplifying their cognitive abilities.

Multimedia has many advantages over traditional verbal instruction by providing students with alternative forms of representation to stimulate multiple visual and sensory channels. Another affordance of a multimedia-rich environment is the motivational features of graphics, animations and sound effects, which are often found in video games and websites. Cognitive technologies can assist students in their thinking in addition to helping them transcend their own cognitive limitations. They help students with organizing, reorganizing, and refining their own memory structures (Jonassen, 1991, 2000; Pea, 1985). Therefore, cognitive tools can be an intellectual partner as well as an effective means of delivering information. Derry and Lajoie (1993) noted that "student models are now being developed to consider the cognition, metacognitive and motivational states of the learner and to encourage reflection on higher-order thinking strategies" (p. 8).

Multimedia-based cognitive tools are a derivation of cognitive tools that specifically use multimedia (information communicated through sensory multiple channels and representations) to facilitate student comprehension. Multimedia also aid students in reorganizing their conceptual understanding and scaffold students' problem solving and higher-order thinking processes.

## CSNüb and the MCT Framework

CSNüb is a tool I developed based on an MCT framework of design principles, a framework that draws from constructivist, multimedia, and motivational learning theories, and cognitive tools design. The guiding principles of MCTs are as follows:

1. Multimedia cognitive tools should adopt a sensory modalities mode of delivery
2. Multimedia cognitive tools should engage students in higher-order thinking and problem solving.
3. Multimedia cognitive tools should evoke metacognition.

4. Multimedia cognitive tools should promote student autonomy.
5. Multimedia cognitive tools should provide intrinsically motivating experiences.

Chapter 3 will provide more detail on CSNüb and its activities. The objective was to have novice students work within an object-oriented framework by using CSNüb to explore, learn, and apply their understanding of OOP. Further, CSNüb was designed not to teach OOP directly: rather, students worked with it to reorganize their conceptual understanding of OOP. In this case, learning is a byproduct of using the tool. CSNüb is a template for an interactive multimedia-authoring tool written in Adobe Flash, and it gives students the foundational code and graphics to construct an underwater role-playing game (RPG). The role-playing game was named Operation SPLASH. Figure 1.1 illustrates the relationships between CSNüb, Adobe Flash, and Operation SPLASH.

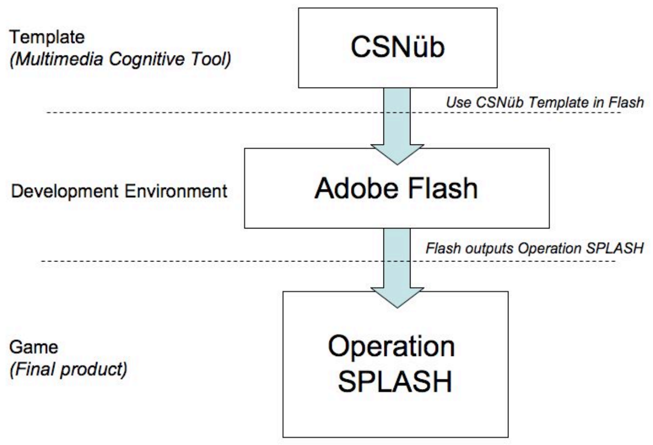

Template
(Multimedia Cognitive Tool)
CSNüb
Use CSNüb Template in Flash
Development Environment
Adobe Flash
Flash outputs Operation SPLASH
Game
(Final product)
Operation SPLASH

Figure 1.1: Diagram of the relationships between CSNüb, Adobe Flash, and Operation SPLASH

The template is akin to a game engine with which game enthusiasts build their own custom games using pre-made functionality and graphics from existing video games. CSNüb uses the multimedia game elements such as characters and objects as metaphors for the classes that students implement. They can see visual realizations of their code and how they act and interact with other objects. I use the term "game template" as CSNüb has a very specific purpose and game design and is not nearly as complex as other game engines or builders.

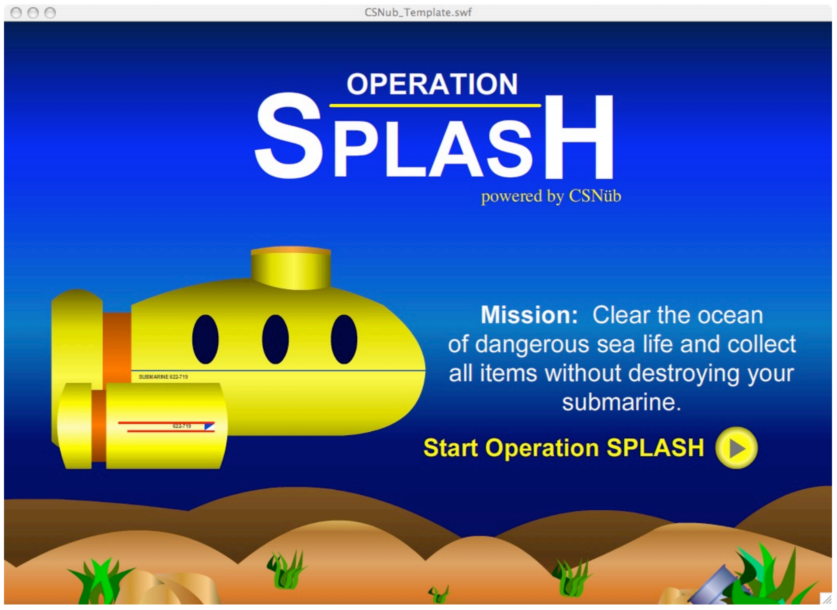

CSNub_Template.swf
OPERATION
SPLASH
powered by CSNüb
Mission: Clear the ocean of dangerous sea life and collect all items without destroying your submarine.
Start Operation SPLASH

Illustration 1.1: Splash page of Operation SPLASH

CSNüb's target audience is novice undergraduate computer science students who may not fully understand OOP, and are most likely new to the computing field. Its goal was to improve novice students' conceptual understanding of OOP and to guide students in higher-order thinking and problem-solving skills within OOP.

## RESEARCH QUESTION

Simply creating a new instruction tool and handing it off to teachers is not enough. First, there needs to be a theoretical assumption as to how the tool's features and/or corresponding activities support the learning and knowledge construction process.

This affords users some predictive outcomes on learning. Once the theoretical framework has been designed and implemented, it becomes necessary to evaluate the actual outcomes on learning. In this study, I am interested in examining how students' conceptual understandings are constructed and transformed by an MCT implementation like CSNüb.

In evaluating CSNüb's effects on cognition, the research question asked in this dissertation is as follows:

*How does using CSNüb affect the conceptual understanding of object-oriented programming for students who are novices to OOP?*

CSNüb was a preliminary attempt at implementing the MCT framework, which was theoretically synthesized from constructivist, multimedia, motivation, and cognitive tool theories. The research question seeks to discover outcomes when using CSNüb on the cognitive and learning processes with respect to OOP. In addition to analyzing the actual outcomes, the theoretical design underlying CSNüb must also be evaluated to see how well it works in practice. Another part of the research question, therefore, looks at what aspects of the MCT design framework work (or do not work) in facilitating conceptual understanding of OOP.

## Significance of the Study

Ultimately, this study seeks to inform the CS education community, through an instructional technology perspective, about the design and use of multimedia cognitive tools in teaching OOP. Student attrition in introductory computer science courses is cause for alarm. Surveys of various CS1 courses illustrate that student attrition is problematic with drop, withdrawal, and failure (DWF) rates ranging from 20% to as high as 50% (Bennedsen & Caspersen, 2007; Doube, 2004; Forte & Guzdial, 2005; Herrmann et al., 2003; McKinney & Denton, 2004). In hopes of improving student performance, this study seeks to inform those concerned with designing instructional tools about how to facilitate comprehension and guide students' cognitive processes through using multimedia. If interactive multimedia learning tools can be effective in supporting knowledge construction of abstract concepts such as OOP, this may help alleviate many of the instructional problems affecting learning and interest in introductory computer science courses.

This study addresses the innovation, programming, and e-learning challenges of computing education discussed in McGettrick et al. (2004). The tool I designed is intended to provide innovative instruction, following the lead of microworld-type visualization tools, such as Alice (Bruce, 2005; Cooper et al., 2000). These tools help to engage novices in "object-oriented" programming with motivating, game-like environments (Conway et al., 2000; Cooper et al., 2000; Henriksen & Kölling, 2004). With respect to the grand challenges in computing, the goals of an MCT are to serve as an innovative instructional tool, to give developmentally appropriate instruction to students as well as taking them beyond their current level of development, and to provide an e-learning environment that is accessible and inclusive to a wide range of students.

Since OOP is an abstract concept and way of thinking, the tool must also facilitate and mediate novice students' understanding of OOP. Other visualization tools like jGrasp, BlueJ, and Lego Mindstorms show graphical (or physical) representations of objects as a way to add concreteness to abstractions in programming (Cross II et al., 2007; Kölling & Rosenberg, 1996; Lawhead et al., 2003).

## Overview of Dissertation

Chapter 1 has highlighted some of the key instructional issues and challenges in computer science education, indicating the need for learning tools that support conceptual understanding, especially with OOP. Following the trends toward using visualizations and innovative instruction, CSNüb was proposed as a way to address these problems. Since CSNüb is a prototype implementation of the theoretically based MCT framework, which is a prototype in itself, it was necessary to assess how CSNüb would work with its target audience of novice CS students, and whether it could meet its learning objectives.

Chapter 2 details the theoretical foundation that supports the MCT framework upon which CSNüb was built. This discussion is informed by constructivist, motivation, and multimedia learning theories, as well as design principles of computer-based learning tools. Past research on tools relevant to MCTs is reviewed in this discussion.

Chapter 3 outlines the methodology employed to carry out the study. This chapter includes discussion of the methods, clinical interviews, surveys, and process-tracing methods, used to answer the research question posited in Chapter 1, and the rationale for using those methods to address the research question. This chapter also includes an overview of the participants in this study and of CSNüb and its activities.

Chapter 4 guides the reader through the analysis of the data starting from the microanalysis. This analysis leads to the emergence of the cognitive processes and factors that CSNüb facilitated to assist students in conceptual understanding of OOP.

Based on the findings from Chapter 4, Chapter 5 attempts to answer this study's research question. These conclusions are then used to construct a model of how employing MCTs may facilitate students' conceptual understanding, and to revise the MCT design framework constructed in Chapter 2. Finally, the implications of CSNüb and the MCT framework in computer science education and in instructional tool design are discussed.

# Chapter 2: Towards a Theoretical Design of Multimedia-based Cognitive Tools

## THEORETICAL FRAMEWORK

CSNüb's design was based on a design framework that draws from constructivist, multimedia, and motivation learning theories and cognitive tools design. A theoretical understanding of how people know and learn is an important first step in designing any instructional intervention (Collins et al., 2004; Gorard et al., 2004). The goal of reviewing theory and past research is to a theoretical foundation for how interventions support learning and conceptual understanding — in the case of this study, how CSNüb could support learning and conceptual understanding of object-oriented programming. The theoretical foundation is composed of four assumptions:

1) knowledge and memory are cognitive structures that are constructed and regulated by an individual;
2) the use of multimedia in instruction can facilitate and enhance learning;
3) multimedia and computer-based technologies have motivational advantages over traditional oral and paper-based instruction; and
4) computer-based tools can facilitate learning.

The following review of literature intends to establish a design framework for tools that use multimedia to enhance instruction through guiding and extending student cognition: multimedia cognitive tools (MCTs). Each assumption was informed by specific instructional theories will be discussed in detail later in this chapter. These assumptions were the building blocks for the theoretical framework of this study's research questions.

The first assumption — that knowledge and memory are cognitive structures constructed and regulated by an individual — is built upon constructivist learning theory.

The spectrum of constructivism ranges from cognitivist constructivism to social constructivism. The cognitivist end of the spectrum emphasizes knowledge structures and cognitive processes. Cognitivist constructivism, thus, focuses on the individual's cognitive development through organization and fine-tuning of such internal memory structures (Derry, 1996; Greeno, Collins, & Resnick, 1996; von Glasersfeld, 1984). On the social constructivist end of the spectrum, emphasizes how knowledge is distributed among people and between objects and learning is explained as a social process that leads to an authentic and socially interdependent goal (Greeno et al., 1996; Lave & Wenger, 1991; Vygotsky, 1978). The intended use of an MCT is aimed toward individual cognitive development, and thus, the design leans more toward cognitivist constructivism. The use of MCTs by an individual, however, involves interaction between the learner and the computer, which can be regarded as a social and dialogic process necessitating the consideration of some social constructivist tenets.

The second assumption — that the use of multimedia in instruction facilitates and enhances learning — is informed by Mayer's Cognitive Theory of Multimedia Learning (CTML) (Mayer, 2001; Moreno & Mayer, 2000). The underlying assumptions of CTML are important in determining the design of MCT for effective use of multimedia in instruction. This assumption also postulates that the use of multimedia is a more effective means of instruction than verbal methods alone.

The third assumption — that multimedia and computer-based technologies have motivational advantages over traditional oral and paper-based instruction — indicates that instructional technologies can be designed so that learners are engaged and motivated to learn the content. The student's affective state is important, as it can hinder student performance and achievement (Lepper & Malone, 1987; McKinney & Denton, 2004). Instructional designers should not assume that students automatically become engaged once presented with a piece of software. Indeed, educational software, such as educational games, can be used successfully to make learning more fun and engaging for students. Capitalizing on student interest and motivation is a goal of MCT instruction. Given the salient role of motivation in learning, Malone and Lepper's framework for creating an intrinsically motivating environment is discussed as the last component of this theoretical framework (Lepper & Malone, 1987; Malone, 1981; Malone & Lepper, 1987).

The fourth assumption — that computer-based tools facilitate learning and cognitive development — relies on the theory behind the design of cognitive tools and their impact on students' cognitive processes and development. Such tools engage students in metacognition and facilitate higher-order thinking and are also dependent on constructivist ideals (Derry & Lajoie, 1993; Pea, 1985).

Due to its prevalence among all assumptions of this theoretical framework, constructivism begins the review of literature. Though the content domain of this study is computer science, the concepts are discussed in this chapter are not limited to CS, and thus this theoretical model can be applied to other content areas as well.

## CONSTRUCTIVISM

Constructivism serves as the foundation for all other the theories discussed in this chapter, since it explains the assumptions about how students know and learn. Any use of multimedia and cognitive tools must adopt the same assumptions in order to affect significantly how people know and learn. Although cognitivist constructivism remains the focus, social constructivist implications for using cognitive tools are also discussed. A crucial component of any learning theory begins with its epistemological stance.

## Knowledge

The constructivist perspective views knowledge as composed of dynamic structures of information that are created through a person's experience, which shapes an individual's world and reality (Brooks & Brooks, 1993; Derry, 1996; Greeno et al., 1996; Piaget, 1952; Winn, 2003). From a biological perspective, Piaget (1952) described the construction of such memory structures as being similar to the process by which an organism adapts to its environment. These structures represent an individual's own internalized mental accounts through which the individual understands a concept, and they affect how he or she learns in new situations and experiences.

Derry (1996) described two extreme views of how knowledge is constructed: "the mind is shaped by nature" and "knowledge is perspectival." The former sees learning as a passive activity while the latter sees learning as an individualized active process. This is the process in which learners construct their own memory structures (knowledge). Constructivism dictates that the learner actively constructs knowledge.

A constructivist epistemological stance differs from an objectivist perspective in that the truth is not thought of as something to be discovered or learned (Crotty, 2003; von Glasersfeld, 1984). Rather, the truth is in the eye of the beholder — a person's reality is interpreted by his or her experiential knowledge. This is in line with Derry's (1996) notion that knowledge is perspectival to the individual. Von Glasersfeld (1984) set forth two principles about this departure from an objectivist epistemology. The first is the acceptance of alternate conceptions; this view suggests that there is no one truth waiting to be discovered by all. In accordance with one of Derry's extremes, just as "knowledge is perspectival," knowledge is personalized. True knowledge cannot be viewed or assumed through any other individual's interpretations (von Glasersfeld, 2005); every individual has his or her own take on what that truth may be, but not all alternate conceptions represent the same truth.

Von Glasersfeld's (1984) second principle about knowledge was that it results from an individual's observations and experiences. Both these principles center on the individual. Objectivist epistemology refers to an absolute truth that claims to be the same for everyone; in constructivist epistemology, however, different people may experience the same situation distinctly. Each person may have a different interpretation of the experience due to previous experiences and even personal preferences and values may alter the experience such that it is interpreted differently from everyone else's experiences of the same situation. These multiple and various reactions lead to many alternate conceptions — none of which should be discounted as non-truths.

### *Schema Theory*

Representation and storage of knowledge is dependent on the individual. Drawing from cognitivist theory, Winn (2003) distinguished between two forms of mental representations: schema and mental models. A schema is an abstract, organized structure that changes with inputs, while a mental model is a representation within the context in which it is presented. Together, schemas and mental models define how an individual comes to know and understand, even though the same structures may not work the same for everyone (von Glasersfeld, 2005). Since the term schema includes mental models, being a higher level of abstraction, schema is the term that will be used for the remainder of this discussion.

Derry (1996) gave three theoretical views of schemas within a (cognitivist) constructivist epistemological stance. The Modern Information Processing Theory focused on problem solving. This theory has two components: procedural knowledge and declarative knowledge. Procedural knowledge is seen as a series of if-then statements, while declarative knowledge is seen as a set of theories and generalizations. Students must develop both procedural and declarative knowledge in order to attain certain skill and performance levels. The Modern Information Processing theory is considered to be only weakly constructivist, since it tends to view the mind as shaped by nature or forces external to the individual.

In contrast, Strong Constructivism is based on the belief that knowledge comes from the individual, and focuses on how old knowledge is used in understanding new knowledge through accommodation and assimilation (Derry, 1996; von Glasersfeld, 1984). A learner's prior experience plays a role in influencing how he or she approaches a new situation (von Glasersfeld, 1987), and supporting the accommodation and assimilation processes helps to build a deeper understanding. This understanding of schema is the most perspectival, as it accepts all individually constructed perspectives.

Finally, Cognitive Schema Theory (CST) focuses on memory structures. Derry (1996) discussed this view as a bridge between information processing (e.g. the Modern Information Processing theory) and radical constructivist perspectives (e.g., Strong Constructivism). Having a thorough understanding of the types of schema that exist gives insight into how construction and maintenance can be facilitated (Derry, 1996).

CST sets forth three categories: memory objects, mental models, and cognitive fields (Derry, 1996). Memory objects are the most basic blocks, similar to mini presentations in the mind — they are minute pieces of information, including simple information such as declarative and procedural knowledge found in information processing theories. Eventually, these separate and simple memory objects merge together and form groups of similar traits; the resulting memory groups are more complex and aid in solving higher-order problems.

A mental model is an understanding of a situation created through testing, adjusting, and organizing. Mental models, like the mental representations discussed in Winn (2003), differ from memory objects because they are context dependent — the memory objects are connected to actual experiences. For example, a teacher may provide many models of a concept for a student to use as his or her own models. Just as related memory objects can be organized to form more complicated cognitive structures, so mental models can also merge to deepen the learning process.

Cognitive fields are a type of schema activated when encountering any situation. Derry (1996) noted that "experience triggers activation of the cognitive field, which in turn delineates the memory objects that are readily available for modeling the experience" (p. 168). In this view, the memory objects that are most pertinent to the situation are activated. Such memory objects are also automatically selected for updating and revision within the experience.

Von Glasersfeld (1984) claimed that success should be measured by how efficiently an individual is able to organize his or her schema. CST implies various levels (or stages) of development with respect to individual schemas. After all, cognitive fields must be developed through experience and fine-tuning of mental models. Mental models are created when students experience situations that link individual memory objects together. This may also explain the difficulty some students experience in trying to understand high-level concepts. Novice students may only have a superficial schema with loosely connected and basic memory objects or mental models.

Each of Derry's (1996) schema types had stages that provided for an individual's level of development, in which early stages are simpler and later stages become more complex and sophisticated. The complexity of an individual's schema is representative of his or her level of development as well as expertise. This brings to light how novices use their knowledge and prior experiences differently from experts.

***Experts versus Novices***

Experts tend to look at the generalizations about a problem and understand the interrelationships between meanings and their relevance to that situation (Woody, 2001). Experts approach problems through a qualitative lens and do not just look at the hard facts, whereas novices tend to see only superficial observations (Tennant & Pogson, 1985). Within the Modern Information Processing theory of schema, procedural knowledge is akin to a series of facts with almost no context (Derry, 1996). Novices are expected to have such knowledge, whereas experts have more declarative knowledge that provides relationships, conditions, and contexts which could be absent from procedural knowledge (Derry, 1996; Woody, 2001).

For example, a novice asked to write an algorithm would most likely write some code, hope it works, and attempt to execute it. This code-test-and-change process continues until it produces no errors and the algorithm produces the correct results. An expert, on the other hand, would examine the underlying algorithm and look for strategies by using his or her previous knowledge and experience on algorithms and programming. When confronted with a problem, experts are better at determining what previously-stored knowledge is most applicable to the situation (Woody, 2001), which is similar to the use of cognitive fields in CST (Derry, 1996).

The differences between experts and novices may also explain the problems instructors may have when teaching students who are new to the field. When an instructor, who is an expert in the content, teaches these novices, several issues may inhibit learning. The first may be that experts tend to forget how novices learn and understand, and may not see where and why they have difficulty. Problems may also arise if the instructor is teaching at an expert pace while the novice student is struggling at a beginner's pace. The differences between thinking processes widen, or at least maintain, the gap between students' learning potential and the instructor's level of communication. Therefore, not only should novices be taught at their own level, but students should also be trained in the expert process of knowledge acquisition and organization in order to facilitate conceptual understanding (Tennant & Pogson, 1985; Woody, 2001).

As students gain experience by engaging in learning activities, their level of development and understanding is expected to progress from novice to expert. Having surveyed the differences between novices and experts, it is pertinent to look at how individuals transition between these cognitive levels of development.

## Learning

A tenet of constructivist learning is that learning is an active process. Learning is a process in which the learner is striving towards the ideal mental state of equilibrium. As learners recover from moments of disequilibrium they will have improved upon their mental structures through adaptation and reorganization, and will have increased their expertise.

Ginsburg and Opper (1987) defined equilibrium as "a state of balance or harmony between at least two elements which have previously been in a state of disequilibrium" (p. 222). Piaget (1952) implied that equilibrium is not a beginning or natural state; thus, an individual always encounters moments of disequilibrium. Piaget noted that the state of equilibrium is achieved both through the assimilation and through the accommodation process. Equilibrium, therefore, comes only after the individual has reflected and reorganized his or her schema to resolve a conflict.

When presented with a new, unfamiliar situation or a problem that conflicts with what previous knowledge, the learner is placed in a state of disequilibrium. Disequilibrium, thus, occurs when a state of cognitive balance is disrupted due to a conflict. Under the CST, a state of disequilibrium can also be triggered when a student cannot activate a cognitive field of mental objects to solve a problem due to a lack of experience. Another cause of disequilibrium may be a result of ambiguous content or tasks that are beyond the ability or potential of the student (Derry, 1996). As learners attempt to resolve their state of disequilibrium, they are trying to adapt their schemas to the conflict.

### *Adaptation*

As mentioned earlier, Piaget (1952) likened learning to an adaptive process in which an organism is attempting to adapt to its environment. The more adapted the organism is, the better it can interact with its environment. Within a learning context, the more adapted a learner is to an area of knowledge, the better that learner can interact with and apply this knowledge. Piaget listed two cognitive techniques for adaptation: assimilation and accommodation. Assimilation "conserves the cycle of organization" by incorporating new information into the learner's current schema (Piaget, 1952). The individual is attempting to explain the phenomenon in terms of his or her own knowledge (Ginsburg & Opper, 1987; Piaget, 1952). Piaget noted that full assimilation is not always possible, as prior knowledge will always make adjustments for itself. Accommodation occurs when an individual needs to modify his or her own schema to account for new information (Ginsburg & Opper, 1987; Piaget, 1952). In this process, learners must use integration and differentiation in their reflection processes to make sense of these unfamiliar experiences (von Glasersfeld, 1987). During integration, learners may be trying to accept new knowledge by deciding where it may fit into their existing schemas. In differentiation, learners look at what changes they may have to make to their schemas in order to make sense of unfamiliar experiences. As learners are attempting to adapt their schema to new knowledge, they are in engaged in metacognition.

### ***Metacognition***

Metacognition is the process in which individuals monitor and reflect on their own thinking and reorganizes their schemas (Greeno et al., 1996; Woody, 2001). Reflection is part of the sense-making process as learners are continuously exposed to stimuli and experiences. Such self-analysis of one's own thoughts and knowledge fine-tunes any conceptual understandings one may initially construct, thereby adding to the continuing process of knowledge construction and furthering higher-order thinking (Greeno et al., 1996). Experts exhibit the trait of consciously trying to reorganize their deficient schemas in order to understand something better or to allow for the integration of new knowledge (Woody, 2001). Adjustments must be made to a schema once an individual finds out that it is inadequate or incorrect (Greeno et al., 1996; Winn, 2003).

Such adjustments are made through Piaget's adaptation methods of accommodation and assimilation. Both accommodation and assimilation involve students reorganizing and reworking their internal understanding. Reflection is necessary for students to understand concepts with multiple layers of abstraction (Ginsburg & Opper, 1987). The ability to reflect also serves as an indicator of a student's cognitive level of development (Greeno et al., 1996). Many high-level concepts in math, physics, and the like require students to keep reorganizing and refining their understandings at each level of abstraction. This process relates to what it means to understand a concept: the more iterations of reflection a learner uses, the more refined an understanding a learner has, thus, removing another level of abstraction from the concept. When learners review their actions, they are able to analyze what they have done, and may notice either that they did something correctly or that they must make corrections and adjustments. Another example is how students review their steps in a math problem to see whether they made mistakes and how they determine where those mistakes in their problem-solving process are. Upon finding errors, students may make adjustments to their schemas and problem-solving strategies to avoid similar mistakes in the future. The metacognitive process, then, becomes a useful instructional tool that promotes higher-order and more expert thinking.

### *Disequilibrium and Reactions*

Each time an individual is exposed to a new situation, he or she has to react. When unable to assimilate the information from the new situation, the individual is in a state of disequilibrium. Ginsburg and Opper (1987) classified three types of reactions an individual may have with respect to new situations or problems: alpha, beta, and gamma. Aligned with the recurring theme of using prior knowledge, reactions are also dependent on the existing schema that an individual already has. These reactions can be found in specific stages of development in children: alpha reactions in the preoperational stage, beta in preoperational and concrete operational stages, and gamma reactions in the formal operational stage (Ginsburg & Opper). Piaget (1952), however, believed that these reactions can actually take place in any stage.

In an alpha reaction, a learner may ignore the conflict. This may be due to the learner being unable to perceive the conflict — either through blissful ignorance or through inaccurately accepting it as something the learner already knows. A beta reaction implies that a learner is aware of a conflict, thus creating into a state of internal disequilibrium. In this case, the learner attempts to accommodate or assimilate the new knowledge into his or her current schema. Such a reaction should induce a metacognitive process in which the learner is reorganizing the schema to make sense of the new information.

As learners are exposed to more similar situations, they are actively developing more complex and sophisticated schemas. This process serves to help create deeper understandings, form generalizations, and make predictions for the next time they encounter a new experience. This is similar to how novices become experts in terms of the level and complexity of how experts think and approach problems. It is at this level that learners develop gamma reactions to new situations. One goal of teaching can be to create environments such that learners will have many moments of disequilibrium with the hope of eliciting beta reactions that can be fostered into gamma reactions. This relates to the requirement that teachers in a constructivist environment should empower students to take charge of their own cognitive processes in a meaningful way (Brooks & Brooks, 1993). To avoid alpha reactions, teachers must know their students' prior knowledge. Teachers must be careful not to present new information that may not be viable to students, since an individual only has prior experiences on which to draw (von Glasersfeld, 1984). Presenting new information in a viable manner affords the teacher the opportunity to teach at an appropriate level that matches the developmental level of the students. Knowing about students' prior experiences can help the teacher in tying in new knowledge and concepts to what the students already know, thereby easing the integration process and the facilitation of beta and gamma reactions.

***The Role of the Learner***

Active learning implies that students are in charge of their own learning. Therefore, the onus of constructing meanings and making sense of what one learns and knows falls on the student. In a constructivist environment, the learner is not an empty vessel filled with information that is passively memorized from the teacher or a book. "The view of the learner [has] changed from that of a recipient of knowledge to that of a constructor of knowledge, an autonomous learner with metacognitive skills for controlling his or her cognitive processes during learning" (Mayer, 1992, p. 407). Brooks and Brooks (1993) have noted that in order to create a constructivist learning environment, a teacher must encourage and promote student autonomy. The student must be empowered to take initiative and self-determination in the learning process: "Educators must invite students to experience the world's richness, empower them to ask their own questions and seek their own answers, and challenge them to understand the world's complexities" (Brooks & Brooks, 1993, p. 5). It is not enough that students take charge of their own learning, however; they still require guidance from the teacher. As mentioned above, teachers need to provide students with moments of disequilibrium to foster knowledge construction and cognitive reorganization.

## Social Constructivist Implications

Much of the discussion so far has focused on the individual, as the concern is on the individual's development and how he or she mediates that development — this reflects the cognitivist end of the constructivist spectrum. An interesting outcome of using cognitive tools is that a learner's abilities and skills are extended beyond his or her actual level of development (Jonassen, 1991, 2000; Pea, 1985). Vygotsky (1978) suggested that transcending one's actual level of development is possible if a student is guided by an expert or a more knowledgeable peer. Cognitive tools can provide such mentoring through meaningful interactions with the student. The discussion now, therefore, must include social constructivist implications of knowing and learning.

Social constructivism deems knowledge to be distributed among people and artifacts and learning to involve interacting with such people and artifacts and participating in communities of practice (Greeno et al., 1996; Lave & Wenger, 1991). Learning is also situated within the community so that what an individual learns is pertinent and relevant to the context in which it is presented (Lave & Wenger, 1991). Communities of practice view the result of learning as the transformation of an individual's identity from being a partial participant within the community to becoming a full participant (Lave & Wenger, 1991; Wells, 2000). Thus, learning also transforms one's identity within the community. Usually, such transformations occur through the mentoring of newcomers by more expert individuals.

### *Learning as a Social Process*

Learning is a social and dialogic process and involves participation and interaction. Lave and Wenger (1991) provided various examples of how participation coincides with learning. In their analysis of Alcoholics Anonymous (AA) meetings, they discussed how newcomers must learn to tell their stories so that they fit within the AA model. Newcomers start their participation process by listening to other members' stories; they can then increase their participation by telling their own personal stories.

Other members pick out parts of that story that are most relevant to them or to the model of what makes an alcoholic and then expand on it by incorporating their own related experience. This discussion allows the other members to shape the newcomer's identity as well as letting the newcomer take part in the AA culture of non-drinking alcoholics. Such participation demonstrates how a dialogic process works to promote an individual's learning.

The metacognitive process of reflection can also be described as a dialogic process. During reflection, students may engage in egocentric speech in which they are taking socially-created external knowledge and trying to make it a part of their own knowledge (Wells, 2000). Vygotsky (1978) referred to egocentric speech as a transitional stage between external and internal speech. To induce such egocentric speech, Vygotsky recommended giving activities that are just outside of a student's ability. When one has no other person to turn to for help, one must begin to turn to oneself to figure out a solution, and thus begins egocentric speech.

As Wells (2000) noted, knowledge construction involves people interacting with other people, though not always synchronously. Artifacts and tools can also mediate such interaction. Teachers must engage students in dialogue either in a collaborative discussion or in egocentric speech. Collaborative dialogues allow students to participate in the wider community while trying to expose their own beliefs in addition to incorporating others' beliefs into their own. Teachers can provide opportunities for egocentric speech so that students can begin to integrate socially created ideas into their existing knowledge structures.

### *Zone of Proximal Development*

Vygotsky (1978) believed that a child's ability to learn is not limited by his level of development. Rather, with the aid of a teacher or an object, students may be able to exceed their level of development and do things they could never have done alone. With such assistance, the student will be operating functions within a zone of proximal development (ZPD). Vygotsky defined the ZPD as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers" (p. 86).

The ZPD is in contrast to a student's actual level of development, i.e., what he or she is capable of doing without assistance (Vygotsky, 1978; Wells, 1999). Instead of looking only at students' current level of development, teachers can use the ZPD to look at students' potential. Facilitating students within the ZPD may involve one-on-one instruction, use of tools, and scaffolding and prompting provided by the teacher. This exemplifies social constructivist learning: the knowledge created in the ZPD could not have existed without the interaction of the participants and the artifacts involved (Greeno et al., 1996; Wells, 1999). Wells noted that this affords students' individualized instruction, since the teacher needs to be able to tailor instruction in response to students' behaviors, goals, interests, and potential.

## Summary

Constructivism views knowledge as structures in memory, which is constantly updated and reorganized as the individual gains more experiences. This places the learner in a constant state of active learning — continually making sense of situations and adapting old knowledge to new knowledge. The complexity of such knowledge structures can define how individuals use their knowledge in encountering new situations and problems. Hence, individuals' level of cognitive development can be determined by the complexity and sophistication of their knowledge structures. Also assumed is that individuals are not always limited to their level of development. On the contrary, with the right amount of scaffolding appropriate guidance from a more expert person, an individual can work beyond his or her cognitive development level within a ZPD.

Adapting a constructivist approach to knowing and learning directs how instruction can support cognition. In order to use multimedia within a constructivist environment, such multimedia must also support the epistemological and ontological tenets of constructivism.

## MULTIMEDIA LEARNING

According to Mayer (2001) and Moore, Burton, and Myers (2004), multimedia learning incorporates the use of at least two or more of the following in instruction: text, visuals, video, and audio. In contrast, a teacher who only posts a textual outline on the classroom board is using only one medium in instruction. As Mayer has noted, the multimedia principle states that students learn better with words (spoken or printed) and pictures together than with words alone.

Mayer (2001) provided two metaphors of multimedia learning: information acquisition and knowledge construction. Information acquisition is made possible through a technology-centered design, as it focuses on teaching and presenting information through multimedia. The learner is presented with information he or she must process. Processing information involves storing it in memory or changing previous related knowledge. This is initially done in short-term memory, but eventually, the new information is stored in long-term memory for future retrieval. This form of passive learning is not much different from a traditional lecture in that students are left alone to construct their own meanings based on previously stored information (Mayer, 2003).

In the knowledge construction metaphor, multimedia serves to engage learners in active knowledge construction. Such use of multimedia learning follows a learner-centered design (Mayer, 2001). As Mayer stated, "Learning occurs when a learner actively builds meaningful cognitive representations" (p. 141). Following this metaphor, multimedia-supported learning provides students with guidance for constructing their own mental models. Thus, the knowledge construction metaphor of multimedia learning is preferred since it prevents students from becoming passive or idle learners. As this study adopts a constructivist instructional perspective, multimedia supports active learning by providing engaging situations that cause learners to constantly reorganize their current schemas, integrate new knowledge, and apply relevant schemas to new situations. In this view, multimedia is not merely used to present information to students but rather is included to help learners with constructing knowledge and adapting their current schemas to incorporate new knowledge.

For example, students can be shown a graph of fuel prices over the past 50 years. In this case, students are provided with visuals in addition to verbal information such as a descriptive passage or numbers alone. Following the information acquisition metaphor, students would be expected to memorize the graph and any other pertinent information. Following the knowledge construction metaphor, however, students would use the graph to think and make sense of the data. The visual aid should also create moments of disequilibrium and conflict for students. As they go through this sense-making process, students may discover and extrapolate the trends in fuel prices over time. They may start applying what they already know to explain the causes of these patterns. Having the visual information supports students' abilities to generalize and predict what may happen in the future. The knowledge construction metaphor suggests the use of multimedia to engage students in thinking and active learning.

**Views of Multimedia**

Mayer (2001) described three views of multimedia: the delivery media view, presentation mode, and sensory modalities. The delivery media view focuses on the technology for delivering instruction and supports the information acquisition metaphor for multimedia learning. This view is concerned with the technology itself and not the learning involved. For example, the use of a PowerPoint slide with merely some bullet points to be memorized can be thought of as a multimedia presentation, but this use of PowerPoint simply delivers information for students to remember in a manner very similar to that of an overhead slide or writing notes on a chalkboard. This use does not necessarily support the knowledge construction metaphor, as such slides are normally used for passive learning.

Presentation mode focuses on the way information is presented. In this case, pictures, narration, and text can be used to present information, providing for different visual channels (verbal and pictorial). In this case, verbal channel refers to textual information. Allowing students the chance to learn the same topic through multiple representations is desirable as it improves and stimulates learning (Lepper & Malone, 1987). For example, illustrations such as animations and pictures have proved to be more effective in teaching recursion to beginning computer science students over the purely-text based methods. In Jehng, Tung, and Chang's (1999) study on using visualization tools to support instruction on programming recursive procedures, students in the experimental groups that had access to visualization along with their code performed better on post-tests (exams on recursion) than the control group which only had static text-based notes alongside their code. Expanding the presentation mode provides different and multiple representation of information that learners can use in combination to better assist learning.

The sensory modality view of multimedia focuses on communicating information through multiple sensory channels: audio and visual (Mayer, 2001; Moore et al., 2004). This differs from the presentation mode in that the presentation mode can provide information for different visual channels (verbal and pictorial) whereas a sensory modality view provides for different sensory channels (audio and visual). The presentation mode and sensory modality view of multimedia are preferred in that they provides learner with an environment in which they are actively processing information through multiple channels and thus gaining different perspectives and representations of the same topic.

The sensory modalities view of learning fits best with cognitivist theories as they relate to Baddeley's (1992; 2001) model of working memory (Mayer, 2001). As it will be discussed later, this model of working memory has separate components for processing auditory and visual information. This model does not exclude the presentation view, in which information can be presented verbally and/or pictorially, but rather the presentation view is encompassed in the visual channel in the sensory modalities view adopted by this study.

Up to this point, the present study has looked at multimedia as a facilitator for knowledge construction and as being used to engage learners on multiple sensory channels. After reviewing the role of multimedia in learning, it is necessary to look at how multimedia can be used to maintain constructivist tenets.

## Cognitive Theory of Multimedia Learning (CTML)

Derry (1996) argued that cognitive schema theory could help to "identify specific cognitive mechanisms that underlie schema construction and revision" (p. 167). The Cognitive Theory on Multimedia Learning (CTML) provides a theoretical basis for using multimedia to support learning and cognitive processing. As previously stated, although the present study assumes a broad constructivist perspective on learning, it focuses on the cognitivist end of the spectrum. The CTML adopts three assumptions: the dual-channel assumption, the limited-capacity assumption, and the active-processing assumption (Mayer, 2001; Mayer & Moreno, 1998; Moreno & Mayer, 2000). These assumptions are important, as they lay out how the human mind works in receiving and processing information as humans see, read, and hear it. Such a framework can guide effective multimedia use in constructing and maintaining knowledge.

### *Dual-Channel Assumption*

The dual-channel assumption follows the dual coding theory, which states that there are two systems for processing information: one for looking at verbal information and one for interpreting nonverbal information such as images (Mayer, 2001; Paivio, Walsh, & Bons, 1994). Between the verbal and nonverbal information are referential interconnections which process images to words and vice versa (Paivio et al.). Within each system, there are associative connections, which activate other information related to words or images. The dual-channel assumption relates to the sensory modalities view of multimedia learning since it suggests that information needs to be provided for both auditory and visual sensory channels.

Baddeley's (1992; 2001) model of working memory is comprised of four components: the central executive, the phonological loop, the visuospatial sketch pad, and the episodic buffer. In Baddeley's model, working memory has two sensory channels, one for visual information — the visuospatial sketch pad — and the other for verbal information the phonological loop (Mayer, 2001). (The central executive and the episodic buffer will be discussed in the next section.) The visuospatial sketch pad is the component in working memory where visual information such as images and pictures are temporarily stored, processed, and manipulated. The phonological loop "stores and rehearses speech-based information," which is primarily beneficial for language acquisition (Baddeley, 1992, p. 556). The loop is comprised of two components: the phonological store and the articulatory control process. The phonological store holds about 1 to 2 seconds of acoustic information, while the articulatory control process is inner speech in which a person is repeating information or translating visual images to verbal means to store phonologically. The phonological loop is useful as a transition phase for moving short-term memory into long-term storage. In this case, the short-term memory needs to be a part of the "loop".

It is this model of working memory that has led the present study to adopt a sensory modalities view of multimedia learning, seeing that audio and visual information is processed through different sensory channels in memory.

### *Limited-Capacity Assumption*

The limited-capacity assumption assumes that each channel is limited by the amount of information it can capture. When students are presented with words or pictures in a multimedia presentation, there is a limit to how much they can retain in working memory (Mayer, 2001). Working memory is a measure of how much an individual can remember (e.g., the number of digits or number of words in a sentence) (Baddeley, 1992). Those with higher memory spans can also make inferences from the information and follow misleading text correctly; thus, working memory is correlated to reasoning skills, though reasoning skills are still dependent on previous knowledge (Baddeley, 1992).

As mentioned above, Baddeley's (1992; 2001) model for working memory has four components: the phonological loop, the visuospatial sketch pad (discussed in the previous section) the central executive, and the episodic buffer. The central executive is the component that decides what information to focus on and divert cognitive resources to (Baddeley, 1992, 2001; Mayer, 2001). Baddeley (2001) implied that the central executive does not actually have storage capabilities. In Baddeley's (2001) latest model of working memory, the episodic buffer is a subsystem of the central executive, and it interacts with long-term memory at the point where visual semantics and language interact. In contrast, the phonological loop only interacts with the language part of long-term memory, while the visuospatial sketch pad only interacts with the visual part of long-term memory. The visuospatial sketch pad and phonological loop, thus, are regarded as slave systems to the central executive (Baddeley, 1992).

In his work with Alzheimer patients who suffer from problems with short-term memory, Baddeley (1992) noted that the central executive was an evident part of working memory, since such patients were not able to perform well on tasks that involved both visual and verbal tasks. Baddeley's (1992) experiment which involved teaching Russian to Alzheimer patients showed that they were unable to remember vocabulary words (in Russian paired with their native language) when presented in an auditory or visual fashion.

Since each channel has its own process and limits, it is important not to overload the channels. Thus, it is essential for the central executive, when presented with a large amount of information, to pay attention only to the important and relevant portions. The redundancy principle is of interest to the limited-capacity assumption of the CTML since redundant information on more than one channel may be cause for overload (Mayer, 2001). Mayer used the example of a multimedia presentation in which a user hears an audio narration and also sees the narration's script on the screen. In this case, the user is hearing and seeing the same information, but on different channels (audio and visual). Such duplication of information is considered redundant and should be avoided to prevent overload.

***Active-Processing Assumption***

Under a constructivist perspective on learning and instruction, a learner is actively and constantly constructing meanings and making sense of what he or she experiences (Brooks & Brooks, 1993). Within a multimedia environment, the learner can still go through this process of sense-making. Mayer (2001) gives three processes for active learning: selecting, organizing and integrating. Through these three active processes, the learner continues to make sense and meaning from the information processed as an active learner. Both selection and organization are processed on different channels for verbal and pictorial information. Mayer regarded the selection of words, the selection of images, the organization of words, and the organization of images as separate steps. In Baddeley's (1992; 2001) model of working memory, the central executive is in charge of "selecting" out important information.

A multimedia environment may provide learners with a great deal of information. Learners can be guided towards selecting the relevant and important words and pictures of a multimedia experience through visual cues such as arrows, graphics, color, and font styling (Hartley, 2004; Mayer, 2001). Visual cues can assist in decreasing the learner's cognitive load, which results in more space for higher-order thinking (Mayer, 2001). The organization process is one whereby students form relations between the chunks of information they have selected. Mayer referred to the connection between selected words and images as "cause-and-effect" relationships. The verbal and pictorial structures are called the "verbal map" and the "pictorial model," respectively.

The last process is the integration of both the verbal map and the pictorial model. This final step of active learning requires the student to combine the structures of both perspectives and make sense out of them. In this sense-making process, learners must rely on using prior knowledge in understanding the information by integrating it into their existing knowledge structures so that it becomes viable (Ginsburg & Opper, 1987; Mayer, 2001, 2003; von Glasersfeld, 1987). Accordingly, if learners successfully integrate such new knowledge, they have adapted to it such that they remain in a state of equilibrium (Ginsburg & Opper, 1987; Piaget, 1952).

## Summary

Multimedia learning utilizes multiple forms of media to support learning. Using Mayer's (2001) CTML, a constructivist learning environment can take advantage of multimedia to support and enhance active learning and knowledge construction and to promote higher order thinking through employing a constructivist perspective that creates a learner-centered environment to promote student autonomy. In addition to its instructional affordances, multimedia can be used for motivational purposes as well.

## MOTIVATION

Salomon and Globerson (1987) argued that having the skills and knowledge does not automatically mean that the learner will be successful. They contended that a learner's knowledge set and actions are mediated by his or her sense of mindfulness, which they defined as "a mid-level construct, which reflects a voluntary state of mind, and connects among motivation, cognition, and learning" (p. 623). One factor that manages a learner's mindfulness is a learner's sense of motivation and efficacy. Jonassen (2000) believed the use of Mindtools can help guide the mindfulness of learners through engaging them in active knowledge construction. Salomon and Globerson (1987) implied that motivation is an important factor in guiding mindfulness.

Another rationale for including motivation theory in the discussion of MCTs relates to their "fun" attributes. Multimedia-rich environments, such as video games, are pervasive in today's society, especially at the college level. For this reason, Guzdial and Soloway (2002) argued, multimedia environments should be used to teach the students of the present generation which has grown up with video games, television, and other multimedia-rich technologies. Jones (2003b) reported that 65% percent of college students surveyed played video games regularly or occasionally. In terms of affect, the report also showed that college students mostly had positive feelings towards playing video games with 36% reporting it as "pleasant" and 34% as "exciting."

Although the scope of MCTs is much narrower than that of any modern video game, the two share common attributes such as immersive multimedia environments, heavy interaction (between user and game and between user and other users), and requiring thinking and problem-solving. The motivational affordances of electronic environments can be capitalized upon in order to improve student affect.

## Affect

Though cognitive goals are important for student achievement, student affect is also of major significance. The problem of student attrition discussed at the beginning of this study may be a result of negative impacts on the affective domain. Aslop and Watts (2003) noted that one severe consequence of negative affect is that it "can overwhelm thinking and concentration so that intellectual efforts are swamped" (p. 1043).

If a constructivist perspective is adopted, then affect must also be included. Knowledge is a personal construction that is based on and connected to prior experience, and such experiences are value-laden with emotion and feeling. Bloom's taxonomy is usually known as a set of categories of objectives in the cognitive domain (Krathwohl, Bloom, and Masia, 1965), but the taxonomy actually included three domains of objectives: 1) cognitive, 2) affective, and 3) psychomotor. According to Krathwohl et al., the five categories of objectives in the affective domain are receiving, responding, valuing, organization, and characterization.

"Receiving" indicates that learners are aware of the content and take an active role in choosing what they want to take notice. Interest and appreciation begins at this stage. "Responding" looks at how much effort and personal emotion the learner includes when asked to react or apply his or her knowledge. "Valuing" can result in the highest level of interest in and appreciation of a topic; this depends on how much learners have internalized such knowledge and the value they tie to it. "Organization" looks at how a learner organizes his or her values with respect to the value systems of others. Lastly, "characterization" deals with how learners use their value systems and their consistency in examining new ideas.

Affect is the emotional value or feeling an individual has towards any given topic (Aslop & Watts, 2003; Krathwohl et al., 1965). McKinney and Denton (2004) observed that “the affective domain supports the internalization of cognitive concepts and fosters the development of curriculum and industry-related interests, attitudes, values, and practices” (p. 236). McKinney and Denton found a significant correlation between affective factors — such as students’ sense of interest, competence, effort, and lack of pressure — and their course grades in an introductory computer science class (CS1). McKinney and Denton also found a significant negative shift between pre- and post-tests that measured students’ affective domain from the beginning to the end of the course, respectively.

Moskal et al. (2004) examined the use of Alice, a 3D animation software, used to teach programming in CS1 to “at-risk” students. The treatment group, which used Alice, showed a significant increase in retention over the course of two years. The treatment group had a retention rate of 88% while the two control groups had 47% and 75% retention rates. Although it was not statistically significant, the researchers found a higher increase in the pre and post affective scores with respect to confidence and liking in the treatment group.

## Self-Efficacy

Affect also includes self-efficacy: how students perceive themselves in terms of their own capabilities, self-worth, and self-confidence. Self-efficacy can be regarded as the catalyst for motivating individuals to do something. Conversely, low self-efficacy can act as a suppressor of action.

Bandura (1994) defined self-efficacy as “people’s beliefs about their capabilities to produce designated levels of performance that exercise influence over events that affect their lives and self-efficacy beliefs determine how people feel, think, motivate themselves and behave" (p. 1). Bandura provided four sources that can affect self-efficacy. Mastery experiences can offer individuals moments of success thus raising their levels of self-efficacy. Failure during these experiences can result in negative impacts. Vicarious experiences affect an individual's self-perceived capabilities when he or she sees others that are similar in abilities succeed or fail. Social persuasion experiences occur when others try to instill faith in another's capabilities, as in coaching or boosting someone's ego. Bandura noted that it can be harder to boost a learner's level of self-efficacy than to lower it. Similarly, self-efficacy can be kept intact by reducing negative or stress-producing experiences.

## Interest

For students to have positive affect towards computer science topics, they must take some interest. Furthermore, interests are tied to very specific ideas or concepts (Schiefele, 1991). In the taxonomy of Krathwohl et al. (1965), learners must begin to receive and accept learning those topics since it is at the receiving stage that individuals' levels of interest begin. On the other hand, interest may also serve as a filter blocking out what a learner does not want to receive (Renninger, 2000). This supports the constructivist belief that knowledge is individual, personal, and affects how the world is perceived.

Renninger (2000) differentiated between two forms of interest: situational and individual. Situational interest is triggered by a particular incident or situation and is only temporary (Renninger, 2000; Schiefele, 1991). Individual interests are enduring and life-long. Also, individual interest develops over time and must be fostered (Renninger, 2000). There are many instructional advantages to developing and using individual interests. As previously mentioned, McKinney and Denton's (2004) study found that student interest was significantly correlated to their grades. The benefits of student interest include an intrinsic desire (motivation) to be deeply engaged in their activity, which may promote student autonomy.

In tailoring CS1 courses to students' academic interests (a traditional course for CS majors, a course for engineers, and a media computation course for non-CS majors), Forte and Guzdial (2005) found a decrease in the drop, withdrawal, and failure rate in the two customized courses as compared with the traditional course. Also, when students taking the media computation course were asked if they would take another (general) CS course, only 6% said they would, while 60% of the same students said they would take an advanced media computation course. This showed that identifying and targeting student interests can increase motivation.

Fostering individual interest should be an instructional objective, but it is also necessary to expose students to new ideas and create new interests. This may have to start with sparking moments of situational interest or drawing from students' individual interests. For example, an educational video game's design can be centered on motivation theory so that having fun within an academic content may lead to taking an interest in the content (Liu, Toprac, & Yuen, 2008; Toprac, Yuen, Steele, & Reimer, 2004; Yuen, Toprac, Steele, & Reimer, 2004). Such a design can also be applied to MCTs, which can exhibit game-like features.

This idea resonates with von Glasersfeld's (1984) view of knowledge — that an individual uses his or her knowledge structures, built by previous experiences, to make sense of new ideas so that they become viable. A learner cannot instantly take an interest in an entirely foreign idea unless there is something to trigger situational interest or something that is related to the learner's individual interest. When encountering a new situation, individuals may have different reactions, and the type of reaction (alpha, beta, or gamma) is dependent on the individual's level of development, amount of exposure to similar situations, and knowledge structures (Ginsburg & Opper, 1987; Piaget, 1952). Likewise, an individual's set of interests may affect how he or she receives and responds to a new situation (Deci & Ryan, 1993). For example, if a new idea is of no interest to an individual, he or she may choose not to receive the information or may choose not to respond.

## Motivation

As Deci and Ryan (1993) have noted, "Evidence indicates that intrinsically motivated activity tends to be associated with greater conceptual learning, more creativity, increased cognitive flexibility, a more positive emotional tone, and higher self-esteem than does externally controlled activity" (p. 32). Greater conceptual understanding is desirable in the context of the present study, since the target content area is computer science, which is known to contain many abstract and high-level concepts. A student is intrinsically motivated when he or she does an activity just for the sake of doing it (Deci & Ryan, 1993; Malone, 1981). In contrast, extrinsic motivation is based on external rewards such as prizes or food as a catalyst for doing the activity. Deci and Ryan (1993) found that extrinsic motivators can decrease intrinsic motivation and can also hinder performance if those rewards are expected. Test and exam scores are examples of extrinsic motivators. Deci and Ryan surveyed a study in which participants learned material either for the purpose of teaching it or for being tested on it and found that those participants who learned in order to teach the material exhibited higher conceptual understanding than those who learned it for a test.

Intrinsic motivation can lead students to "spend more time and effort learning, feel better about what they learn, and use it in the future" (Malone, 1981, p. 335). This relates to the advantages of individual interests and the concept of flow (Csikszentmihalyi, 1990; Renninger, 2000). Like playing on students' individual interests, thus, intrinsic motivation becomes a desirable and useful instructional tool. Using intrinsic motivation may help students develop their own interests and values. If constructivist learning requires student autonomy, then increasing students' intrinsic motivation should help them in this area. According to Malone and Lepper (1987), an intrinsically motivating environment must provide individuals with elements of control, challenge, curiosity, and fantasy.

### *Control*

Control is the amount of power a user has within a situation to affect and cause outcomes as well as the freedom to choose his or her actions (Malone & Lepper, 1987). From a constructivist perspective, teachers need to promote student autonomy and allow students to form their own ideas and values (Brooks & Brooks, 1993). Similarly, a goal of intrinsic motivation is for students to take initiative and produce this sense of self-determination (Deci & Ryan, 1993). In both cases, students must have control of their own cognitive processes and the situation in which they place themselves, or at least have the perception of control. For example, Pea (1985) believed that students need to learn how to search for solutions on their own. In his review of the cognitive tool Algebraland, Pea found value in allowing students to solve algebraic equations using any operation they chose. This showed control on both the student's cognitive level, in terms of arriving at his or her own solution, and on the student's power level, in terms of using his or her own solution.

Learners must have the freedom to explore the environment, choose their own path within the environment, and select any tools or information they deem necessary. Choice is a result of a user's individual interest and guides the selection of actions taken (Deci & Ryan, 1993). Allowing users to select their own paths from a number of options supports their belief that they are in control of the situation. Malone and Lepper (1987) suggested that an optimal amount of choices is needed, since too few options limit users' control, while too many options devalue any feeling of control. Either of these extreme cases can lead to a decrease in the sense of control, which may lead to lower intrinsic motivation. Malone and Lepper referred to contingency as a reflection of control: the user can see the effects and consequences of his or her actions. This is directly related to feedback prevalent in the discussion of challenge.

Related to the issue of control of the environment is user pacing. In electronic learning environments, there are two types of control: user-controlled and system-controlled. User-controlled environments allow users to learn at their own pace, moving on to the next topic when they are ready. In a system-controlled environment, the computer program determines when the learner is ready to continue. Aly, Elen, and Willems (2005) studied the effects of user-controlled and system-controlled courseware in a dentistry class. Students were tested on their knowledge and understanding of orthodontic appliances before and after using the courseware. The study found no significant difference in post-test scores between students who used a user-controlled system and those who used the system-controlled courseware. In this case, the results were not dependent on the type of pacing. Liu et al. (2008) found that middle school students using a multimedia-based, problem-based learning environment, Alien Rescue, reported negative affect when their sense of control was taken away by a video expert tutor. This was mainly a learner-controlled environment in which students were able to navigate at will, but the expert tutor tool disrupted this control by requiring students to watch the video all the way through.

On the other hand, Tabbers, Marens, and Merriënboer (2004) considered that the pacing control depends on the type of media involved. They found that the use of visual text was more learner-paced than audio text. Visual text gave users more control since they could read as much or as fast as they needed, while audio text required users to sit through the whole recording, and thus users were not allowed to move on when they felt it was necessary. Moreno and Valdez (2005) found that specific conditions of interaction led to lower performance. In their study, there were two groups of students: one group had to arrange pictures in a sequential order while another group's pictures were displayed in a static order. Moreno and Valdez found that the first group, who had to rearrange pictures in order, performed more poorly than the group who only had to view pictures that were already ordered, and they speculated that the negative effect was due to the conditions surrounding the interaction: students were required to order the pictures within a set time. A further analysis, however, revealed no significant differences between setting time limits and allowing students as much time as they needed.

Although MCTs should provide a user-controlled environment, there must be some control from the software so that appropriate guidance can be given to users. Most of the environment should give the user control as much as is instructionally effective in order to sustain intrinsic motivation and empower student autonomy.

### *Challenge*

Challenge refers to "performance goals whose attainment is both uncertain yet likely to contribute to enhanced feelings of self esteem" (Lepper & Malone, 1987, p.

275). Challenge cannot be marked by the goal itself alone; it also involves the individual trying to attain it. Deci and Ryan (1993) noted that providing individuals with goals that are too easy can result in boredom. Conversely, very difficult goals can result in frustration. Both extreme cases have negative impacts on an individual's affective domain, which in turn, decreases motivation (both intrinsic and extrinsic).

Goals and feedback are two important components of challenge. Malone (1981) stated that "a good goal is personally meaningful." In the affective domain, an individual must be willing to receive and respond to the goal. This process is regulated by the interests the individual has towards the topics, which are encompassed by the goal. If the student has no interest or negative attitudes towards the goal, he or she will not attempt to achieve it. Similarly, the objective of the activity must present a problem that is recognizable to learners so that they have the desire to respond and react (Krathwohl et al., 1965). Otherwise, a student may not even notice the disturbances, as in the case of alpha reactions (Ginsburg & Opper, 1987).

An appropriate level of challenge must be sustained to keep users engaged. The key is to find an optimal challenge in which the task is neither too difficult nor too easy. Further, the goals of each learning module must also be relevant and attainable to the users in order for them to be intrinsically motivated to attempt to accomplish them. Since users may vary in terms of developmental levels, it is necessary to accommodate a range of abilities and skills and adapt to them, just as informative feedback is essential for guiding students in regulating their knowledge structures and positive feedback is necessary to sustain positive affect.

Pellegrino, Chudowsky, and Glaser (2001) described feedback as being "essential to guide, test, challenge, or redirect the learner's thinking" (p. 233). The type of feedback is conducive both to student development and to student affect. Positive feedback, such as encouragement, helps to increase intrinsic motivation by raising students' level of self-efficacy and associating positive values with whatever topic they are learning (Deci & Ryan, 1993; Lepper & Malone, 1987; Malone & Lepper, 1987). Along with failure, negative feedback has the opposite effect and decreases intrinsic motivation and self-efficacy (Deci & Ryan, 1993), and thus the affective values towards that topic may also grow more negative.

Feedback is also essential for guiding students through knowledge construction as well as in their ZPD (Wells, 1999). Related to the issue of control, contingency is a key issue; adapting to a user's level of development, derived from user input, is an example of contingency, since the changes in activity and feedback are a direct result of what the user does and inputs. Lockee, Moore, and Burton (2004) noted that such adaptations were also found in programmed instruction in which intrinsic programming provided instruction that was adapted to the user's previous inputs. Intrinsic programming also provided multiple levels of instruction, offering more or less help to students depending on their levels of understanding.

### *Curiosity*

An intrinsically motivating environment stimulates users' curiosity, allows them to follow that curiosity, and then provides them the opportunity to resolve that curiosity (Malone, 1981). This helps to focus their attention on the topic in question. Malone and Lepper (1987) described two forms of curiosity: sensory and cognitive.

Multimedia lends itself to evoking sensory curiosity on multiple sensory levels. A multimedia environment offers a wide range of media (e.g., graphics, animations, videos, sounds, and music) that can be used to tantalize the senses and catch users' attention. The curiosity thus evoked can entice users to interact with the program (Malone & Lepper, 1987).

Cognitive curiosity is similar to providing students with moments of disequilibrium. Malone (1981) hypothesized that there are three characteristics to the state of equilibrium: completeness, consistency, and parsimony. Within an instructional scenario, students are faced with information that makes them feel that what they know is 1) incomplete — missing something needed to understand the information; 2) inconsistent — the information does not align properly with what the student already knows; or 3) unparsimonious — the information requires too much effort to process. Each of these scenarios leads students to reorganize their schemas so that they can return to a state of equilibrium.

Just as there needs to be an optimal number of choices, so also there needs to be an optimal level of curiosity. Malone (1981) believed that the mechanisms for providing curiosity should not be entirely out of the user's scope. In other words, "they should be novel and surprising, but not completely incomprehensible" (p. 362). As previously mentioned, students should be engaged in beta reactions when cognitive curiosity is evoked as opposed to alpha reactions (Ginsburg & Opper, 1987).

### *Fantasy*

One tool for intrinsically motivating students is to provide elements of fantasy where they can momentarily depart from their own realities. Malone (1981) claimed that "intrinsic fantasies are both (a) more interesting and (b) more instructional than extrinsic fantasies" (p. 361). Two advantages that Malone listed for intrinsic fantasy are that: 1) they allow individuals to apply prior knowledge to new and different situations, and 2) they can help improve memory. Fantasy can also give the illusion of a personalized experienced which may ease or encourage participation in the activity (Lepper & Malone, 1987). Such customization can be accomplished by referring to the user by name or adapting to the user's interests and responses. Liu et al. (2008) found that the fantasy component of Alien Rescue, based on its science fiction storyline and role-playing as a scientist helping aliens find a new home, served as one of the anchors to its motivational design. They found that one of the sources of intrinsic motivation was that students felt like they were voluntary actors in the role of a scientist.

Fantasy should always work to enhance the activity's instructional objectives (Lepper & Malone, 1987). Just as feedback needs to be constructive and formative to be instructionally valuable, the outcomes within the fantasy environment involving failure should not overshadow those involving success. Such consequences for incorrect actions would detract from the instructional objectives since users would be more inclined to pursue the failure routes. These consequences must also be carefully avoided when the user is doing something correct. Examples of undesired outcomes include spectacular scenes like explosions, fantastic changes to the storyline, scoring points, and so forth. All these examples can reduce the activity to an extrinsically motivating experience, and thus deter the students from reaching the instructional objectives.

Malone and Lepper (1987) distinguished between two types of fantasies, the exogenous and the endogenous. In exogenous fantasy, the fantasy is dependent on the skills involved, but the skills may have nothing to do with the fantasy (Malone, 1981; Malone & Lepper, 1987). Exogenous fantasy implies that the required skills may not be authentic to the fantasy's context. Malone (1981) used the example of the Hangman game in which the player tries to save the life of a person from being hanged by guessing the letters of the word. The fantasy of saving a person's life is dependent on the player solving the word correctly. Guessing letters, however, is unauthentic to the scenario of saving a person from being hanged.

Endogenous fantasy not only depends on the skills involved, but the skills required also further the fantasy (Malone, 1981; Malone & Lepper, 1987). Doube (2004) used computer science concepts related to real-world metaphors — for example, the concept of a linked list is related to a fishing line with various lures attached. The bidirectional relationship between endogenous fantasy and scenario tasks can be seen here: the objective of adding a new lure to the fishing line is situated within the fantasy. The fantasy also depends on the skill: if the user does not add the new lure correctly, the entire fishing line can be lost. Within the "real" context of a linked list, if operations are not done correctly, the entire linked list will be lost. Additionally, the fantasy provides yet another representation that users have to negotiate meaning.

**Summary**

Motivation is essential in a field, such as computer science, where student attrition remains an issue of concern (Beaubouef & Mason, 2005; Doube, 2004; Forte & Guzdial, 2005; Herrmann, Popyack et al., 2003). The need to engage students' interest in computing is a theme that resonated through McGettrick et al.'s (2004) report on challenges in computing instruction. Students need to be engaged so that they are receptive and attentive to what they are learning. It is when they are capable of receiving and valuing the knowledge that they become interested in learning (Krathwohl et al., 1965). Therefore, it is useful to incorporate motivation theories into instruction, especially when dealing with students who may not yet have individual interests in the content area.

For students who already have the individual interest, instruction must sustain those existing interests, one aspect of which is positively impacting students' affective domain so they still value what they are learning, and believe they can succeed in the field. Including innovative instructional activities can help this. Malone and Lepper (1987) offered their taxonomy for designing intrinsically motivating environments. Such environments afford students with a sense of control, challenge, curiosity, and fantasy. Such frameworks can be incorporated into the instructional design of any activity to motivate students. The ultimate goal is to encourage students to take a momentary, situational interest in the content so that it can develop into a long-term individual interest.

The previous sections on multimedia and motivation have mentioned computer-based environments but have not discussed specific uses of computer-based tools on learning. This will be examined in the next section on computer-based learning tools and cognitive tools.

## COMPUTER-BASED LEARNING TOOLS

The last assumption of the MCT framework deals with computer-based tools that can facilitate learning and extend cognition. Vygotsky's (1978) ZPD assumed that with the guidance of a more expert person an individual can operate in a zone of outside his or her cognitive limitations. But should the source of this guidance be limited to a person? This section discusses computer-supported learning tools as a source for guiding students within and outside their own levels of development.

## Cognitive Tools

Computer assisted instruction, such as tutorials, programmed instruction, and drill and practice-type programs, is an example of learning from computers which is good for promoting student automaticity and training (Jonassen, 2000; Lockee et al., 2004). Such instruction, however, places the student in a passive role where learning is not meaningful and is measured by behavioral and performance objectives. Jonassen believed in using computers as intellectual partners to support: 1) knowledge construction, 2) exploration, 3) learning by doing, 4) conversing, and 5) learning by reflection.

Pea (1985) defined cognitive technologies as “any medium that helps transcend the limitations of the mind, such as memory, in activities of thinking, learning, and problem-solving” (p. 168). Jonassen (2000) made the distinction between “learning from computers” and “learning with computers.” Such “learning with computers” places students in an active learning role by focusing on developing their cognitive processes as well as expanding on their cognition, which Pea (1985) described as a role of cognitive technologies. Such technologies are cognitive tools.

Derry and Lajoie (1993) classified cognitive tools into three categories: model builders, non-modelers, and a middle path between the first two. Cognitive tools that guide the development of mental models and knowledge construction are model builders (Derry & Lajoie, 1993; Lajoie, 2000). As an alternative, non-modelers focus on extending students’ cognitive processes, as they argue that a computer could not grasp a person’s mental model adequately. Derry and Lajoie (1993) pointed out that non-modelers require no artificial intelligence in their design. Examples of this type of cognitive tool can be found in Jonassen’s concept of Mindtools (Jonassen, 2000; Jonassen et al., 2003). Mindtools’ general premise is, not to reduce the amount of information that students process, but rather to increase students' higher-order and critical thinking skills. The third category assumes that model builders and non-modelers are not mutually exclusive; rather, students' modeling and cognitive extending are complementary. Since computer science involves higher-order thinking and problem-solving skills, technology supported-instruction should not be limited to visualizations or other forms of media but should also facilitate and support students' cognition so they can acquire higher-level and complex thinking.

Though the forms and definitions of cognitive tools vary, they usually share the same qualities. Cognitive tools do not always imply computer-based technology (Jonassen, 2000; Pea, 1985), but for the purposes of this discussion, cognitive tools will be regarded as computer-based programs. These programs can be specifically designed as cognitive tools or they can be regular productivity tools (e.g., databases, spreadsheets, etc.) that are used in a specific manner (Jonassen, 2000; Pea, 1985). Using these tools in such a manner helps students in reorganizing their knowledge structures and engages them in critical and higher-order thinking. Jonassen (2000) believed that cognitive tools drive students to think more deeply when using them.

Informed by Jonassen (2000) and general constructivist learning ideals, below is a summary of three features that define the term "cognitive tool":

1. Cognitive tools extend the learner's cognition by facilitating higher-order thinking (Derry & Lajoie, 1993; Jonassen, 2000; Lajoie, 1993; Pea, 1985). Vygotsky (1978) believed that students' ability level is not limited by their developmental level. Instead, with proper guidance by a more-expert person, students can go beyond their actual level of development operating within a zone of proximal development (ZPD). The actual memory structures and development level are not changed. Pea (1985) regarded cognitive tools as

amplifiers when they functioned in this way, since higher-order processing is accomplished by reducing the amount of lower-order processing, by making a problem easier, or by removing the simpler, lower-level tasks. Again, Jonassen (2000) felt that such Mindtools should make students think even harder than if they did not use the tool.

2. Cognitive tools evoke the metacognitive process. Pea (1985) provided two metaphors for the role of cognitive tools: they can serve as amplifiers, which increase the learner's abilities to think and problem-solve, or they can serve to facilitate the reorganization of the learner's knowledge structures. Such tools provide thus learners with different perspectives on the same knowledge and evoke metacognition. Though Pea emphasized cognitive tools as more as reorganizers rather than as amplifiers, both roles are considered here.
3. Cognitive tools are learner-centered in at least two ways: first, the learner initiates the learning process, which supports the constructivist tenet that learner is in control of his or her own learning (Brooks & Brooks, 1993; Jonassen, 2000). This also promotes student autonomy, in which the learner claims ownership and responsibility over his or her own learning (Brooks & Brooks, 1993). Second, the use of cognitive tools means that instruction is adapted to the needs of the student. Even in the days of programmed instruction, such computer-based tools adapted to student's input through feedback, branching, and prompting (Lockee et al., 2004). Students are also expected to be active learners by interacting with the cognitive tool; otherwise, the program merely becomes a video or flashcard. Students need to be able to interact with the cognitive tool so they can explore, manipulate, and test ideas (Pea, 1985; Harper et al., 1991; Jonassen, 2000).

## Overview of Learning Tools in CS Education

Due to the complex and abstract nature of computer science concepts, some instructional designers have found it helpful to provide students with learning tools that utilize visualizations, innovative activities, and cognitive scaffolding (Boyle et al., 2003; Brusilovsky et al., 2006; Chen & Cheng, 2007; Cross II et al., 2007; Doube, 2004; Ferguson, 2003; Henriksen & Kölling, 2004; Hood & Hood, 2005; Jehng et al., 1999; Kölling & Rosenberg, 1996). Such tools were designed to help students with understanding the flow of the program as well as its behaviors and the underlying concepts. The following section provides a brief overview of current tools used in teaching OOP and in assisting conceptual understanding. These examples illustrate the variety of tools used in practice and provide an overview of how visualizations are used.

### *Alice 2.0*

Alice (http://www.alice.org) is an interactive 3-D graphics authoring tool aimed at novice CS students who are not computing majors (Conway et al., 2000; Cooper et al., 2000; Pausch, 2008) with which users create and add objects (e.g., people, furniture, buildings, etc.) to a 3-D world. Through a graphical user interface, users attach behaviors and properties to each of these objects to create an animation, such as making an object move around or interact with other objects in the scene.

Once a scene is constructed, the user clicks the play button to see the animation. Alice also allows for the activation of behaviors based upon user input. For example, the user can add a character in the middle of the world. The user can "program" the animation so that when the Z button is pressed, the character moves to the left one meter (in the world), scales down to half its size, etc. Most of the programming consists of dragging and dropping objects' methods into event handlers; even values are assigned via drop-down boxes.

Even though Alice is a 3-D environment, much of the complex mathematics behind 3-D computer graphics, such as matrices, lighting, and XYZ coordinates, is hidden from the users (Conway et al., 2000). Though Alice is good for visualizing objects, Powers, Ecott, and Hirshfield (2007) strongly suggested that instruction must explicitly tie Alice to other high-level computing languages, since they found that the Alice analogy did not work well with students when learning about advanced OOP concepts within a high-level language like C++ and Java.

### *BlueJ*

BlueJ (http://www.bluej.org) is a popular visual development environment that facilitates students' thinking in terms of objects (Kölling & Rosenberg, 1996). As students write their classes in Java, a visual representation (a rectangle) appears on the screen. Since objects are expected to interact with each other, several dependencies occur, and such relationships between classes are displayed. One such dependency is one class inheriting from another class. Students can create a new class file and manually edit the code stating what it inherits from another class. BlueJ allows for a visual approach: instead of writing this code, students can go through the visual representation and connect the visual representations together with a unidirectional arrow. A change in one class may affect other dependent classes. As students design their programs they can see the dynamic changes in their object-oriented solution.

In their evaluation study, van Haaster and Hagan (2004) found that students believed that using the tool helped them to pass the course and to understand OOP.

They found that, according to Bloom's Taxonomy, when using BlueJ, students tended to be working within the higher levels of analysis and synthesis in terms of cognitive objectives and the higher levels of valuing, organizing, and characterizing in the affective domain.

### *Greenfoot*

Greenfoot (http://www.greenfoot.org) is a development environment that teaches lower-level students object-oriented programming (Henriksen, 2004; Henriksen & Kölling, 2004). Greenfoot provides students with a 2-D visual in which their classes are realized into visual representations in a "World." The World represents the graphic environment while the "Actor" represents any object that appears in the World including characters, timers, scoreboards, etc. Students can create their own characters and program their behavior within the environment, as well as interacting with other objects (or characters) in the environment. Though it is built on top of BlueJ, Greenfoot provides a more fun game-like microworld.

Students create subclasses that inherit from either of the two predefined classes: World and Actor. Students have to write the appropriate Java code to define their subclasses and also have to assign an image to each class (e.g., an icon) that is the visual representation of their class. Once their classes are complete, they can instantiate them by dragging objects into the World. When the students click the Play button, they are able to see their objects in action.

### *jGrasp*

jGrasp (http://www.jgrasp.org) is a development environment in which users can write programs and see a visual representation of the program (Cross II et al., 2007). jGrasp acts like a debugger in which the user watches every line of the program being executed and there is a visual depiction of the program as well. It can also display a high-level overview of the different classes in the program and their relationships between each other.

One of the affordances of jGrasp is its Viewer feature, which shows visual representations of data structures while the program is running. Users of the program can see a step-by-step process, with respect to the code, of 1) the construction of the data structure, 2) the insertion of new data, 3) the removal and management of data already in the structure, and 4) the contents of the data structure. These visualizations are not limited to static pictures; rather, it is more like viewing an animation, though the user can pause it at any time to examine the state of the program.

Cross II et al. (2007) found a significant difference in productivity between two groups of students when implementing a traversal function for a linked binary search tree. Both groups used jGrasp, but only Group 1 got to use the Viewer feature. They also found a significant difference in accuracy (in terms of finding and correcting bugs placed in the linked binary search tree program) between these two groups. Both of these experiments measured the time it took students to complete their tasks. Cross II et al. also found that students who used the Viewer function had higher average test and exam scores than students in the other group.

***Lego Mindstorms***

Lego Mindstorms (http://mindstorms.lego.com) is a unique way to teach programming through robots (Hood & Hood, 2005; Lawhead et al., 2003). Instead of interacting with a visual environment on a computer screen, students can physically build and interact with robots. Lawhead et al. (2003) proposed a CS1 course using Lego Mindstorms robots to teach programming. Through such a kinesthetic approach, students were able to see the effects of their code. McNally, Goldweber, Fagin, and Klassner (2006) also noted that robots can be a physical embodiment of the objects students design. Students can identify the different classes that make up the robot system, such as its sensors and motors (Barnes, 2002).

Robots are constructed with Lego blocks, gears, wheels, sensors, and a control brick. Students write a program that controls the behavior of the robots, which can be based on input from sensors. For example, a student can write a function that moves the robot backward and turns it completely around if one of its bump sensors is hit. This program is downloaded into the control brick, which controls the gears that move the robot's parts. Unlike the previous visualization tools discussed, however, Lego Mindstorms require a large amount of physical equipment, and thus the idea of one robot per student can become infeasible and very expensive (McNally et al., 2006).

## Visualizations versus Metaphors

The previous section has surveyed five tools used in computer science education to enhance instruction through the use of visualization. They provide users with additional scaffolding by offering the concepts in different forms of representation. One form is visualizations, which are typically used to highlight each line of code as the application traces through it, as in jGrasp. These tend to be animation-based, as this can show a computer program in action. Some extensions, such as Jeliot 3, described in Moreno, Myller, and Bednarik (2005), also afford users a line-by-line visualization for the code.

Another form of representation used in teaching is through metaphors. A prime example is the Lego Mindstorms robots, which serve as the embodiment of an object; they serve as metaphors — individual lines of code are not highlighted or visualized through the robot. The base package of BlueJ and Greenfoot show representations of classes as diagrams and graphic characters, respectively. Metaphor-based tools do not visualize individual lines of code but rather offer a visual representation of a code object. Alice allows users to see the effects of their "code changes" through the animation playing. The objects users place in the scene are visual metaphors that are represented in the code. The "code" that users augment in Alice, however, is far simpler than that in jGrasp. One distinguishing feature of a metaphor is the context: the visualization not only serves to illustrate the code, but also adds another layer of abstraction. As examples, Alice has an animation of avatars and other objects being active within a scene; classes written in Greenfoot are represented as characters and objects in a simulation or game; BlueJ maps out the classes in a program and their interrelationships; and users write programs for their Lego Mindstorms robots to interact with their physical environment.

Though the general term "visualization tools" is applicable to both types, the distinction between visualizations and metaphors — the method of information delivery — allows us to be more specific in the discussion of tools. The difference is how code is represented to the user. To summarize, visualizations are used to illustrate code in more of a line-by-line manner whereas metaphors add another layer of abstraction in representing code; that is, the extra layer of abstraction is the metaphor through which students must understand the concept. For example, students may see a Lego Mindstorm robot as a metaphor for an object — but this representation is one level above the code that implements the object.

## THEORETICALLY-BASED DESIGN PRINCIPLES FOR MULTIMEDIA COGNITIVE TOOLS

Drawing from constructivist, multimedia, and motivational learning theories, and design principles for cognitive tools has provided a conceptual framework for how people know and learn, and how such knowledge and learning can be supported and enhanced. The following conclusions from the review of literature help to build a design framework for the implementation of multimedia-supported cognitive tools.

### Multimedia Cognitive Tools Should Adopt a Sensory Modalities Mode of Delivery

A sensory modalities delivery of multimedia offers instruction on both the auditory and visual channels. This works within Baddeley's (1992; 2001) model of working memory, which has two components for processing auditory and visual information: the phonological loop and visuospatial sketch pad. This assumption of working memory is an integral part of Mayer's (2001) CTML which is a basis for an MCT's design.

The tools discussed in the previous chapter focused on visualizations and targeting the visual channels. Despite the widespread availability of high-speed Internet connections, the file size of the MCT program must be kept minimal. If the sensory modalities delivery of multimedia is adopted, then audio and video files must be included, which will render the program rather large. Streaming such media files may still take a longer time. As Tabbers et al. (2004) noted, visual text may be more efficient than audio text, especially if these tools require download time. They observed that the longer download time for the audio increased students' boredom level. Since affect is another target of the MCT framework, it is imperative that large media files such as video and audio are kept to a minimum size to preserve the sensory-modalities view of multimedia learning.

Another aspect to consider is the redundancy principle, in which information should not be duplicated for more than one channel (Mayer, 2001). Since it can be a costly addition to MCTs' performance, audio is not a required channel in which to provide information. Rather, the audio channel should be kept open for whenever the verbal or pictorial channels have an excess of information.

## Multimedia Cognitive Tools Should Engage Students' Higher-Order Thinking and Problem Solving

The main advantage of cognitive tools is that learners' cognitive abilities can be extended beyond their own levels of development. Vygotsky (1978) stated that students can learn and function beyond their actual levels of development in the ZPD, provided they have the guidance of a more capable person (Vygotsky, 1978; Wells, 2000). In this case, students' knowledge structures are amplified by another person, thus achieving more than they can alone. Jonassen (2000) noted that Mindtools can support students in their ZPDs by giving them appropriate scaffolding and feedback. Thus, MCTs need to amplify a student's abilities by providing them with appropriate scaffolding. Scaffolding requires that the program provide only minimal help to the students when needed so they can complete the problem (Lajoie, 1993). Scaffolding can also be provided through the use of hints and clues.

Doube (2004) and Herrmann et al. (2003) designed their instructional modules to target different levels of mastery based on various taxonomies of educational objectives. Herrmann et al.'s (2003) study found that a course using differing levels of instructional modules reported higher grades than courses that had a one-size-fit-all type of instruction. Therefore, MCTs must provide information that is viable for varying levels of students and scaffolding that is appropriate. In adapting to the student's level, MCTs can also guarantee instructional activities that provide sufficient amount of challenge to keep the student motivated as well as maintaining a positive level of self-efficacy (Bandura, 1994; Malone & Lepper, 1987).

## Multimedia Cognitive Tools Should Engage Students in Metacognition

Metacognition helps students in reorganizing and refining their schemas (Greeno et al., 1996). The more intricate and complex one's schema is, the better one is able to approach and solve problems. Jonassen (2000) and Pea (1985) preferred the use of cognitive technologies to spark reorganization of students' thinking. Jonassen (2000) believed that cognitive tools such as Mindtools could make students think even more intensely. This is why cognitive tools are partners in learning — in order to facilitate complex thinking. To set off the metacognitive process, MCTs need to provide students with moments of meaningful disequilibrium, and this requires that students be able to have beta and gamma reactions (Ginsburg & Opper, 1987). Such reactions will make students aware of new situations and drive them to integrate such experiences into their knowledge structures. Cognitive curiosity can be a catalyst for metacognition by raising some doubt or the awareness of some flaw in a student's schema (Lepper & Malone, 1987). Evoking moments of sensory curiosity is highly doable within a multimedia environment through its visual and audio affordances.

Assimilation is an important step in integrating new knowledge into existing schemas. The CST views one level of knowledge as mental models in which memory objects are tied to a context or situation (Derry, 1996). Students can choose to come up with their own mental models based on what they already know or a teacher can provide modeling examples. MCTs should be able to help students with the construction of mental models. Doube (2004) provided students with real-world, everyday metaphors (e.g., fishing rod, tackle) that helped explain abstract computer science concepts. Lawhead et al. (2003) proposed how Lego Mindstorm Robots can represent various objects and programming concepts. Alice used game characters and everyday items (e.g., chairs, lamps) within a scene as metaphors for objects (Cooper et al., 2000). These tools provided meaningful representations that were relevant to students. BlueJ and jGrasp allowed students to see the visual representations of their object-oriented code as well as the Java code itself (Cross II et al., 2007; Kölling & Rosenberg, 1996), thus giving students different forms of representations of the same piece of code. Negotiating the commonalities in these different forms is part of the metacognitive process.

Similarly, the role of multimedia is to provide students with several viable and meaningful representations of the concepts covered. The selecting and organizing cognitive processes may be facilitated by multimedia. In Jehng et al.'s (1999) program, VisualScheme, graphics were used to point out relevant or important concepts. Such scaffolding allows students to decipher and select out what is most important.

Egocentric speech is also part of the metacognitive process (Vygotsky, 1978). Students may try to make sense of what is on the screen and try to internalize it by engaging themselves with inner dialogue. The scaffolding provided by the MCT gives students the adequate support needed to operate within their ZPDs.

## Multimedia Cognitive Tools Should Promote Student Autonomy

The role of the learner, in a constructivist learning environment, is active and learners are in charge of their own learning (Brooks & Brooks, 1993; Mayer, 1992). Students should therefore enable themselves to seek out people and additional resources when needed. MCTs are resources that should be available to students whenever and wherever they need them.

Telling students they can be autonomous is not enough, since teachers cannot expect students to take on that responsibility immediately or efficiently. In fact, MCTs need to promote the concept of autonomy, not just support it. MCTs can motivate students to take charge of their own learning and to use the tools. The MCT needs to be an interactive application, but it must be up to the user whether to choose to use it or not. MCTs must not place the learner in a passive mode; otherwise, MCTs would follow the information acquisition metaphor of multimedia learning in which the user is passively looking and, it is hoped, remembering the information. In this delivery media view of multimedia, MCTs would not be much more sophisticated than a slideshow (e.g., flashcards on computers). Optimal use of MTCs would ensure that students are engaged within a dialogic process with the MCT and within themselves via metacognition and egocentric speech. Knowledge construction relies on this step since students must process, assimilate, and integrate new knowledge through what they see in the MCT. MCTs need to use multimedia learning as a knowledge construction tool.

**Multimedia Cognitive Tools Should Provide Intrinsically Motivating Experiences**

First, as previously noted, if students find a program boring or useless, they probably will not use it. Malone and Lepper's (1987) design principles for intrinsically motivating environments are key to luring students into using the MCTs and then engaging them for continued use. Second, negative affect can have a negative influence on student performance (McKinney & Denton, 2004; Moskal et al., 2004). One instructional tool that can be used to trigger bouts of situational interest is to play on student curiosity. Multimedia can entice students on multiple sensory levels through flashy graphics or sound effects, as Doube (2004) and Lawhead et al. (2003) proposed, using everyday objects to transfer students' individual interests in one area to computer science. Conway et al. (2000) used the interactive 3-D graphics tool, Alice, to reach a broad range of students, not just the computing/mathematically-inclined students. Alice now uses characters from the popular game, The Sims 2 (http://thesims2.ea.com), as part of its library and this is yet another way to entice students.

MCTs need to offer an environment in which students can freely roam and explore without much restriction (e.g., time limits). Control is an essential feature for an intrinsically motivating environment (Malone & Lepper, 1987). This idea is similar to allowing a student to freely navigate a website, click on any desired link, and follow any path. Content can be provided in a nonlinear sequence that allows students to construct their own sequences of instruction, and thereby creating multiple perspectives on a concept (Driscoll, 2004). Such an environment provides students with the perception of control conducive to student autonomy and maintains an intrinsically motivating environment. Microworld environments such as Greenfoot and Alice provide students the power of constructing their own games, worlds, and animations (Conway et al., 2000; Henriksen & Kölling, 2004).

## Summary of Design Principles

The principles listed below are based on general cognitive tool design and are further informed by constructivist, multimedia, and motivational learning theories. According to the literature survey, such a design provides students with a computer program that helps them transcend their cognitive limitations while facilitating knowledge construction of abstract concepts.

1. Multimedia cognitive tools should adopt a sensory modalities mode of delivery.
2. Multimedia cognitive tools should engage students in higher-order thinking and problem solving.
3. Multimedia cognitive tools should evoke metacognition.
4. Multimedia cognitive tools should promote student autonomy.
5. Multimedia cognitive tools should provide intrinsically motivating experiences.

These summary statements also serve as a framework of principles that instructional designers can use to develop MCTs. Each of these five principles captures the most important and relevant aspects of applying computer-based tools that use multimedia to affect cognition. An MCT can involve any number of elements, activities, or features, but all of them must adhere to one or more of the principles. Although these principles were derived from theory and past research, they have never been evaluated as a whole in practice. CSNüb, as mentioned in Chapter One, was the first implementation of the MCT framework. Therefore, assessment of the MCT framework starts with assessment of CSNüb.

# Chapter 3: Methodology

## Introduction

This chapter explores the present study's methodology, which was informed by the epistemological assumptions that arose from the research question. These assumptions influenced the selection of methods for collecting data, procedures for carrying out the study and analyzing the data, and assurance of rigor and trustworthiness of the study's results.

## Epistemological Assumptions of the Research Question

In review, the main research question in this study is as follows:

*How does using CSNüb affect the conceptual understanding of OOP for students who are novices to object-oriented programming?*

This question seeks to explore and discover emerging trends and outcomes of novice students using CSNüb with respect to cognition. Not only does this question pertain to CSNüb, it also relates back to the MCT framework. These outcomes are not definitive nor do they imply a hypothesis; rather, the question and its answers are mainly exploratory and open-ended.

The research question assumes a constructivist perspective on learning. Therefore, at the foundation of this study's theoretical framework is the constructivist notion that knowledge and meanings are constructed by individuals and through interaction with tools. These assumptions are applicable to the learning objectives of an instructional design and they also affect this study's methodology via its epistemological stance. Truth, knowledge, meanings, and realities are a product of social construction in constructivist ontology (Mertens, 1998).

Crotty (2003) referred to this reality as constructionist in his explanation of different epistemologies that affect research epistemology. Constructionism is a middle ground between objectivism and subjectivism. An objectivist epistemological stance views knowledge as external to the human consciousness or emotions. A subjectivist epistemology has an "anything goes" perspective, contending that an individual's interpretation of an object is mediated by that individual's interaction with anyone or anything aside from the actual object. In a constructionist perspective, knowledge is constructed from an individual's interaction with some object. Moreover, Crotty (2003) differentiated between constructionism and constructivism, with the former being aligned with the social construction of reality and the latter being aligned with the individual. As there are many different interpretations and labels, Mertens (1998) categorized these beliefs under "interpretivism/constructivism." Since these stances are regarded as complementary — each accepts alternate realities — the general term constructivism is applied to this study's epistemology.

The research question and its epistemological stance imply that a qualitative approach be taken in order to understand the phenomena: "The methodological implication of having multiple realities is that the reason questions cannot be definitively established before the study begins; rather, they will evolve and change as the study progresses" (Mertens, 1998, p. 15). Following this assumption, constructionist research does not begin with a hypothesis based on theories and past research that is to be tested. Instead, such ideas and theories arise from the study itself and are grounded in the data collected (Strauss & Corbin, 1998). This includes the use of interviews, focus groups, documents, videos, recordings, and so forth as data for analysis (Mertens, 1998; Strauss & Corbin, 1998). Using an interpretivist theoretical framework, data collection methods such as interviews and focus groups require the researcher to interact with participants, and sometimes be involved within the context of the phenomena being studied, while remaining unobtrusive (Lancy, 1993; Mertens, 1998). Since constructivist ontology accepts that realities are constructed, the researcher also brings his beliefs, which may assert themselves in the analysis of the data. Such personal biases should be declared at the outset of the investigation to show the lens through which the researcher is conducting the study.

Following this constructivist assumption, the clinical methods discussed by Ginsburg (1997) are most appropriate for exploring and analyzing students' mental models, since the research question asks how knowledge is constructed and transformed through CSNüb. Observation and analysis of these cognitive processes require data collection methods such as the process tracing methods discussed by Hayes and Flower (1980; 1983). Superimposing process-tracing methods on top of the clinical interviews allows the researcher to see how users apply their knowledge in addition to gaining insight on how their knowledge is structured.

## PARTICIPANTS

Participants were recruited from a semester-long CS1 and CS2 class from a large four-year research university in Texas. The CS1 course focused on the basics of computing and a programming language at the beginning of the course. OOP was introduced towards the last third of the course. The CS2 course covered data structures, algorithms, recursion, correctness, and OOP using the Java programming language. The first half of CS2 reviews the topics covered in CS1.

These particular CS1 and CS2 courses were the first courses in the computer science curriculum at this university. Both courses were chosen on the basis that either class could be the first computer science course taken on the undergraduate level. Some students were exempt from taking CS1 due to previous experience. In either case, students in these courses were in their first undergraduate year of study of CS and were relatively new to object-oriented programming. Data collection began roughly 2 to 3 weeks after the topic of OOP was covered in each course.

## Description of Participants

There were 12 participants in this study (n=12). Only male students volunteered for this study. Though participation was entirely voluntary, each participant was paid $40 for his involvement with the study. Table 3.1 contains the demographic breakdown for each participant. The names listed for each participant are pseudonyms.

Participants' ages ranged from 18 to 23 with a mean age of 19.25. All participants were undergraduates at the time of this study and ranged from first year to fifth year students with a mean year-in-school rank of 1.833 years. Participants self-reported on how well they knew general programming and object-oriented programming using the following values: *3 – Excellent, 2 – Good, 1 – Fair, 0 – Needs Improvement*. Overall, participants self-reported "fair" to "good" understanding of general programming knowledge (1.583) and object-oriented programming knowledge (1.667).

Nine participants said they were solely majoring in Computer Sciences. This count includes those that were enrolled as pre-CS students who have yet to be officially enrolled in the CS program. Two participants, Henry and George, were double majoring in both Computer Sciences and Mathematics. Frank was majoring in Computer Engineering, but was pursuing a minor in Computer Sciences. Only two participants, Frank and Louis, reported working with Adobe Flash before the study. All participants were taking their first computer science course on the university level.

| Participant | Age | Year | Programming Knowledge | OOP Knowledge | Flash Experience | Major | Current CS Course |
|---|---|---|---|---|---|---|---|
| Alex | 18 | 1 | 2 | 2 | No | CS | CS2 |
| Brian | 19 | 1 | 2 | 2 | No | CS | CS2 |
| Chris | 20 | 3 | 2 | 1 | No | CS | CS2 |
| Daniel | 19 | 1 | 3 | 3 | No | CS | CS2 |
| Ethan | 19 | 2 | 0 | 1 | No | CS | CS2 |
| Frank | 18 | 1 | 2 | 2 | Yes | CE | CS2 |
| George | 23 | 5 | 3 | 3 | No | CS/Math | CS2 |
| Henry | 18 | 1 | 1 | 2 | No | CS/Math | CS2 |
| Isaac | 20 | 3 | 1 | 1 | No | CS | CS1 |
| Jared | 21 | 2 | 1 | 1 | Yes | CS | CS1 |
| Kyle | 18 | 1 | 1 | 1 | No | CS | CS1 |
| Louis | 18 | 1 | 1 | 1 | No | CS | CS1 |
| Minimum | 18 | 1 | 0 | 1 | | | |
| Maximum | 23 | 5 | 3 | 3 | | | |
| Mean | 19.25 | 1.833 | 1.583 | 1.667 | | | |
| Std Dev | 1.545 | 1.267 | 0.900 | 0.778 | | | |

Table 3.1: Demographic data for study participants

## THE CSNÜB ACTIVITY

As briefly described in Chapter 1, CSNüb is a game template written in Flash and ActionScript 2.0 which students use and work with to build a simple role-playing game while achieving a better understand object-oriented programming. The design of the tool and the activities were based on the MCT design framework. CSNüb consists of a base file with graphics and animation and the foundational code. Students mainly work with the code by creating and modifying existing class files. Their interaction with the multimedia aspects of CSNüb is restricted as not to introduce an entirely new concept (multimedia authoring) to this activity.

## Topics for Study

Since there are many topics within OOP, it was necessary to focus the objectives of the activity to specific topics within topics. As a discussant in the “Killer” Examples workshop at OOPSLA 2007 (the Object-Oriented Programming, Systems, Languages, and Applications Conference) in Montreal, Quebec, one of our tasks was to have a discussion on student mental models for understanding OOP. Our group, comprised of computer science professors and students, came up with a rough sketch of how mental models are created. Figure 3.1 is a reconstruction of the diagram we created at the session.

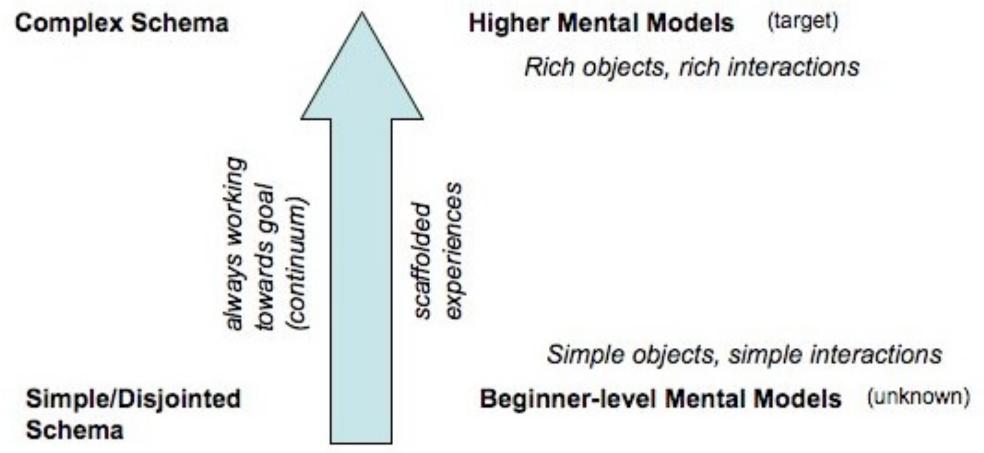

Complex Schema
Higher Mental Models (target)
Rich objects, rich interactions
always working towards goal (continuum)
scaffolded experiences
Simple objects, simple interactions
Simple/Disjointed Schema
Beginner-level Mental Models (unknown)

Figure 3.1: Student Mental Models for Understanding OOP

The topics at the bottom of the figure represent the most fundamental knowledge of what our group expected students in CS to know: simple objects, simple interaction. Since CSNüb is geared towards beginning and struggling students, its objectives targeted the “simple objects, simple interaction” part of the mental models continuum. Object-oriented programming is a very broad topic, but its distinguishing and foundational components are encapsulation, inheritance, and polymorphism (Ben-Ari, 1996). In the "objects-early" approach, Bruce (2005) suggested that the curriculum should focus on encapsulation, message sending, and dynamic method invocation. The latter facilitates the object interaction that was discussed at the workshop. Thomasson, Ratcliffe, and Thomas (2006) also implied that class relationships must have a place early in the curriculum. This relates to design issues of object-oriented systems where object relationships can be complex and numerous, and it relates to inheritance as well. The CSNüb activities were designed to address the concerns, with respect to topics, of these discussions: the tasks focus on basic levels of encapsulation, inheritance, and simple object interaction (see Appendix A for specific instructional objectives of CSNüb with respect to these topics).

## Tasks

Participants were asked to complete four[1] tasks. Each one implemented a feature of the game using CSNüb.

### *Task 1 - Giving life to the Submarine and Squid*

Task 1 required the participants to delve right into CSNüb's structure and into using inheritance. Participants were asked to give the Submarine and Squid specific values for HP, AP, and DP. The correct implementation required the participant to use the *hit_points, attack_points,* and *defense_points* variables in the CSNub_Character class from which CSNub_Submarine and CSNub_Squid extend.

[1] The original activity had six tasks. The original Task 1 was used to re-familiarize participants with CSNüb and the Flash environment, and was removed from analysis. Tasks 4 and 5 of the original activity were combined in this data analysis and labeled as Task 3 since they were two parts of the same task.

***Task 2 – The Submarine versus the Rock***

Task 2 asked participants to deal with interaction between two objects: the submarine and a rock. First, participants were asked to create a new class for CSNub_Rock. Next, participants were asked to rotate the submarine 180 degrees, displace the submarine by a set amount of pixels in its new direction, and then decrease the submarine's *hit_points* by one when the submarine runs into a rock. Task 2's implementation set the tone for how the submarine's collision with other objects would be implemented in subsequent tasks. Since only simple interaction between objects were required at this level, virtually all interaction was simplified to the point where all interaction code need only be placed in CSNub_Submarine's *intersect()* method.

***Task 3 – The Submarine versus an Energy Barrel***

Task 3 asked participants to implement the interaction between the submarine and an energy barrel. This was similar to Task 2. Participants had to create a new class for CSNub_EnergyBarrel. The effect of the interaction is that the energy barrel disappears and the submarine's *hit_points* increase by 3 points. It was in this task that participants had to be able to distinguish between different objects that the submarine can intersect.

***Task 4 – The Submarine versus a Squid***

Task 4 asked participants to implement the fight sequence between the submarine and a squid. Participants were given formulae for how to determine the damage one would inflict on the other using each character's *hit_points, defense_points*, and *attack_points*.

## INSTRUMENTS

The data collection methods for this study were clinical interviews and process-tracing methods. These methods were selected because of their ability to answer the research question and to delve into the effects of the intervention for each participant. Additionally, a demographic survey and a rubric-based score of participants' solutions created in the clinical interview activity served as supporting quantitative data.

### Clinical Interviews

Clinical interviews are based on Piaget's discontent for standardized measures for measuring student intelligence and achievement: "The clinical interview can be used to examine different aspects of the child's (or adult's) thinking, including the understanding of basic concepts of number, complex ideas about reality, moral judgment, and solutions to IQ test items" (Ginsburg, 1997, p. 38). The purpose of the clinical interview was to probe the participants' minds and discover how they come to know and understand a given concept (von Glasersfeld, 1984). Through this method, it was possible for the researcher to construct a model of an individual's knowledge structures on that topic and how it may relate to their other structures. Primarily, the researcher would have each participant work out problems while thinking aloud and being asked probing and explanatory questions. Clinical interviews were individualized for each participant; thus, although interviews for each participant might start off the same and have the same agenda, the process might be adapted, changed, or take a different course depending on the individual (Ginsburg, 1997; von Glasersfeld, 1984). Though Piaget and Ginsburg conducted such studies with children, these methods are also applicable to adults.

This study used Ginsburg's (1997) guidelines to carry out clinical interviews. Three areas of concern were kept in mind during the study. First, the well-being — the affective state — of the participants was always of concern. Ginsburg emphasized keeping the participants comfortable during this process by developing a good rapport, informing them of what the study entails, using language appropriate for their level of development, and beginning with achievable goals. Thus, the researcher must serve as a coach, providing support, motivation, and encouragement; should a participant feel negative affect towards any topic, those feelings may have an adverse effect in shaping his or her understandings.

Second, the interview must produce rich data that emerge from the participants themselves. For this reason, Ginsburg promoted the autonomy of participants, cautioning the researcher to avoid the tendency to teach or correct the participants. If a participant has taken an incorrect route in performing the task or has provided an incorrect answer, those misconceptions must also be explored to see where they began and how they have developed and affected learning (Smith et al., 1993). Ginsburg asked that participants be introspective and retrospective, and that this introspection and retrospection be done aloud so that the researcher can hear these thought processes. The researcher must also probe the participants about their answers or explanations. As mentioned above, correct and incorrect understandings must be explored to ascertain how they were conceived and why participants feel justified in what they believe to be correct.

Third, the type of questions asked during a clinical interview is important. One of the most important roles the researcher must assume in a clinical interview is that of an observer who must watch participants as they complete tasks. As Ginsburg suggested, the clinical interview was limited to fundamental questions that mostly asked the participant to verbally demonstrate their understanding. The researcher should not ask leading questions that would make the participant do something he or she would not normally do, so as not to influence or taint their responses. Also, the participants should do the majority of the talking so that the researcher can draw from a rich set of data to construct a model of the participants' minds.

For this study, the clinical interview set up the environment in which the participants were observed completing tasks. These tasks will be discussed further in the chapter. While participants underwent the clinical interviews, process-tracing methods were also applied, mainly for the purposes of gathering observation data on what participants did with the tool, in order to add these data to the verbalized reflective data.

## *Process-Tracing Methods*

The process-tracing methods used by Hayes and Flower (1983) explored the cognitive processes involved in writing. Though process-tracing methods are used in writing research, there are similarities in the composition and problem-solving (in CS) processes that warrant their use (Yuen, 2007a), According to Hayes and Flower (1983), there are multiple stages to the writing process that are easily adaptable to problem solving in computer science. The first stage is planning: before pen is put to paper, how do students plan out their composition? In computer science, students are normally taught to think out their solutions before they begin to type up their program on the computer. The second stage is translation from planning into an attempt to produce a composition. In computer science education, this is the stage in which students start to write in code on the computer. The third stage is review of the composition in which students must evaluate or proofread their own writings and make any necessary revisions or adjustments. The computer science equivalent has students debug their own code by attempting to compile it. Should the compiler return any errors or bugs, students must make the necessary corrections before the program can run. The final stage in the writing process is the monitoring stage in which students must decide which of the other stages they are in: planning, translating, or reviewing. As in writing, computer science students must decide if they will plan out their solution first or just start typing code. How they decide to solve their problem depends on their previous knowledge: the strategies they know or the methods they have been taught.

Whereas clinical interviews were used to assess and evaluate a student's level of understanding, process-tracing methods focus on mapping out the cognitive process. The goal is to come up with a "program" of the flow of the cognitive process (Lancy, 1993). To do this, a researcher must find out the strategies and methods people employ. Protocol analysis methods are used to determine how students attempt to complete a task or solve a problem. Such strategies may not be the most effective or even correct but it is of interest to look at the strategies and processes that students undertake to negotiate these understandings.

There are four categories of process-tracing methods: behavior protocols, retrospective reports, directed reports, and think-aloud protocols (Hayes & Flower, 1980, 1983; Lancy, 1993). These are task-oriented protocols in which the participant is asked to perform a task while the researcher is making specific types of observations (Hayes & Flower, 1983; Lancy, 1993). Behavior protocols look at what the participant is doing when performing a task and not necessarily the internal cognitive process. Retrospective reports ask participants to give a review or summary of what they have done after completion of the task. Think-aloud protocols ask participants to verbally communicate their thought processes to the researcher while completing the tasks. Directed reports are similar to the think aloud protocol with the exception that the researcher is only interested in specific aspects of the task, and thus the participant only thinks aloud when confronted with those aspects while performing the task (Hayes & Flower, 1983). Though Hayes and Flower (1983) used process-tracing because of its abilities to look at the cognitive processes in writing, the rationale for its inclusion is due to the similarities between the composition (in writing) and problem-solving (in computer science).

The purpose of including process-tracing methods in this study was to provide a structure for the clinical interviews. The exclusion of directed reports was due to the desire to explore all aspects of the cognitive processes in understanding OOP instead of limiting the data. The process-tracing methods gathered the following types of data within the clinical interviews.

1. Behavior protocols – These observations were compiled during the clinical interview and afterwards in the review of the video recordings. These are physical actions that participants engaged in during the task. Such actions included designing on paper before coding, continually testing their games to check for correctness, plug-and-test coding, looking through ancestor files for code hints, etc.
2. Retrospective reports – Participants reviewed and summarized their solutions as they would in the review stage of the writing process. Retrospective protocols were useful for member checking in which the participant could be asked questions for clarification and confirmation of any observations made throughout the interview.
3. Think-aloud protocols – This is equivalent to the think-aloud/questioning by the researcher in clinical interviews and is redundant.

### *Rubric*

To gain a holistic picture of student cognition, the source code from students' final CSNüb solutions was assessed. The rubric used to assess the solutions was based on the Primary Traits Assessment (PTA) that Cable (2001) used in her junior level course on object-oriented programming. With respect to her students' programming assignments, Cable identified seven primary traits as matching behavioral objectives for a successful student: specification, GUI design, class design, documentation, correctness, re-use, and testing.

In adapting Cable's PTA to fit the needs of the present study, only the traits class design, documentation, correctness, and re-use were applicable. Participants were not expected to create a specifications document since those were already provided. They did, however, need to have an idea of how their design should be implemented. Since interface design is not a primary goal of CSNüb, the GUI design component was not used. Another modification to Cable's PTA was changing its language focus from Java to ActionScript, the Adobe Flash environment, and the CSNüb template. The last modification was removal of the testing trait, since in the present study students were not required to write a test suite. Participants were expected to periodically test their code, although not in a formal fashion; they did this by running their games and checking for errors in the output or incorrect behaviors in the game.

Table 3.2 outlines the modified PTA rubric used in assessing the artifacts created by the participants. With the exception of re-use, all the traits are scored on a 3-point scale. Re-use is scored on a 2-point scale. The 1pt value for Re-use was modified to include variables in addition to classes and methods re-implementation.

**Class Design**
3 – Students design appropriate classes and make appropriate decisions about the use of composition and inheritance.
2 – Most design decisions are appropriate.
1 – Several design decisions are inappropriate.

**Documentation**
3 – The program contains a comment for each public class and for each public member of a public class. The comments are correct and unambiguous, explaining "what" not "how". Spelling and grammar are correct.
2 – Most necessary comments are present, correct, and unambiguous.
1 – Several comments are missing or wrong.

**Correctness**
3 – The program implements all required features. The program behaves correctly for both typical and unusual (but correct) input. The program also handles bad input appropriately.
2 – The program fails to implement some minor feature in the specification. The program behaves correctly for all typical input and most unusual input.
1 – The program does not behave correctly for some typical input or fails to implement a major feature or two or more minor features in the specification.

**Re-use**
2 – The program makes appropriate use of the CSNüb's predefined classes and object-oriented framework to create new classes
1 – The program re-implements classes, variables, and/or methods available in the CSNüb template or the program uses CSNüb classes incorrectly.

Table 3.2: Primary Traits Assessment (PTA) rubric from Cable (2001) adapted for CSNüb

### Demographic Survey

Participants were asked to complete a paper survey (Appendix B.1). They reported their gender, age, year in school, major(s) and minor(s), and experience in programming, OOP, and Flash.

## PROCEDURES

Participation consisted of two sessions. The first session gave participants a tutorial on the Flash environment and introduced them to the template. These tutorials were done informally, either in a group or individually. The demographic survey was distributed at this time.

The second session consisted of individual interviews, which ranged from 2.5 to 4 hours. These sessions commenced within two weeks of the first session. During these interviews, participants were asked to complete a series of tasks, which implemented features within their game. As required by the structure of clinical interviews and process-tracing methods, participants verbalized their thinking. At times, the researcher must sometimes elicit speech from the participants since thinking aloud may not be natural process. The researcher must also probe with questions. The interviews were semi-structured in that there was a set of standard questions asked during the interviews. Examples of questions include: *What are you doing?*, *Why did you do that?, How do you plan to do this?*, and *What happened here?* Also, a more "emphatic approach" to questioning was taken in that the line of questioning depends on the context and interaction with the participants (Fontana & Frey, 2005); that is, questions were changed, asked at different times, and occasionally discarded or added depending on the immediate situation, thus making the questioning more unstructured and customized to the individual. This method of questioning is also required in clinical interviews.

At the conclusion of each task, participants were asked to summarize their work. Observations and field notes were recorded. All interviews were video recorded to capture sound, the computer screen, and the physical actions of the participants.

## Role of the Researcher

There were moments in the data collection process in which I had to be involved in the problem-solving process. I was careful to follow Ginsburg's (1997) guideline stating that the researcher does not teach during clinical interviews. Due to the complexity of the activity, however, I had to provide participants some guidance during these times in the following areas:

1) Syntax errors – Very often, participants would have syntax errors that involved spelling errors, missing or unmatching parentheses in method calls, missing or unmatching braces in method or class bodies, etc.
2) ActionsScript reserved words – ActionScript has an abundance of reserved words that I had participants avoid when coming up with their own identifiers (e,g., the identifiers x, y, etc.).
3) *instanceof* – I did not expect participants to know of the *instanceof* operator. When they asked if there was a way for them to differentiate between object types, I provided the *instanceof* keyword for them. Many knew to use some equivalent of *getClass()* — Java's version of *instanceof*.
4) Serious errors – When participants suffered serious errors in their programs such as those that result in crashing Flash, I had to intervene to resolve this problem. These problems generally resulted from unsaved or misdirected files.
5) Stalled progress – In many cases, the task might be beyond the ability of the participant. Depending on the specifics surrounding the stalled progress and the participant's affective state, I had some participants put the task on hold and move on to another task or just accept whatever they had completed. This depended on a real-time assessment of what I believed the student could actually handle and accomplish.
6) File linkage and object registry – Participants were reminded that they had to link their class files to the symbols in the Library as well as put them in the

object_registry. This was an additional step required by the nature of Flash and CSNüb's support of event-driven applications.

## SOURCES OF DATA

Although much data was collected during the interviews, only a subset proved to be especially illuminating. A grounded theory approach was used in the analysis of the data below.

### Behavioral Protocols

The main source of data was the behavioral protocols of what participants did throughout the activity. Behavior protocols were organized into a log of individual actions committed by each participant while using CSNüb to completing the tasks. Such actions were observed during the original clinical interviews and reviewed through the video recordings. Actions were also triangulated by field notes, student source code, and/or interview transcripts. One of the criticisms of think-aloud methods is that the subject has already reflected when he or she speaks, and thus what was said may not a true indicator of what was being thought at the time (Hayes & Flower, 1980, 1983). In breaking down the data for microanalytical purposes, therefore, it was important that each action was devoid of the researcher's assumptions and of participant thought as much as possible. Researcher and participant thoughts were, however, included as a context for the action and for descriptive purposes. Table 3.2 contains sample entries from the behavioral protocol logs.[2]

[2] The behavioral protocol logs entries represent the original task numbers and participant numbers. The task numbers have been re-assigned for this dissertation, and the participant numbers were replaced with pseudonyms.

| Part. # | Action # | Task # | Description | Notes | Codes |
|---|---|---|---|---|---|
| 3 | 14 | 3 | LOOKUP CSNub_Obstacle. He is checking this class first: "I assume there's going to be multiple obstacles in this game and I don't know exactly an obstacle should do yet. So, if I look at the comments and the methods provided in the parent class, I'll know what a rock is supposed to do." | I have to make the assumption that he is aware of the hierarchy and thus, was able to assume the hierarchical relationship between CSNub_Obstacle and CSNub_Rock. | exploration, lookup, hierarchy, object relationship |
| 8 | 85 | 6 | He tried to call CSNub_DisplayPanel's setText function even though he can't access that object from CSNub_Submarine. | He doesn't see that the intersect method returns a string that is displayed in the display_panel. | visibility, bigger picture, object interaction |
| 11 | 1 | 2 | LOOKUP Tutorial Activity. He's in the Tutorial Activity to find out more information on the HP and the other values in the game: "I'm looking for like, I guess, how this works how I use the HPs, APs and what all that means." | He is trying to have a better understanding of the architecture of the game. He remembers that in the Tutorial Activity, we went over controlling the submarine. So, he's looking for similar information on the values. | lookup, exploration, bigger picture, strategy, prior knowledge, architecture, planning, design |

Table 3.2: Example of behavioral protocol log entries

## Transcripts, Field Notes, and Memos

Only the transcripts covering the clinical interview activity were used for the analysis. Whereas behavioral protocols provided a record of what the participants actually did, the transcripts provided the reflection and thought behind the action performed. Field notes were taken throughout the interviews. In the field notes, I highlighted what I thought was of importance, recorded any theories that I might have developed while observing a participant, and took general notes on what a participant was saying and doing. Field notes also included my post-interview reflections, which summarized the interview, suggested some theories on what been occurring, and noted any questions that might have been raised through the interview.

### Demographic Data and Quantitative Measurements

To obtain a more holistic view of the data, demographic data and quantitative measures were included for further descriptive and illustrative purposes where required. Quantitative measures included timing how long it took participants to complete the tasks and the entire activity, the number of times they performed a specific action (e.g., opening a file), and the rubric based on the primary traits assessment (Cable, 2001) of the final product.

## VALIDITY

Lincoln and Guba (1985) addresses the concerns of credibility, applicability, dependability, confirmability, and neutrality in qualitative research. An auditing system that followed Lincoln and Guba's concerns was put in place to establish the trustworthiness of this study's findings, using peer reviews, peer debriefing, inquiry audits, and stepwise replication. The multiple sources of data (e.g., videos, behavior protocol log, transcripts, and field notes) were used for triangulation purposes. In addition to these traditionally qualitative procedures for ensuring trustworthiness, a quantitative tool was also used: the *kappa* statistic. As this analysis was dependent on coding, the *kappa* statistic seemed like a complementary reliability checker since it also deals with coding and categorization.

### Kappa Statistic

Cohen's *kappa* (unweighted) statistic was used to test for the reliability of findings at the open coding stage of data analysis (Cohen, 1960). The *kappa* test was done at this stage due to a high level of subjectivity in assigning codes to each action item in the behavioral log, and because that set of codes is the foundation for the synthesis of findings. Cohen's *kappa* statistic tests for agreement within a coding scheme between two coders (I refer to the coders as reviewers).

According to Landis and Koch (1977) the *kappa* statistic yields a value between 0 and 1, where 1 means there is perfect agreement between both coders and 0 means that there is no agreement. Table 3.3 shows Landis and Koch's interpretation of the *kappa* statistic.

| *Kappa Statistic* | *Strength of Agreement* |
|---|---|
| < 0.00 | Poor |
| 0.00 – 0.20 | Slight |
| 0.21 – 0.40 | Fair |
| 0.41 – 0.60 | Moderate |
| 0.61 – 0.80 | Substantial |
| 0.81 – 1.00 | Almost Perfect |

Table 3.3: Interpretation of *kappa* statistic (Landis and Koch, 1977)

The *kappa* test was done on two sets of data: the behavioral protocol log and the source code. Due to the amount of data, reviewers only audited approximately 20% of the behavioral protocol log and source code. There were a total of three reviewers.

## Behavioral Protocol Log

Two auditors were used to audit the behavioral protocol log: both were graduate students who conduct research and who identified themselves as qualitative researchers.

Reviewer A: Ethan for Tasks 1 – 4; Brian for Tasks 1 & 2

Reviewer B: Chris for Tasks 1 – 4; Henry for Tasks 3 & 4

Each reviewer was given only the behavioral protocol log for the participants and tasks he or she had been assigned. These logs consisted of the task number, description of the action, and related field notes or reflection excerpts.

Reviewers were provided with a list with the following codes/categories, which arose from the open coding process: exploration, refinement, awareness, scaffolding, and disequilibrium. Since there were other codes that were eventually subsumed into the final set of codes, "sub-codes" were provided along with a description of each code (See Appendix D.1). Reviewers assigned one code to each action item in the behavioral protocol log, with an option to sort the action item into "none of the above" if none of the available codes fit.

Cohen's *kappa* statistic assumes that all the codes are mutually exclusive. My initial open coding analysis yielded multiple codes for each action item. In the selective coding stage, as I narrowed down the findings, I tried to come up with a set of five cognitive processes and actions that result from using CSNüb. Although this may diminish the value of the *kappa* statistic, the five codes are related to and interact with each other. Even though there were three separate reviewers, Landis and Koch's (1977) adjustment for multiple coders was not used since none of the reviewers' data overlapped. The *kappa* value found was 0.7741, which implies substantial agreement between the auditors and myself on the open coding process.

**Source Code**

Reviewer C was responsible for auditing the PTA scoring of the source code that the participants edited or created. Reviewer C was a professional software developer who reviewed the following participants' work: Alex, Frank, Daniel, George, Isaac, Jared for Tasks 1 – 4. Reviewer C has had experience with CSNüb. The reviewer was provided with the participants' source code, the base code for CSNüb, the hierarchy of CSNüb class diagram, a cheat sheet on the basic syntactical differences between AS 3.0 and Java/C++, the participants' games, and the PTA rubric (see Appendix D.2). The *kappa* statistic was computed for each of the four categories: class design, documentation, correctness, re-use, and the total PTA score. An unweighted *kappa* was used even though each of the PTA score values subsumes the lower value — that is, a score of 3 assumes that the solution meets the requirements of the score of 2 and 1. This was due to the low number of data with which any weights did not affect the calculation.

For class design, the *kappa* statistic was .7143, which implied substantial agreement between the auditors. Out of six possible match-ups, the reviewer disagreed with only one score. For documentation, the *kappa* statistic was not computed since the reviewer agreed with all scores, and those scores were 1. For correctness, the *kappa* statistic was 1.00, which implied mutual consensus on all scores. For reuse, the *kappa* statistic was .5714, which implied moderate agreement between the reviewer and my original scores. Out of six possible match-ups, the reviewer disagreed with only one score.

## SUMMARY

This chapter has presented an overview of the methods used to carry out the study. All participants were first year, undergraduate students who were taking a computer science course. Participants were asked to implement four features of a role-playing game in CSNüb within the setting of a clinical interview. Process-tracing methods were used to observe and record what students were actually doing and how they were interacting with the MCT to solve the problems. The next chapter discusses the procedures for analyzing the data (a grounded theory analysis was used) and the findings.

# Chapter 4: Data Analysis and Findings

## Introduction

The analysis of data followed an interpretivist approach with the goal of discovering and uncovering patterns as to how CSNüb facilitated and mediated conceptual understanding of object-oriented programming. Due to the vast amount of rich data gathered for this study, it was necessary to focus on one part of the data as the primary source of data. Aside from the transcripts, it became apparent that the bulk of the data lay in the behavioral protocols: these were the focal point where all the other data met, and best served to describe how CSNüb affected conceptual understanding of OOP.

While behavioral protocols served as the main source of data for the analysis, field notes, source code, transcripts from the clinical interviews, demographic data, and quantitative measurements served as supplemental data for triangulation purposes and to provide additional context to the data. Glaser (2001) suggested that factual data such as quantitative measurements may decrease the conceptual and abstraction power of any generalizations or theories created from the data. Nonetheless, such factual data will be included in the present analysis to provide a more holistic picture of the participants in this study.

## Analytical Procedures

Several categories of behavior emerged during the open coding process. As the properties and dimensions of each category were being discovered through axial coding, it became apparent that many categories shared similar traits. This led to many categories being merged or subsumed into others. For this analysis, only the general categories that were noted in at least 8 of the 12 participants (75%) are discussed in order to avoid patterns that may be too incidental or specific to special cases.

In coding the data, it was important to keep in mind the conceptual framework within which this study was conducted. First and foremost, this was a cognitive study. Next, this study explored how a multimedia-based cognitive tool affects learning. Lastly, this study examined the various stages in students' understanding of OOP. Although these ideas formed the lens of the analysis, they did not exclude other perspectives that might arise from the data. Throughout the analysis, I planned to look for trends and patterns within the data that could explain how students work with CSNüb to facilitate and transform their understanding of OOP. Further, I wanted to examine the nature of such patterns and their causes and effects with respect to the learning process. Doing so could result in the construction of a model or theory about students' cognitive processes when using CSNüb.

## Grounded Theory Analysis

The analysis utilized a grounded theory approach in which generalizations and patterns describing a phenomenon may emerge from the data. The benefit of a grounded theory approach is that the emergent theories in this study will help inform the design framework for multimedia-based cognitive tools. Grounded theory "comes from data, but does not describe the data from which it emerged" (Glaser, 2001, p. 4). As Glaser vehemently argued, description is not grounded theory — grounded theory is not just about describing and categorizing data. In both describing and categorizing, no theory is being built from the different descriptions and categories (Strauss & Corbin, 1998). In contrast, grounded theory is about the conceptualization of social processes and actions. As Glaser (2001) wrote, "Conceptualization is the medium of grounded theory for a simple reason: without the abstraction from time, place and people, there can be no multivariate, integrated theory based on hypotheses" (p. 13).

In this study, I was exploring the interaction between students and CSNüb, which is an attempt at being a multimedia cognitive tool, and how that affects conceptual understanding. What I hoped to find in a grounded theory analysis was a general conceptualization of how those interactions developed and how they related to other outcomes that arose from those interactions.

## Coding Procedures

Since grounded theory emerges from the data, a systematic way of analyzing the data was needed. Such an analysis could give rise to patterns and concepts within the data while also uncovering different facets and relationships of those patterns and concepts. I began with a microanalysis of the data, specifically, of the behavioral protocol logs. Every action, including relevant quotes, was examined and coded. This detailed analysis broke the data into many small parts, which were individually analyzed for meaning and then for how they related to the whole (Strauss & Corbin, 1998). Each piece of datum was given a label (or code) that served as the basis for conceptualization of the phenomena.

As Richardson (2005) observed, "Coding is not merely to label all the parts of documents about a topic, but rather to bring them together so they can be reviewed" (p. 87). The basic tools in microanalysis are questioning and comparisons (Strauss & Corbin, 1998). Questions must be asked to bring attention to specific aspects of the data, and to hypothesize relationships between concepts. Labels are constantly compared with others with regard to their properties, dimensions, and variations. These constant comparisons between labels help to build relationships between concepts, and serve as the foundation for grounded theory (Dey, 1999). Strauss and Glaser's (1998) coding technique involves three types of coding: open, axial, and selective.

## *Open Coding*

The goal of open coding is to uncover all the concepts and their properties within the data through systematic labeling of each data fragment. In essence, the massive amount of data collected in a study can be broken down into smaller parts that can be analyzed and managed more effectively (Dey, 1999; Richardson, 2005; Strauss & Corbin, 1998). Through open coding, the different labels are categorized and different dimensions and properties are also identified. As the labels start to relate to each other, broader categories of behavior emerge. Each category represents a phenomenon found within the data. Strauss and Glaser (1998) also suggest the use of memos, in which the researcher writes out his or her thought process concerning how the labels were selected and the comparisons made between the time of their selection and identification. In the present study, such memos and notes were included as part of the analysis.

Axial coding further explores the categories found through open coding by relating their properties and dimensions. Axial coding is also described as putting the data back together after open coding has taken it apart (Dey, 1999; Strauss & Corbin, 1998). In axial coding, "categories are related to their subcategories to form more precise and complete explanations about phenomena" (Strauss & Glaser, 1998, p. 124). During this process, relationships are discovered between categories; in the present study, this was accomplished by grouping all the action items in the behavioral protocol logs by common codes. Only major codes that were apparent in at least 75% of the participants underwent this level of coding. The Findings section will guide the reader through the axial coding analysis.

"Selective coding," as Strauss and Glaser (1998) noted, "is the process of integrating and refining categories" (p. 143), whereby categories are joined together to form a general theory for a phenomenon. Strauss and Glaser recommended finding a core category through which all the categories can be connected as a first step toward integration. It will become necessary for the researcher to use explanations and propositional statements to integrate concepts that exist under specific conditions or variations. Dey (1999) noted the problem of knowing when to stop coding. The marker for stopping the coding process is is the point at which a sense of theoretical saturation is reached whereby further analysis results in no new categories, properties, dimensions, or relationships (Strauss & Corbin, 1998). In the present study, the resulting "theory" generated a model for how MCTs can facilitate conceptual understanding. Therefore, this level of analysis is discussed in the next chapter when attempting to answer the research question.

## FINDINGS

After categories had been fine-tuned and integrated in the selective coding stage, a model of how CSNüb affects conceptual understanding of OOP was constructed. This model shows the relationship(s) within the final set of categories, which are exploration, scaffolding, awareness, refinement, and disequilibrium. These categories can be thought of as cognitive processes or factors that influence cognition when using CSNüb. Before proceeding to the overall model found, the following sections discuss how each category was formed within the analysis.

Since these processes are interrelated and overlap in their properties, they can be pieced together to form an overall cognitive model, as is expected in the axial coding stage of a grounded theory analysis (Strauss & Corbin, 1998). From selective coding, specific dimensions and properties were discovered and constructed for each cognitive process and factor. The following sections detail each category of processes that were found in the behavioral protocol logs. Eventually, in the next chapter, these categories of cognitive processes will be connected together and their relationships will be explicitly established.

## Definitions of Categories

Since part of the analysis requires constant comparison between categories, a cursory set of definitions for each category provides the context needed for such a discussion. Other categories found in the open coding process that were later merged or integrated within each larger category are included below as subcategories:

1) *Exploration* is the process of seeking out other resources within CSNüb and its accompanying documentation. Other resources do not include those that participants were specifically told to use. Exploration was part of the planning/design and refinement stage. *Sub-categories: lookup, inheritance, object-oriented programming, encapsulation, hierarchy, problem-solving, planning, design*
2) *Disequilibrium* is a moment in which the participants encountered an unexpected or incorrect behavior from their games. *Sub-categories: feedback, redundant/superfluous code, _rotation*
3) *Awareness* is the scope and range of what participants were able to see with respect to the object-oriented design of the CSNüb system. This can be also thought of as the level of understanding of object-oriented systems. *Sub-*

*categories: object-oriented programming, hierarchy, higher-order thinking, visibility*

4) *Scaffolding* includes resources that supported conceptual understanding and learning, and might affect how a participant designed or implemented his solution. *Sub-categories: modeling, prior knowledge*
5) *Refinement* is the process in which a participant was attempting to improve his conceptual understanding through his solution. *Related categories: debugging, efficiency, consistency, experimentation, testing, consistency.*

## Primary Traits Assessment Scores

For contextual purposes, it is useful to have an overview of how participants completed the tasks before delving into each category. Table 4.1 shows, in minutes, the minimum, the maximum, and the average times to completion for each task and for all tasks combined.

| **Task** | **Minimum Time to Complete** | **Maximum Time to Complete** | **Average Time to Complete** | **Complete Time Standard Deviation** |
|---|---|---|---|---|
| 1 | 4.5 | 23.75 | 9.156 | 6.745 |
| 2 | 30.5 | 108 | 53.292 | 20.317 |
| 3 | 11 | 30.5 | 23.167 | 5.605 |
| 4 | 11.5 | 78.75 | 31.063 | 18.943 |
| All Tasks | 78.75 | 174.75 | 116.667 | 32.620 |

Table 4.1: Overview of task completion in minutes. (n=12)

Some participants failed to complete the tasks, in which case, the completion time was marked when the participant stopped or when I had the participant move on to the next task. On average, participants took a longer time to complete Task 2 than any of the other tasks. This was the first task in which participants had to implement the interaction between two objects, which required them to understand how the game engine works to facilitate that interaction. It was also contingent on how correctly participants performed Task 1, which asked them to assign values to inherited variables in CSNub_Submarine and CSNub_Squid (performing this task incorrectly results in several problems that will be discussed in further sections.) Task 3 is similar to Task 2, which could account for its taking a shorter amount of time: once the participants were able to solve Task 2, they had an easier time working with similar problem. On average, Task 4 took about 12 minutes longer than Task 3. This task asked participants to implement the interaction between a submarine and a squid. Most issues came down to syntactical and arithmetic issues, which will also be discussed in further sections, and this could possibly account for the longer completion time. Like Tasks 2 and 3, successful completion of Task 4 is also dependent on how correctly Task 1 was performed with respect to inheritance.

The Primary Traits Assessment (PTA) scores in Table 4.2 give an indication of how successful participants were with the activity.

| | **Class Design (3pts)** | **Documentation (3pts)** | **Correctness (3 pts)** | **Re-Use (2 pts)** | **Total PTA Score (11pts)** |
|---|---|---|---|---|---|
| **Minimum** | 1 | 1 | 1 | 1 | 4 |
| **Maximum** | 3 | 1 | 3 | 2 | 9 |
| **Mean** | 2.333 | 1 | 2.083 | 1.333 | 6.750 |
| **Standard Deviation** | 0.888 | 0 | 0.900 | 0.492 | 1.712 |

Table 4.2: Participant PTA scores. (n=12)

All but three participants were able to understand the hierarchy and extend classes correctly by the end of the interview, and thus received a score of one point. The average class design score was 2.333, which implies that participants' classes were generally correct with respect to design and how they fit in with the CSNüb architecture; in other words, most students were able to extend the correct classes. Examples of incorrect design decisions included extending the wrong classes and not extending any classes at all.

The documentation score was consistently one point. The lowest score for any of these primary traits was one point, but it should be noted that none of the participants chose to document their code with comments. A few participants mentioned that comments were useful and should be recorded, but none did so. Comments were only used to hide parts of the code from the compiler. This could be due to the study environment and the perceived time-limitations that participants experienced.

The correctness score was based on two scenarios: the player wins the game, and the player loses the game. Six participants' games worked correctly, even though some of the implementation and design decisions were incorrect. Four participants received a score of two due to run-time problems — for example, the common problem among these participants that the appropriate feedback did not appear when the player won or lost the game, such as the "game over" screen failing to appear when the player clearly lost.

The re-use score implied that some students were unable take advantage of inherited code, and therefore duplicated variables and methods. As mentioned before, the re-use score had the maximum value of two points. Since the PTA score was applied at the end of all the tasks, it may also be worth noting that many participants started out not re-using code at all, but as they worked with CSNüb more, they realized that there was code they could re-use, and started taking advantage of this. Eight participants had a score of one point by the end of the activity.

## Disequilibrium

Disequilibrium was a category central to the findings because it often evoked or affected the other cognitive processes found in the analysis. In turn, these moments of disequilibrium affected how participants understood the nature of OOP and how they engaged in problem solving. As participants were thrown into states of disequilibrium, they were in a state of cognitive conflict. Usually, disequilibrium was initiated by visual and textual feedback from the game when the output was inconsistent with what the participants expected. This also included compiler error messages, though most of this type of feedback was related to syntax errors, and not necessarily to OOP. Specific moments of disequilibrium can be tied to the other categories of cognitive processes. There were three main moments of disequilibrium experienced by students, and these are described below.

### *The _rotation Problem*

In Task 2, participants were asked to rotate the submarine 180 degrees when it hits the rock. There is a _rotation property in CSNub_Submarine that is inherited from its grandparent class, CSNub_Object. All participants assumed that *_rotation* was a relative value; that is, they believed that setting *_rotation*'s value to 180 would actually turn it 180 degrees. In reality, *_rotation* is a fixed value whereby a MovieClip will face up when it is 0, down when it is 180, left when it is -90, and right when it is 90.

Therefore, participants saw some sort of reaction when the submarine hit the rock, but the submarine usually turned in an unexpected direction.

### *The Redundant Code Problem*

In all tasks, participants were asked to use variables that were not visible in the class definitions, but were inherited from a parent or ancestor class. For example, in Task 1, participants were asked to set the submarine's *hit_points* property to 10. When participants looked at CSNub_Submarine, there was no *hit_points* variable declaration. Some participants then declared their own variable for *hit_points,* which became a local class variable, even though that this property was inherited from CSNub_Character — the parent class of CSNub_Submarine.

This becomes a problem because the game engine is always looking for *hit_points,* specifically, the *hit_points* inherited from CSNub_Character due to dynamic binding. The scope of this new local class variable is limited to CSNub_Submarine. Since it would have no value (because of the new local variables), the game engine will never have the true value. In fact, it will assume that the value of *hit_points* is 0.

### *The Never-ending or Early-ending Game*

In these games, the players must clear the ocean floor of items and dangerous sea life while remaining alive. In other words, they had to intersect with each item while keeping *hit_points* greater than 0. Obstacles such as rocks were excluded from this requirement. When the player was successful, a game over screen appeared with a congratulatory message. If the player failed, a similar game over screen appeared, but with a message of regret. If the solution was incorrect, the Game Over screen appeared too early, never came in, or contained the wrong message. This was generally caused by an arithmetical error, a wrong object interaction, or an incorrect solution design.

### *Other Issues*

A significant factor to using CSNüb was that most participants were new to Flash and game programming. Only two of the participants reported any experience with Flash, and then only at a superficial level. Only one participant reported any experience in game programming using a game building tool. Thus, though using CSNüb did not necessarily create a moment of disequilibrium, it was an entirely new environment with which participants were unfamiliar, and they relied on their prior knowledge of programming, OOP, problem solving, and preliminary experience with CSNüb in the Tutorial Activity. Along with the other moments of disequilibrium mentioned above, this exposure to a new environment often caused or affected the cognitive processes and factors discussed in the next sections.

### *Summary of Disequilibrium*

These cases of disequilibrium were important in affecting changes in participants' understanding of CSNüb's object-oriented (OO) design and brought attention to key concepts such as inheritance and encapsulation. Polymorphism was covered within the tasks — several inherited classes overrode methods that were defined in a common parent class; however, polymorphism was never explicitly stated within the scope of the activity. The data showed that the moments of disequilibrium set off the cognitive processes described in the next sections.

*In CSNüb, therefore, moments of disequilibrium affect conceptual understanding of OOP by bringing attention to potential flaws and inaccuracies in students' thinking*. These moments engage students in resolving their sense of incompleteness and incorrectness by learning more about OOP and the OO system to perfect their game. The next section describes the process of exploration, which is one method by which students attempted to deal with disequilibrium.

## Exploration

Exploration was key to understanding an unfamiliar OO system. Every participant explored CSNüb's code base and documentation. Though there were a few instances of haphazard exploring, the most interesting finding was that participants' exploration was mainly conducted in a systematic, orderly manner. The purpose of exploration was to seek out other resources that would assist participants in their problem solving. Participants must have a "bigger picture" understanding of CSNüb's object-oriented design to complete the activity, and this was accomplished through looking at the hierarchy diagram in the Tutorial Activity and looking through other files and source code.

The concept of exploration first arose when it was noticed that participants relied on many different files to help them with their tasks, and that those files came in an orderly manner. Table 4.3 shows how many times participants looked up other files, source code, or documentation for each task. A lookup was counted when participants were able to demonstrate their awareness of any class file as being connected to a much larger system (through inheritance or object interaction); cases in which participants looked up other files just as a guide to syntax were not counted.

Table 4.3 shows how many times participants looked up other files according to each task. Some files and code were visited more than once during the same task.

| Task | Minimum Number of Lookups | Maximum Number of Lookups | Mean Number of Lookups | Standard Deviation |
|---|---|---|---|---|
| 1 | 0 | 5 | 2.417 | 2.065 |
| 2 | 1 | 20 | 7.833 | 5.997 |
| 3 | 2 | 17 | 7.083 | 4.999 |
| 4 | 0 | 12 | 3.417 | 3.397 |
| All Tasks | 8 | 42 | 21.25 | 10.515 |

Table 4.3: Exploration counts for "lookups" of other files and source code for the object-oriented purposes (n=12).

On average, participants looked up other files and code 21.75 times during all tasks. The number of lookups increased during Task 2 and Task 3, which can be explained by the nature of the tasks. In Task 2, participants had to implement a rock, which required them to write a new class CSNub_Rock. To make this a part of CSNüb, it had to extend the base class CSNüb_Obstacle, which was provided. Most importantly, doing so was the only way that the game engine could recognize that an instance of CSNub_Rock was in the game. Extending CSNub_Obstacle provided some properties and methods that CSNub_Rock needed to work correctly. In order to be aware of this relationship, participants either had to look at the CSNüb class hierarchy in the Tutorial Activity handout or discern the relationship between the two files. They could also open CSNüb_Obstacle and read the comments about what type of objects would inherit from it. Task 3 was similar in that the new file CSNub_EnergyBarrel had to inherit from CSNub_Item — not only were needed properties and methods inherited from CSNub_Item, the energy barrel also needed the functionality provided to it from CSNub_Object, which is CSNub_Item's parent class.

Another explanation for the increase of lookups during Task 2 is that this is the first task in which participants were asked to implement an interaction between two objects: the submarine and a rock. The role of the game engine is to detect the collision of the two objects and delegate the reaction effects to the submarine. The source code for the game engine was found in CSNub_EventHandler, a MovieClip instance offstage. Six participants (Ethan, Chris, Henry, George, Kyle, and Isaac) actually explored the game engine code. Another three participants (Alex, Brian, and Jared) were only concerned with how it worked, descriptively.

For any given task, participants were explicitly asked to work within a specific set of files.

Task 1 – CSNub_Submarine, CSNub_Squid

Task 2 – CSNub_Rock, CSNub_Submarine

Task 3 – CSNub_EnergyBarrel, CSNub_Submarine

Task 4 - CSNub_Submarine, CSNub_Squid

Though exploring and using other files was not explicitly mentioned, it was necessary to complete the activity correctly. The following paragraphs detail the exploration process for each task.

In Task 1, the hierarchical relationship between CSNub_Submarine and CSNub_Character with CSNub_Character was already coded for the participants, and already recognized by the game engine. This could be why Task 1 had the fewest lookups. This task required participants to assign values to *hit_points*, *attack_points,* and *defense_points* to CSNub_Submarine and CSNub_Squid. Participants should have noticed that these two classes extended CSNub_Character, since that was where those properties were declared. Six participants (Alex, Brian, Daniel, Isaac, Louis, Jared) started looking in other files for the point variables before coding.

On average, Task 4 had approximately half the number of lookups as Tasks 2 and 3 did. This may be because by the time they came to Task 4, participants already knew about the hierarchical relationship between the classes and were familiar with what the base classes provide. Task 4 asked participants to implement the fight sequence between the submarine and a squid. This required participants to call methods that get and set the various *points* for each character in the fight, which were defined in the parent class CSNub_Character. The loser of the battle needed to turn invisible, and that functionality was in CSNub_Character's parent class, CSNub_Object. Another explanation for why Task 4 had fewer lookups than Tasks 2 and 3 is because some participants ran into the problem of having redundant code — specifically, local variable declarations in Task 1 which override the same properties that should have been inherited from CSNub_Character. Such participants would not have needed to access the setter and getter functions for those variables since they were unaware that they existed.

The problems that led to exploration generally occurred during the debugging stage. In Task 1, half the participants did not explore any files, and this resulted in the redundant code problem. The next section discusses the role of exploration at the design stage of problem solving. An important time for exploration to occur is at the onset of a task — during the design and planning stages and before coding.

### ***Exploration in Design***

Some participants knew that they needed to go to other files at the design stage, when they were planning out their solution before actual coding. This strategy was most prevalent in Task 1, where participants had to set values for the point variables in CSNub_Character and CSNub_Submarine, which were inherited from CSNub_Character. Chris said, "If I'm going to set the following values, I'm going to find the class of it." Jared said, "I gotta go find the information for the submarine and both of the squids." These participants demonstrated their intuitive sense that the CSNub_Submarine and CSNub_Squid classes were actually part of a large interconnected system. They assumed that the point variables might already be defined somewhere within this larger system and inherited by these classes. Henry expressed this idea in Task 2: when he wanted to know how certain variables worked together in positioning the submarine, he knew he had to "go up the chain." By this he meant that he saw in the class header that CSNub_Submarine extends CSNub_Character, and that going up the chain would require him going to look at CSNub_Character for more information. Louis also believed that these variables should be declared somewhere, and noted that CNNub_Squid and CSNub_Submarine extended from CSNub_Character. He looked in CSNub_Character "because it has the submarine and squid in it." Although this sentence was semantically incorrect, it showed that he was aware of the hierarchical relationship between CSNub_Character and the other two files, and he proceeded to that class to look for the point variables.

Isaac felt that the point variables should be defined somewhere and proceeded to open the different files: "I guess the submarine must somehow already have these values set, I was thinking somehow, or maybe tell me to set these values. So, I'm trying to find how to do that." In this case, Isaac's sense of exploring was not methodical: he just opened different files. There was some thoughtfulness in his strategy, however, in that he was aware that he was working within a larger system of classes — he was simply unsure of the relationship between the classes. Alex went straight to the CSNub_Character to look for the point variables as he started on Task 1.

Alex: The variables that I need are already there in the character class.

Interviewer: So how is it that they're already in the previous class?

Alex: Through inheritance.

Although he explored the correct files in a systematic manner, he ended up copying and pasting the variable declarations from CSNub_Character into CSNub_Submarine and CSNub_Squid. Alex understood how classes were related, but was unable to implement his solution using inherited variables.

Brian explored other files to determine the nature of each class — that is, what the class represented and how it related to other classes. In determining whether CSNub_Rock would extend CSNub_Obstacle in Task 2, he said the following:

> I assume there's going to be multiple obstacles in this game and I don't know exactly [what] an obstacle should do yet. So, if I look at the comments and the methods provided in the parent class, I'll know what a rock is supposed to do.

In this case, his decision was based on what he found in CSNub_Obstacle.

Exploration was a necessary process for participants in designing their solution correctly. This led to the discovery of parent class files that provided code and insight on how the CSNüb classes work and relate together. As evident in the redundant code problem, participants who did not explore had either a narrow perspective of the CSNüb architecture, or an incorrect view that was based on personal assumptions. Exploration was important because it helped students gather the resources they needed to help them plan their solution. A couple of participants looked at ways to improve the design of CSNüb. Though they did not always implement these improvments, their planning showed evidence of higher-level thought related to CSNüb's object-oriented design. Some participants were able to move beyond the confines of the immediate files they were asked to use. Participants who engaged in such high-level planning took full advantage of CSNüb's architecture in terms of object design and interaction, and worked within the framework to its fullest level. Participants wanted to know what code was available. This was not always successful, as some participants were unable to find what they needed or looked in the wrong place. This type of planning involved exploring other files and code sources. Sometimes, this planning occurred *post hoc* or as a reflection, after they had already implemented some code.

Ethan wanted to put the reaction code between the submarine and the rock into CSNub_Rock even though CSNüb was set up so that all reactions should go directly into CSNub_Submarine. He knew that the submarine had to interact with the rock and he knew that “this is definitely object-oriented programming because we’re going to be doing stuff in the submarine class, ” but he was uncertain where to place that interaction: “I don’t know if you would put the stuff in the rock class or the submarine class.”

In Task 2, Brian wanted to put the interaction code outside the CSNub_Submarine and CSNub_Rock since he claimed that “[t]he intersect method needs to be universal to any object,” perhaps as a utility class. He was wondering if there should be a higher up intersect() method that “would deal with all of the characters that have when they intersect with the main character.” He was thinking of having an intersect() method for each kind of object the submarine could run into. In this instance, Brian was thinking beyond the confines of CSNüb and was considering changing the nature of the system. Kyle wanted CSNub_EnergyBarrel class to have its own intersect() method to handle the item collection sequence. This would add a level of complexity to the design, since the entire activity was focused on putting the interaction code in CSNub_Submarine.

Louis assumed that the variables he needed to implement in Task 2 were already provided for him.

Interviewer: What do you think you can find in the display panel?

Louis: At this point, I’m trying to look what’s in the display panel. Because at this point I don’t see a display for health. And possibly it would tell me where the health values are set.

Interviewer: Are these set in the submarine class?

> Louis: At this point, they do not appear to be.
>
> Interviewer: What are you going to open now?
>
> Louis: Character
>
> Interviewer: And why?
>
> Louis: Character, it has the… ok… it has the classes submarine and squid in it.
>
> Interviewer: And did you find what you were looking for?
>
> Louis: Partially at this point. I definitely found the variables. So, I assume in the constructor, I set the values in there.

Following this line of thought, Louis assumed that the point variables existed elsewhere. He looked at CSNub_DisplayPanel, as he knew that those point variables would be displayed there eventually. When he could not find any hints from using object interaction as a reference, he explored in a hierarchical fashion.

Following the assumption that these variables were not defined, Brian worked to figure out where *hit_points, attack_points*, and *defense_points* should be defined within the hierarchy.

> Brian: Would it be feasible… is it something that all squid, sub, shark has. I guess it would make more sense to put this in the… to put these particular variables in the CSNub_Character class.
>
> Interviewer: And why that one? Why the CSNub_Character?
>
> Brian: Because the sub, the squid, and I'm assuming other things I'm going to create on down the line all have… are going to have HP, defense, and armor. Even if it doesn't attack, set the attack to zero or what not.

At this point he did not know that those variables were already declared. He was planning where to put these properties in a way that would make most sense to the hierarchical relationship between certain classes.

### *Outcomes of Exploring Code*

Three general outcomes of exploration related directly to object-oriented programming and general programming. First, participants explored the code base and documentation to gain a better understanding of an object-oriented system, as related to CSNüb's design. As previously mentioned, this helped solidify participants' assumptions how about classes were related within CSNüb. When participants were unsure of the relationships between classes, they explored other resources to help them determine their relationships. George knew that he wanted to extend CSNub_Rock from a class, but was not sure from which class it should extend. He looked at CSNub_Character to see what similar traits a rock and character have. After this comparison, George found that CSNub_Rock should not extend CSNub_Character unless "you want it to blow up." By this, he meant that a character has the potential to be destroyed or have some other action applied or executed on it, whereas a rock will always just be there. Ultimately, and also incorrectly, he decided that CSNub_Rock had more in common with CSNub_Object and extended this class accordingly. Kyle also struggled to decide what class CSNub_EnergyBarrel should extend to:

> Kyle: I don't know whether to extend it or not.
>
> Interviewer: Why do you choose to extend it at all?
>
> Kyle: I just assume that maybe the energy barrel does something.
>
> Interviewer: Do you know what that class was?
>
> Kyle: I have to look through. I needed to know how I got the submarine's hit_points.

In making his decision, Kyle felt that CSNub_EnergyBarrel should extend CSNub_Submarine because the former needs to have access to the submarine's *hit_points*. In this incorrect implementation, Kyle confused inheritance with object interaction and extended CSNub_EnergyBarrel from CSNub_Submarine because he "needed to know how [he] got the submarine's hit_points," since an energy barrel increased the value of *hit_points*. Nevertheless, exploring other files gave George and Kyle their own ideas of how the object-oriented architecture of CSNüb was organized.

The second outcome of exploration was that other files provided participants with existing code for modeling and structural purposes. This began with a discovery process in which participants found code that they felt would be beneficial for them. Exploring other files and source code helped some participants find out where their code should go when implementing their solution. As mentioned, some participants found it useful to look at the game engine source code. The game engine's source code is essentially a loop that keeps calling characters' *move* methods, listening for keyboard input, and whether or not objects intersect.

This particular code fragment of the game engine proved useful for some participants.

```
// check to see if the hitpoints is at 0
if( main_character.getHitPoints() <= 0 )
      _root.endGame( false );
```

Chris was trying to figure out why his game did not end correctly:

> It's going to check every single time what HP is, is less than zero it goes to the game over screen. It's not doing it. Game over. Do I have to force it to be game over?

Upon seeing the code, Chris detected the lack of connection between the game engine and his submarine. Similarly, Henry found what he needed in the game engine source code to help him design his code such that the game engine determines that the game is over:

> Yeah. I think it's not ending because it caused the game… oh. Wait. Wait. So, I guess the game takes in a Boolean. It's true if the mission — if you won.

> It's false if you lost. Oh, okay. Never mind. Oh, so there's no getHitPoints method. So, I need to implement method too.

Henry realized that he had to write his own *getHitPoints()* method since this was an important part of the role-playing game. The game's end was dependent on the value of the hit points. In this case, the game engine source code let him know what his design was missing. Like Chris, George found that *getHitPoints()* was important since he did not write this method, and was probably why his game was not ending correctly — there was no game over screen when the submarine's *hit_points* went below zero. Seeing this code fragment got him to think about how *hit_points* should have been implemented:

> George: It's perhaps… part of the CSNub_Character class.
>
> Interviewer: You think it's part of it?
>
> George: Yeah
>
> Interviewer: Why is that?
>
> George: It's the parent class. If everything has *getHitPoints()* with it, which would make sense, especially since you… the way it's written. The submarine is not unique to being the main character.

Another facet of exploring for existing code is to use it as models, is when it is applied to syntax. Although not related to object-oriented programming, it is interesting to mention this particular use of CSNüb's existing code. For instance, Brian asked, "Do you particularly mind what syntax we use to create it? Instead of using 'CSNub_'?" In this case, he was asking about coding style and conforming his code to match the coding style of the CSNüb source code. Similarly, as he was typing the header for CSNub_Rock, Frank looked at CSNub_Obstacle and said, "I'm looking at the syntax of the class." Isaac reiterated the same strategy: "I'm just going to look at the classes to see how the syntax works." This also helped him notice that his CSNub_Rock class should extend something.

> Isaac: So now that I have character classes and they're extending… it makes sense.
>
> Interviewer: What are they extending?
>
> Isaac: The character class, right? The squid extends… it makes sense that I have to extend the obstacle class.

Since he noticed that CSNub_Squid was extending CSNub_Character, it made sense for him that CSNub_Rock extends CSNub_Obstacle.

The third outcome of exploration was that it helped participants resolve problems they had with their games; that is, exploration was not limited to the design stage of solving the tasks. Some participants learned to explore other files when they encountered a problem while going through the CSNüb activity. For those who explored in the design stage, some now had the assumption that code might already be written for them. In Task 2, Chris looked up CSNub_Obstacle and said he found help information there, such as the methods for setting and getting the points: "Instead of me writing out the method, it already has it…. Okay, it has points. I can get it and set it." He came to this conclusion because he was aware that CSNub_Rock can inherit from CSNub_Obstacle, although he was uncertain if this applied to both variables and methods:

> It's going to inherit variables — I'm not sure about the variables. It inherits all the methods pertaining to it and it makes it the same objects or basically same properties, just different methods.

Upon discovering needed variables in CSNub_Character in Task 2, Frank learned to look in other files for the subsequent tasks. He referred to this particular moment as a "learning experience," since taking advantage of the inherited properties would "have been a more efficient solution" than what he had done during Task 1, in which he did not explore other files. Ethan looked in other files also: since the squid on the screen was affecting the submarine when it should not have in Task 2, he looked in CSNub_Squid. Frank had a logic issue in the intersect code and went looking around at other files to see where it was because he was using code inherited from CSNub_Character. Jared's first file to explore was CSNub_Character: "It's a place for me to start. It's what I referred to most." He made this reference when he needed to find code that would make his squid invisible in Task 4.

***Summary of Exploration***

Exploration was systematic and methodical; the order in which participants explored generally followed the hierarchical relationships of the object-oriented architecture of CSNüb. More often than not, participants explored other resources because they assumed that they were working within a much larger system than the immediate files. This also included the assumptions that the classes within the system were related and worked together, and thus that some code was already provided for them. Though exploring mainly occurred while participants were designing their solutions, it also happened when participants ran into problems with their games. They began exploring when objects did not interact appropriately with each other and when the states of the objects where incorrect.

*With CSNüb, exploration affects conceptual understanding of OOP by revealing the composition, relationships, and states of objects.* Exploring other files helped ground any assumptions that the participants may have had. For those who were not aware of the larger system, exploration helped them discover relationships between files, the nature of the interaction between objects, and pre-existing code that could be inherited. It was through exploration that participants gained a wider and deeper perspective of the OO system and how OOP worked. Exploration also helped them to discover resources available to them and the limitations of the code. The next section discusses factors involving participants' awareness of the OO system.

## Awareness

When participants were exploring other files, they were sometimes limited in what they were able to see in these other resources. Awareness can also be described as the range of visibility of the OO system within which the participants were working. The scope of awareness guides the exploration process and limits the effectiveness of the cognitive scaffold that CSNüb can provide. Awareness issues can be classified as a spectrum from low/narrow to high/wide. Though some participants exhibited signs of low awareness, this did not imply that they had low awareness throughout all tasks; rather, awareness varied by participant and by task, and often changed during the activity.

### *Low/Narrow Awareness*

Low/narrow awareness was noted when participants were unaware of the class hierarchy and that properties and methods can be inherited from parent classes. Low awareness led to issues such as the redundant code problem and extending the wrong class. Another descriptor of low awareness is narrow awareness — that is, participants had a narrow perspective of the entire CSNüb system; their vision of CSNüb focused only on the files they were asked to look at, not the larger system. This can be thought of as a form of "tunnel vision" within OOP. Table 4.4 is a chart showing how participants exhibited low awareness.

| Participant | Evidence of Low Awareness | First Encountered | When Resolved |
|---|---|---|---|
| Alex | Redundant variables* | N/A | N/A |
| Brian | Redundant variables and methods | Task 4 | Never |
| Chris | Redundant variables | Task 1 | Task 4 |
| Daniel | N/A | N/A | N/A |
| Ethan | Redundant variables | Task 1 | Task 4 |
| Frank | Redundant variables | Task 1 | Task 2 |
| George | Extends incorrect class | Task 2 | N/A |
| Henry | Redundant variables and methods | Task 1 | Never |
| Isaac | N/A | N/A | N/A |
| Jared | Redundant variables* | N/A | N/A |
| Kyle | Redundant variables and methods | Task 1 | Never |
| Louis | N/A | N/A | N/A |

* - Alex and Jared had redundant variables, but they were not caused by low awareness.

Table 4.4: This table shows the participants who exhibited low awareness, which led mainly to redundant code problems.

Below is an excerpt from CSNub_Character showing the point variables (*hit_points, attack_points,* and *defense_points*). Since CSNub_Submarine extends CSNub_Character, the point variables do not need to be re-declared; rather, the participants can just start using the variables.

```
class CSNub_Character extends CSNub_Object
{
      // characters has the following properties

      // the number of hit points is the amount of damage you can take
to remain alive.
      private var hit_points:Number;
      // the number of attack points is the amount of damage you can
inflict on another character
      private var attack_points:Number;
      // the number of defense points is the amount of damage you can
absorb without losing
      // hit points when attacked by another character
      private var defense_points:Number;
```

Here is an example of redundant code from Henry:

```
class CSNub_Submarine extends CSNub_Character
{
```

```
private var HP:Number;
private var AP:Number;
private var DP:Number;

// constructor
function CSNub_Submarine()
{
      HP = 10;
      AP = 3;
      DP = 2;
}
```

The redundant part of Henry's code is that the point variables were newly declared and did not make use of the point variables inherited from CSNub_Character. Other cases of redundant code were similar to Henry's solution, with the exception of different identifiers.

Redundant code was the main indicator of low or narrow awareness: 7 of the participants declared variables that were inherited from a parent class; 3 of the 6 participants with initial low awareness were able to resolve this issue. Participants were largely unaware of the problems with redundant code. Redundant code did not become a problem until participants started dealing with the "game over" sequence related to losing the game: when the submarine's *hit_points* went to 0 or below, a game over sequence was triggered. Without using the inherited variables, the game engine never knew correctly when the game over screen should appear. This tended to occur in Task 2 when the participants implemented the rock interaction, which was the first object that (negatively) modified the *hit_points* of the submarine. It was in Task 1, however, that redundant code was mainly initiated in those participants who had this issue.

In Task 1, Ethan did not find the point variables, which "should be at the very top" of CSNub_Submarine, nor did he notice that this class extends CSNub_Character. This resulted in the declaration of redundant point variables. It was interesting to see that even though Ethan declared his own point variables in CSNub_Submarine, he still called its parent class's *setHitPoints()* methods. This showed an inconsistency between what Ethan believed and what was actually inherited.

```
var HP: Number = 20;
…
function intersect( other:Object )
{
        …
        setHitPoints( HP )
        …
}
```

*HP* was the variable that he declared to keep track of the submarine's hit points, but he also updated the *hit_points* variable (inherited from CSNub_Character) which the game engine checks, so even though the design was inconsistent, the game appeared to function correctly. Ethan's awareness increased in Task 4 when he explored CSNub_Character and noticed that the point variables and setter and getter functions for these values were available.

> Ethan: Oh wow!
>
> Interviewer: What?
>
> Ethan: *getStatus()* and everything. So I could just use… oh my gosh, I could just use these things that you already created.
>
> Interviewer: How come you didn't use them before?
>
> Ethan: I didn't realize they existed. I didn't notice that you had them in a different class
>
> Interviewer: Did you see the extends earlier?
>
> Ethan: I did, but not — I just glanced… I didn't really think about it. Oh my gosh!

He was surprised to find the inherited code. This was an "a-ha" moment, which brought attention to the fact that CSNub_Submarine extends CSNub_Character and that code is inherited and available for reuse. In Task 3, Ethan also noticed that

CSNub_EnergyBarrel extended CSNub_Item inherited *points* variables because he looked in CSNub_Item. He applied what he learned from Task 4 to Task 3 (Ethan completed Task 4 before Task 3).

In Task 1, Chris said "it should be in the constructor, but there are no instance variables. Do I create those variables?" Since the variables were not there, he was thinking about declaring his own variables. He believed that the constructor was only used for initializing variables and that the point variables should at least be mentioned there. Henry also created his own point variables in CSNub_Submarine. He was really confident about his design — almost to the point of thinking he was starting from scratch: "I'll probably have to define them myself… and then initialize the constructor." He even contemplated writing his own setter and getter functions for these values.

Ethan, Chris, and Henry had a similar thought process: look for variables where they are normally declared (before any constructor or method) or defined (in the constructor). Since there was no mention of point variables in either place, their first instinct was to declare their own variables and they failed to consider that these variables might have been declared elsewhere, as perhaps hinted in the class header.

## *High Awareness*

High awareness is the ability to see any object within the context of the larger system with respect to how objects relate to each other and interact. As shown in Table 4.4, Alex, Daniel, Isaac, Louis, and Jared exhibited high awareness by looking in parent class files for inherited properties and methods. Brian also exhibited high awareness, except in creating an extra *character_name* variable and in the supporting methods he declared in CSNub_Character, which already existed in CSNub_Object. George had the insight that the classes CSNub_Rock and CSNub_EnergyBarrel should be integrated into the CSNüb system, but he did not inherit from the appropriate classes. These participants had an adequately high sense of awareness with respect to inheritance and class relationships. This awareness, however, was generally due to prior knowledge, as it was part of their initial problem-solving strategies.

Object interaction was one factor in CSNüb that was not as apparent as inheritance, and to which prior knowledge might not have been as easily applicable. Having an understanding of how objects interact adds another layer of complexity to the relationship between classes. The game engine was the code that drives the game and facilitated the interaction between objects. When an object on the screen collides with another, it calls the *intersect()* method in both objects. Since the class files were easily accessible through their folder, I had to show participants how to view the source code for the game engine. Participants were given a brief descriptive overview of what the game engine does, but some felt the necessity to see the actual source code, which furthered their understanding of how object interaction was facilitated through CSNüb.

Alex thought it was "pretty cool" that the game engine detects the intersection between objects. Chris felt that it was necessary to know how objects collide with each other, asking questions such as "So, when it intersects something, it's going to pass what it intersects, right, which is an *other*?" and "When it intersects... how do I make it do that?" Henry found that this code helped him design his code so that the game engine determines that the game is over: "Oh, so there's no *getHitPoints* method. So I need to implement that method, too." He recognized that CSNub_Submarine must contain a *getHitPoints()* method, but was unable to recognize that the method was already inherited from CSNub_Character and available for use. Though Henry exhibited a low awareness of inheritance, his desire to understand object interaction within CSNüb gave evidence of a high awareness of object interaction. George wanted a bigger picture of how the game engine works to detect collisions so that he would know how to implement the *hit_points* variable in Task 2: "If I knew how it was going to use the HP object… how the game is actually going to determine how HP is implemented, it may be useful." Like Henry, George felt that understanding how the game engine worked with the submarine object was an important concept in determining the flow of the game.

When Kyle was trying to figure out why the Game Over screen was not appearing, he realized that the game engine code was not connected with his own code after seeing the game engine code. He noted, "I need to make the method *getHitPoints()*," which could be a precursor to figuring out that this method was already inherited and available for use. Kyle even asked, "Can I keep this up?" since he wanted to continue referring to the game engine source code. Thus, Henry and Kyle wrote their own methods to retrieve the new variables they declared — the most important method being *getHitPoints()* in CSNub_Submarine, even though it was inherited from CSNub_Character.

Isaac found the game engine code important to understanding the flow of the game and how the Game Over sequence was initiated. In debugging Task 2, Isaac said, "We kept running into the rock, we went into negative number, right? And hp is 0, it should get the game over screen." He wanted to resolve the issue of *hit_points* going into negative numbers when the Game Over screen should appear when *hit_points* is equal to 0. To try to figure it out, he said, "So, I'd like to go back to that… where you actually create the game over screen."

Louis was able to discern a relationship between CSNub_Submarine and CSNub_Squid's parent class — not necessarily by looking at the class header, but by how the classes were related. First, he looked at CSNub_Submarine "to see what information the submarine has." He assumed that the "beginning health values would be in there." He looked and did not find them in the class file and looked in CSNub_Character "because it has the submarine and squid in it." He interpreted the relationship as one of CSNub_Character subsuming CSNub_Submarine and CSNub_Squid, instead of a hierarchical relationship.

Several participants showed evidence of high awareness in that they had a good understanding of the OO architecture of CSNüb, but ran into problems with implementing their designs. Alex saw that the point variables were defined in the parent class, but declared his own redundant code in Task 1 by copying and pasting the variable declarations from CSNub_Character to CSNub_Submarine and CSNub_Squid:

> Okay, with the submarine file I implemented the variables for hit points, attack points, and defense points. Within the constructor, I set the hit points equal to 10, the attack points equal to 3, and defense points equal to 2. In the squid class, I also instantiated the hit points, attack points, and defense points. I set the hit points equal to 5, attack points equal to 3, and defense points equal to 1.

By "implemented", he meant he that he declared variables within this class, even though he knew that they were inherited. Alex displayed a further disconnection between knowing about inheritance and actually using it by calling *this* — which "makes sure that it calls this submarine — the class; it's syntax to call the class" — using *this* to ensure that the variables *hit_points, attack_points,* and *defense_points* in CSNub_Submarine were called without regard to the same variables that are inherited.

It should be noted that redundant code was not always a sign of low awareness. For example, Alex and Jared had redundant variables in CSNub_Submarine despite exploring CSNub_Character. Both participants copied and pasted the instance variables from the parent class into the inherited class. This was an example of participants' exhibiting signs of high or wide awareness, but without the adequate ability to implement their understanding in code. These examples were not included in Table 4.4.

Jared and Alex knew that CSNub_Submarine extends CSNub_Character and "inherits" the point variables in Task 1, yet both participants copied and pasted variable declarations from CSNub_Character into CSNub_Submarine. This may be due to the participants' wanting CSNub_Submarine to have those variables explicitly. Jared knew the variables needed to be "carr[ied] over" and to "reset the variables," and he copied them over as a way to "mimic" the CSNub_Character class. Alex knew that CSNub_Character had the variables that were to be inherited by CSNub_Submarine — "The variables that I need are already there in the character class" — but he also ended up copying and pasting the variables. In these instances, it was possible for students to know what it means to inherit variables from a parent class and yet be unable to correctly implement inheritance. At this point, Alex and Jared were unaware that they had implemented their knowledge of inheritance correctly due to the lack of feedback. Jared did not test his game. Alex tested his game and no compiler errors occurred, so he assumed that the game worked; the problem was that he was unable to test any type of behavior that made use of these point variables, as it was not until Task 2 that participants could see how point variables were factored into the game play (when the submarine ran into a rock).

For Task 2, Frank knew to go to the parent class CSNub_Obstacle to help him with his implementation of CSNub_Rock, whereas he neglected to do the same in Task 1 with CSNub_Character and CSNub_Submarine. Chris noticed that CSNub_Item extends CSNub_Object, but did not notice that CSNub_Submarine and CSNub_Squid extends from CSNub_Character in Task 1. There was some inconsistency among the participants as to when and whether they decided to explore other resources or not, which was an indicator of the participants' range in awareness. Despite Frank's being aware of the hierarchy, he also copied and pasted *setPoints()* and *getPoints()* from

CSNub_Obstacle into CSNub_Rock, even though he explicitly extended CSNub_Rock from CSNub_Obstacle "so this class can have that method, I guess." Frank, however, also used *super* in these method calls, which was redundant since those methods are already inherited. Below is Frank's source code for CSNub_Rock that illustrates the disconnection between knowing about inheritance and not being able to implement it:

```
class CSNub_Rock extends CSNub_Obstacle
{

	function CSNub_Rock()
	{
	}

	function setPoints( p:Number ):Void
	{
		super.setPoints();
	}

	function getPoints():Number
	{
		return super.getPoints();
	}

}
```

He explained that "it's called *super* so it can access its public methods and that's private methods, but that's what the *getPoints* is."

Though not directly related to object-oriented programming, it was interesting to note that some participants' awareness issues stemmed from not reading instructions or the contents of the files. In Task 3, George explored CSNub_Object looking for the methods that dealt with awareness: *setVisible()* and *getVisible()*. Although they were there, he just did not see them, which was possibly due to the length of the file. In Task 1, Isaac was trying to find a class where he thought the point variables were defined, "trying to find where these characters' properties are defined… like where… and how do we access that. And it seems like this has a mention of it. Where can I find HP?" He believed that they were in CSNub_Item because there was a mention of points (the *points* variable). He did not see that CSNub_Character was the mediating piece between CSNub_Submarine and CSNub_Squid. His awareness with respect to OOP was high enough to understand that the code was probably defined elsewhere, but he was unable to see the line:

```
        class CSNub_Submarine extends CSNub_Character
```

Kyle saw that CSNub_Submarine extended CSNub_Character, but failed to look at the class. This may be due to limited awareness of the task, since he believed it was so simple that it would not require any inheritance.

One affordance of inheritance is that it helps with code reuse. High awareness gave participants the foresight to seek out code that might be available for use. In Task 1, Alex knew to go directly to the parent class to look for the point variables:

> Alex: I think I'm going to make the class constant… class instance variable of the HP. I think… [*Alex opens CSNub_Character*]. The variables that I need are already there in the character class.
>
> Interviewer: So, how is it that they're already in the previous class?
>
> Alex: Through inheritance.

Nevertheless, he ended up copying and pasting the variable declarations, which led to the redundant code problem.

Without assuming that the point variables were already created, Brian felt it was best to declare in CSNub_Character "something that all the squid, sub, sharks, etc., has. I guess it would make more sense to put these particular variables in the CSNub_Character class." Exploring the CSNub_Character bolstered his beliefs about where the variables should be declared and which classes have access to them: "Ah, you've already created them." Daniel was able to see the relationship between CSNub_Submarine and CSNub_Character because he "couldn't find anything about HP and stuff, so I just thought, but since those are characters — submarine, squid — maybe those components are here." Following that assumption, he looked in CSNub_Character for the point variables. Kyle used *._x* and *._y*, but assumed without looking that they were declared elsewhere: "I guess they were made in another class… the super class, the character." This could be explained by the existing code in CSNub_Submarine, which utilized these properties. Regardless, Kyle exhibited some higher-level conception of what properties a submarine might have, as well as any hierarchical relationships that CSNub_Submarine might have with another class. Such relationships, however, were not explained.

### *Transitioning from Low to High Awareness*

Like those with low awareness, Frank knew that the "submarine actually ha[d] those values," yet he did consider that other characters (e.g., the squid) would require those values. Eventually in Task 2, Frank noticed that CSNub_Submarine extended CSNub_Character, and decided to look in CSNub_Character. He was laughing because the *hit_points, attack_points*, and *defense_points* are similar to the variables that he created for CSNub_Submarine and CSNub_Squid: "It seems just what I did. [laughs] I could have been using that for a more efficient solution." He proceeded to change his code in CSNub_Submarine by removing the variable declarations for the points and using the setter methods inherited from CSNub_Character to set the values for the class. This was repeated in CSNub_Squid, offering an example of how awareness was changed through using CSNüb.

In Task 2, Frank had a bigger picture understanding of CSNüb's object-oriented system through the game objects:

> Looking at the graphics, it would seem weird to have made — well, the obstacles are all doing kind of one purpose, and it would have been to weird to have the

rock as its own superclass — so, that the obstacles are doing something and have different characters. It seemed like it should have been a subclass.

Frank was negotiating where in the CSNüb hierarchy CSNub_Rock should have gone: should it be a class disconnected from the system, or a class that extends an existing class? His design strategy depended on what he believed were the behaviors of rock versus an obstacle, and on discerning the relationship between the two. In Task 3, the first thing he did was to figure out which class CSNub_EnergyBarrel should extend by looking at the CSNüb hierarchy in the Tutorial Activity. After finding CSNüb_Item, he looked in there to see what code was inherited: "I guess I need to see what methods it does so that energy barrel can do what it's supposed to."

Interviewer: So you're looking at the hierarchy?

Frank: Yes.

Interviewer: And why are you looking there?

Frank: At the Item class to see which one [he opens CSNub_Item.as].

This is a further illustration of how his previous experience in Task 1 informed subsequent tasks.

Ethan also had a learning moment when he explored the game engine code in Task 4. He realized that when he was using HP, a redundant variable he declared in CSNub_Submarine, was never accessed by the game engine: "It is something I created and not in the parent class." Ethan completed the Task 3 after Task 4, and like Frank, Ethan's strategy changed in the next task. When writing the class for CSNub_EnergyBarrel, he determined from the hierarchy diagram in the Tutorial Activity that the parent class was CSNub_Item and looked in the file for any needed code. He had learned from the previous task to determine the parent class and look in it prior to coding the class.

In Task 2, Chris realized that the game engine code was actually calling *getHitPoints()*, a method in CSNub_Submarine's parent class "because the game over screen is getting the hitpoints of its parents. But the parent doesn't know the hitpoints of the child because the parent's hitpoints is a 100." Not only did he believe that "it would have saved me in typing", he also believed that he would not have been able to complete the task without seeing the game engine code.

***Summary of Awareness***

Awareness is the scope of what participants were able to see with respect to the object-oriented system of CSNüb. Participants with high/wide awareness took advantage of CSNüb's object-oriented nature to figure out how to implement their solutions. This awareness of the larger system provided participants with more scaffolding, since they understood how the classes were related, what properties and behaviors were inherited, and how objects interacted. These three issues were key to solving the tasks correctly. In contrast, those with low/narrow awareness saw only the immediate files they were asked to work with and did not immediately see the rest of the system, and thus were unable to take advantage the provided scaffolds.

For participants with high awareness, CSNüb was a tool helping them apply and work with their understanding of OOP. High awareness, however, did not always imply that all participants were able to execute their solution's design. Some participants understood the object-oriented nature of CSNüb and had a conceptual understanding of object interaction and inheritance, but were unable to implement it in code. Redundant code was a problem that resulted from low awareness and caused the game to have incorrect behaviors. Through the debugging process, participants were able to discover inherited code and how objects interact, which, in turn, widened the range of their awareness.

*CSNüb affects students' conceptual understanding of OOP by helping to transform their level of awareness of an object-oriented system.* It is their awareness that determines how extensive and how deep their understanding of an OO system is. Exploration yielded the discovery of resources that widened students' awareness. Such resources served to scaffold student understanding of OOP. The next section describes the different forms of scaffolding that CSNüb provides to increase awareness and understanding.

**Scaffolding**

CSNüb was a contained environment, in that participants had only the template, the code, and documentation with which to work. Through exploring other files, participants were able to develop a better understanding of the object-oriented nature of CSNüb's architecture and this assisted in organizing their problem-solving strategies. These discovered resources were useful as conceptual models for CSNüb's object-oriented system, as well as for syntax. For example, source code in the class files told participants how the files related to each other and helped them build a model of related classes within an OO system. Moreover, participants gained additional scaffolds via the visual and textual feedback from the games they created as well as cues from the Flash interface.

***Modeling an OO System***

Participants gained a better understanding of how classes were related to other through inheritance by looking at the code, the diagram of CSNüb's architecture, the Library folder structure, and the game engine source code. Doing so led to the exploration of different files and the discovery of additional code that served as cognitive scaffolding. Significantly, this exploration was carried out in a systematic manner as it pertained to inherited relationships. Participants were able to determine the relationship between the class files and then followed the chain of hierarchy and interactions to discover needed information.

Participants explored parent files in order to figure out what properties and methods were inherited. After examining CSNub_Squid class in more detail, Ethan noticed that it extended CSNub_Character and he explored that file. Alex went to the CSNub_Character file as his initial strategy in Task 1. When exploring CSNub_Item, Chris saw that it extended CSNub_Object and explored further by looking in CSNub_Object. Louis looked at CSNub_Character, CSNub_Object, and CSNub_Obstacle to see what he had available to him in CSNub_Rock for Task 2. Many participants relied on the *extends* part of class header to see which class files to explore, though they did so at different times within each task.

Not all hierarchical relationships were figured out through the class header. The hierarchy diagram provided in the Tutorial Activity (see Figure 4.1) showed the hierarchical relationship between classes, and Ethan and Alex said they were able to figure out the hierarchical relationships through this diagram; in doing so, they were learning how to integrate CSNub_Rock into the hierarchy. When asked why he was looking at the Tutorial Activity's class hierarchy diagram, Alex said he was looking to see "[i]f there was something that [CSNub_Rock] should extend or not," and thus, he found CSNub_Obstacle.

```
* CSNub_Object
        |-----> * CSNub_Character
                        |-----> CSNub_Submarine
                        |-----> CSNub_Squid
                        |-----> CSNub_Eel
                        |-----> CSNub_Ogrefish
                        |-----> CSNub_Shark
        |-----> * CSNub_Item
                        |-----> CSNub_EnergyBarrel
                        |-----> CSNub_RadioactiveBarrel
        |-----> * CSNub_Obstacle
                        |-----> CSNub_Rock
                        |-----> CSNub_Kelp
        |-----> * CSNub_GameObject
                        |-----> CSNub_DisplayPanel
                        |-----> CSNub_EventHander
CSNub_ObjectRegistry
```

Figure 4.1: The CSNüb architecture as shown in the Tutorial Activity.

A few participants used the Library's folder structure as a way to determine whether a new class they created (e.g., CSNub_Rock, CSNub_EnergyBarrel) should extend something. This was an unanticipated behavior since the Library's folder structure was simply means of organizing graphics so that users could easily locate the required symbols to drag them onto the Stage (see Illustration 4.1). Daniel and Louis extended CSNub_EnergyBarrel from CSNub_Item because the energy barrel symbol was in the *item*s folder in the Library:

Interviewer: What are you extending it from?

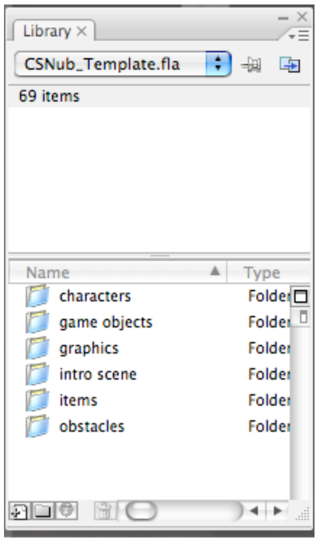
Daniel: From the item object.

Interviewer: And why did you choose the item object?

Daniel: Because it was in the item library.

Louis knew that CSNub_EnergyBarrel was "going to inherit from the item class" because he saw how "so far, the things in the library is divided by the [pointing to the folders]." Henry knew to extend CSNub_Rock from CSNub_Obstacle because the rock symbol was found in the *obstacles* folder in the Library:

> I saw that there was a CS Nub obstacle class, and I'm going to go ahead and guess that's it… Yeah. I guess another obstacle would be — I don't know — I'd say look at the file and find another obstacle and see if [incoherent] help as an obstacle.



| Name | Type |
|---|---|
| characters | Folder |
| game objects | Folder |
| graphics | Folder |
| intro scene | Folder |
| items | Folder |
| obstacles | Folder |

Illustration 4.1: Library window from the Adobe Flash environment.

The game engine code scaffolded the participants' understanding of how objects interacted in CSNüb. When participants were unaware of how the game engine detected a game over situation, they found the reasoning in the event handler code. The game engine was always checking to see if *getHitPoints()* was greater or less than zero. This code fragment was like a requirement (or a guideline) to which participants had to adapt their solutions.

Upon discovering this code fragment, Henry and Kyle felt that they needed to have *getHitPoints()* because they did not write it themselves. Henry and Kyle wrote their own *getHitPoints()* method, but that resulted in the redundant code problem. Though there was a flaw in class relationships, pertaining to inheritance, the source code guided their understanding of how objects interacted within CSNüb. Brian said that the game engine code told him he needed to write a *getHitPoints()* method. George found it useful to view the game engine code to see how it detected object collisions. Kyle even kept the code up for reference in Task 2 and asked "Could I keep this up?" The code also informed him that he needed a *getHitPoints()* method.

***Modeling Code***

Not only did CSNüb provide models of an OO system, it also modeled code for participants. Previously written code in CSNüb gave participants support in terms of designing the logic of their code and programming syntax. Both these issues were best illustrated in Task 2, in which many participants experienced the *_rotation* problem. Participants had to deal with a property that involved the submarine's heading. The solution to the *_rotation* problem was generally solved by using existing code as a model of how to use *_rotation*. Some assumed that *_rotation* was a relative value, that is, they believed that setting *_rotation* to 180 would turn the submarine around 180 degrees.

The CSNüb code base also served as a model for ActionScript 2.0 — a language with which the participants had never used before. Additionally, participants from the CS1 course were unfamiliar with *switch* statements and had to learn it as they went along. Table 4.5 shows how scaffolding was used in the case of the *_rotation* problem in Task 2.

This table deals only with previous code used for modeling the state of the submarine object.

| Participant | Set *_rotation* = 180? | Scaffolding Received | Resolved *_rotation* problem? |
|---|---|---|---|
| Alex | No | *move()* | N/A |
| Brian | No | CSNub_Object, *keyboardInput()* | N/A |
| Chris | Yes | *move(), keyboardInput()* | Yes |
| Daniel | Yes | *keyboardInput()* | Yes |
| Ethan | Yes | None | No |
| Frank | Yes | *move(), keyboardInput()* | No |
| George | No | *keyboardInput()* | Yes |
| Henry | Yes | *move()* | Yes |
| Isaac | No | *keyboardInput()* | N/A |
| Jared | Yes | *move()* | Yes |
| Kyle | Yes | *move(), keyboardInput()* | Yes |
| Louis | Yes | *move()* | Yes |

Table 4.5: Scaffolding participants received for Task 2 and the *_rotation* problem

At the outset, 7 of the participants initial solution to turning the submarine around after hitting the rock was to set *_rotation = 180*, whereas 4 participants understood from the beginning that *_rotation* was a fixed value. Whether participants experienced the *_rotation* problem or not, they were still scaffolded by the existing code. Either it helped participants to avoid the *_rotation* problem, or it helped them to resolve the *_rotation* problem. The most beneficial existing code , which provided a model for how *_rotation* worked, was from the *move()* and *keyboardInput*() methods. *move()* changed the direction based on the value of *_rotation*. *keyboardInput()* changed the submarine's *_rotation* based which of the arrow keys the player pressed. Below is the code for these two functions.

```
function move():Void
{
        switch( this._rotation )
        {
                case 0:
                        this._y = this._y - 5;
                        break;
                case 180:
                        this._y = this._y + 5;
                        break;
                case 90:
                        this._x = this._x + 5;
                        break;
                case -90:
                        this._x = this._x - 5;
                        break;
                default:
                        break;
        }
}

function keyboardInput( keycode:Number ):Void
{
        switch( keycode )
        {
                case Key.UP:
                        this._rotation = 0;
                        break;
                case Key.DOWN:
                        this._rotation = 180;
                        break;
                case Key.RIGHT:
                        this._rotation = 90;
                        break;
                case Key.LEFT:
                        this._rotation = -90;
                        break;
                default:
                        break;
        }
}
```

When dealing with the submarine's *_x*, *_y*, and *_rotation* properties, Frank scrolled up to look at the *move()* and *keyboardInput()* methods, which used these variables. Some of the other participants took a more pragmatic approach and copied the code from *move()* and pasted into the *intersect()* method to handle the turn around of the submarine. Daniel said, "I thought if I just put move method, then it would automatically get the rotation and keep the code that way."

Though Daniel copied and pasted the code, he only used one of the *_rotation* statements and set it to 180. Jared copied and pasted the *move*() code into *intersect()* even though he thought it was "repetitious," which meant that he was aware that he was re-using coding. Even so, he cut and pasted all the cases within the *switch* statement except for the third. He was using the existing code to help him understand how to rotate the submarine and how to use a case statement (he had not used switch statements before).

Similarly, Kyle saw that the *move()* method processes the submarine's *_rotation* property and used that as a base. He said that *move()* had "the same cases as what I have." The cases referred to the different values *_rotation* could assume. In CSNüb, *_rotation* only had four possible values: 0 (up), 180 (down), -90 (left), and 90 (right). He asked, "Will I need to, like, go a certain way when it turns?" which revealed his understanding that turning the submarine around might depend on its *_rotation* value. He concluded that this behavior was similar to *move()* and copied and pasted the code into *intersect()* for Task 2. For the displacement portion of Task 2, Chris looked at the *move()* method to figure out what direction (the submarine is facing) matched with each value of *_rotation*: "Wait, there's a special case for every one? For every direction it's hitting. So if it's hitting it from up, it should face down and go down." Alex realized this at the beginning of Task 2: "I'm trying to figure out if I can just add 180 to the rotation? Or so do I something with a switch right here because it doesn't go… like, I guess it goes from 180 to -180." With this line of thought, he figured out that there was a range of values for *_rotation* from observing the previously written code.

Brian and George were the only participants who went to where *_rotation* was originally declared, in CSNub_Object, to see if they could find more information about it. George explored CSNub_Object's *_rotation* "for consistency," which implied that he wanted a model of how to use *_rotation*. He also remembered from the previous activity that *_rotation* was not relative and even tried to figure out a more effective way to rotate the submarine without having to use a *switch* statement. Isaac noticed that *_rotation* required some logic and attempted to negotiate his understanding by looking at the previously written code:

> It will make the rotation number first. So, it seems to be the way they did it here, this._rotation = 0, maybe that way you could just flip it. So, with this logic here [*keyboardInput()* and *move()* methods]. Does this make any sense? *this._rotation* […] No matter where it's coming from, you flip it 180 degrees. So, if it's zero, it'll go to 180. If it's 180, it's going to go back.

He knew that if the *_rotation* value was 180, it had to go to 0 and vice versa. The existing code gave him insight on how to design his solution using the *_rotation* property.

Though this aspect is not specific to CSNüb's MCT design, it is interesting to discuss the role of source code as a model. Reuse, as a property of OOP, served as cognitive relief for some participants, in that they did not have to do everything from scratch. Only Chris and Jared had any experience with Flash — their current CS course was in Java. Though both ActionScript and Java are very similar and conform to ECMA standards, there are still some syntactical differences.

Chris thought that *getStatus()* was "cool" because he did not have to write this function, which would be needed in Task 4. He also expected that there would be an already existing method or code to make a character disappear, which did exist as *setVisible()*. As a result of looking in CSNub_Character in Task 2, he thought that what he found in CSNub_Character would have been helpful: "It would have saved me in typing." It is also notable that some participants reused code purely for syntactical purposes. Chris copied and pasted the class code from CSNub_Squid because he doesn't "know how to create a class from scratch." Jared also copied the CSNub_Rock class definition into CSNub_EnergyBarrel, and made the changes to the class header and constructor.

## *Gameplay Feedback*

Even if participants had entered their code correctly and compiled it without errors, there was no guarantee that the game would work. In fact, the moments of disequilibrium discussed here did not occur at the code level but rather occurred when participants were playing their games. Running and playing their games were the only ways participants could know whether they had implemented the features correctly. Thus, the visual and textual feedback from interacting with the game through provided cues indicating correct or incorrect behaviors.

The *display_panel* was a source for scaffolding because of the output. When the submarine encountered an object, the interaction was detailed in the *display_panel*. Participants relied on the visual output messages in Task 4 to see whether the fight sequence code was correct and this was used mainly for debugging purposes. The reliance on the output messages occurred mostly when the expected interaction did not occur visually.

In Task 2, Ethan saw that the submarine passed over the rock without any reaction, whereas colliding with the squid evoked a message. He realized that he needed to incorporate CSNub_Rock into the hierarchy: "I don't see why wouldn't it. It inherits from Character. Ahhh… it doesn't extend. Okay. Extends Obstacle…" Isaac and Louis's submarine also ignored the rock, and in this case, they forgot to give the rock symbol and instance name as well as to add it into the *object_registry*.

## *Summary of Scaffolding*

*CSNüb provides resources such as documentation, previous code base, and multimedia feedback to scaffold students' conceptual understanding of OOP.* Existing code in CSNüb served as cognitive scaffolding, which assisted participants in constructing a more complete understanding of the object-oriented design. Students became aware of such code when they explored and their awareness increased. Through the *_rotation* problem in Task 2, participants were guided by existing code to help formulate their solution and to better understand the state of the submarine with respect to its *_rotation* heading. Participants used the existing code as foundational knowledge that they incorporated into their own knowledge; thus, even when participants were copying-and-pasting, they still needed to adapt the code to their own use. This also resulted in an emergent use of CSNüb as a model for syntax and logic — when participants were unfamiliar with AS 2.0 syntax (e.g., variable declarations), CSNüb-specific logic or new programming constructs (e.g., switch/case). Finally, playing the game provided participants with multimedia-based feedback and cues that illustrated objects' behavior and interaction. Though their code might have been compiled without errors, the actual game play may have uncovered many bugs. Participants were able to test their theories regarding the state and behaviors of their objects and how objects interact through playing their game. Such feedback through experimentation helped participants refine their conceptual understanding of the objects in the game.

## Refinement

Software is never perfect. Thus, an essential point of software development is to make sure that the product is as correct as it can be, usually through extensive testing and debugging. Naturally, all participants in this study had gone through the debugging process while attempting to complete the tasks. However, debugging was just one part of a bigger process, which is referred to as refinement. Refinement has a larger meaning and encapsulates the debugging process as well as making code more efficient. These two are related, in that both processes are trying to make the code better.

Refinement began when the participants tested their movie, which occurred when participants compiled their code. After which, the Operation SPLASH game is loaded. If the code compiled without errors, the game ran and the participant was able to play the game. If the participant had made errors, the game either did not run or it would appear, but not function correctly. In either case, testing the movie was the beginning of the refinement process.

Table 4.6 below shows an excerpt from the behavioral protocol log, which can illustrate the refinement process for Henry:

| Action # | Task # | Description |
|---|---|---|
| 30 | 2 | He uses Math.abs to change the rotation |
| 31 | 2 | TESTED THE MOVIE - He sees that the submarine keeps flipping back and forth. |
| 32 | 2 | TESTED THE MOVIE - He runs the movie again and sees the same thing. |
| 33 | 2 | TESTED THE MOVIE - He sees the flipflop problem again. |
| 34 | 2 | He continues to tweak his intersect method in terms of ordering rotation and displacement code. |
| 35 | 2 | He copies and pastes the rotation method's body into the intersect method. |

Table 4.6: Refinement process example – Participant 7, Task2

Henry was testing his game and using multimedia-based feedback to help refine his understanding of the nature of the *_rotation* property. By Action #30, he tried to assign different values and mathematical functions to *_rotation*, but saw no change. Then, he started to experiment with the ordering of his code in Action #34. In Action #35, he used code he found in the *intersect()* method in CSNub_Submarine that involved the *_rotation* property. In this example, Henry used multimedia-based feedback, through playing his game, to find clues on the nature of *_rotation*. There was a hint of experimentation when he was refining his solution through assigning absolute values to *_rotation* (calling *Math.abs()*) and changing the order of his code. Another strategy was to reuse code: he used existing code as cognitive models for how *_rotation* should behave and be implemented.

To further illustrate the concept of refinement, Table 4.7 shows how Frank attempted to resolve a problem in Task 4 when he received no feedback:

| Action # | Task # | Description |
|---|---|---|
| 106 | 4 | TESTED THE MOVIE – Nothing happens when he hits the pink squid. However, when he pilots it away from the squid, the game over screen comes on. |
| 107 | 4 | He is tweaking the formulae. |
| 108 | 4 | TESTED THE MOVIE – He is moving the submarine between the two squid and nothing happens. |
| 109 | 4 | He goes back to looking at the intersect() code. |
| 110 | 4 | LOOKUP CSNub_Character. |
| 111 | 4 | TESTED THE MOVIE. Nothing happens when he continues to run into the pink squid. |

Table 4.7: Refinement process example – Participant 4, Task 4

Frank tested his game, but nothing happened when the submarine hit the squid. This lack of feedback, which is actually feedback in itself, directed him towards a problem with his code. He experimented by tweaking the part of his code that calculated the amount of damage each character can inflict on each other. Part of the refinement process was to gather more information through exploration, and Frank turned to exploring CSNub_Character in Action #110. For both Henry and Frank, refinement began with playing the game, receiving incorrect or unexpected feedback, implementing a strategy, and then playing the game again. The strategy included exploring, experimenting/tweaking the code, and applying new resources found.

Testing the game was the primary method for participants knowing whether their game worked correctly or not. Table 4.8 shows the number of times participants tested their game.

| Task | Minimum Number of Tests | Maximum Number of Tests | Mean Number of Tests | Standard Deviation |
|---|---|---|---|---|
| 1 | 0 | 1 | 0.167 | 0.389 |
| 2 | 3 | 19 | 10.167 | 4.745 |
| 3 | 3 | 10 | 5.167 | 2.25 |
| 4 | 0 | 22 | 9.333 | 7.679 |
| All Tasks | 16 | 42 | 25.25 | 8.572 |

Table 4.8: Counts of how many times participants tested their games. (n=12)

On average, participants tested their games 25.25 times throughout the interview. Participants tested more often in Task 2 and Task 4 than the others. In comparison to these two tasks, participants tested a little more than half the times for Task 3. Task 2 was the first task in which participants were asked to create a class from scratch and have it work within the CSNüb architecture, as well as have its objects interact with others.

Task 3 was similar to Task 2, and could be an explanation of why there are fewer tests. This can be shown by Table 4.9, which displays time spent per task. On average, participants took 53.292 minutes to complete Task 2, but only took 23.167 minutes to complete Task 3. In terms of learning goals, the main difference was that Task 3 asked participants to use a method inherited from a grandparent class, whereas Task 2 limited the scope of inheritance to the parent class. Since they had just completed Task 2, the participants were familiar enough by that time to do Task 3 without much more testing.

| Task | Minimum Time Spent | Maximum Time Spent | Mean Time Spent | Standard Deviation |
|---|---|---|---|---|
| 1 | 4.5 | 23.75 | 9.146 | 6.744 |
| 2 | 30.5 | 108 | 53.292 | 20.317 |
| 3 | 11 | 30.5 | 23.167 | 5.605 |
| 4 | 11.5 | 78.75 | 31.063 | 18.943 |
| All Tasks | 78.75 | 174.75 | 116.667 | 32.620 |

Table 4.9: Time spent on each task in minutes. Some participants were unable to complete the task. (n=12)

On average, Task 4 took participants 31.063 minutes — the second longest task. The purpose of Task 4 was for participants to get used to the idea of encapsulation and data hiding. The goal was to make use of setter and getter functions between CSNub_Submarine and CSNub_Squid as a way for these two objects to interact within a fight sequence. Participants were given formulae to implement in ActionScript. The explanation for this much time spent was mainly a syntactical problem; participants were unable to correctly implement the formulae, not necessarily because of syntax errors, but because of logic and arithmetic errors.

One explanation of the low number of tests in Task 1 is its simplicity: the task was to set the values of some variables in CSNub_Submarine and CSNub_Squid. Since no visual output was expected, some participants elected not to test the code at all. Only Isaac and Louis tested the game in Task 1 for in order to receive feedback. Isaac said, “I’m going to test the game to see if I can make the game end,” which implied that he wanted to interact with the game and see the effects of his code. For this task, participants mainly tested their games for to check the correctness of syntax for their variable declarations or for the correct value assignment for inherited variables. Since such tests were only for syntax and other compile-time errors, they were not included in the counts for Table 4.8.

Two aspects of refinement were found in the data: 1) *initiation* of the debugging/testing process, which was evoked by the visual/textual feedback from the game, as well as moments of cognitive disequilibrium that might result from such feedback; and 2) the actual *execution* of the debugging process, which involves exploration and experimentation. In CSNüb, efficiency pertains to making the code take better advantage of its object-oriented design. Students believed efficiency meant maximizing re-use of code and inheritance. The difference between debugging and efficient coding lies in the assumption under which the coder is working. Debugging assumes that the code will probably not work perfectly, whereas making code more efficient assumes that the code works correctly, but is not optimal in terms of performance. Another issue affecting refinement is how consistent participants were when designing, implementing, and testing their solutions.

### *Starting the Debugging Process*

The debugging process started when participants were provided with feedback that was contrary to what they expected. Generally, feedback came from the real-time behavior of the game, which participants saw through the visual and textual cues while playing the game. Daniel checked for syntax errors in Task 3, but chose not to run the game. When asked how he knew that his code was correct, however, he did run the game:

Interviewer: Is there a way for you to know for certain if that worked or not?

Daniel: Yeah, you can just test it out and see if it works.

In Task 2, participants had to rotate the submarine 180 degrees away from the rock and displace it by about 50 pixels in the new direction. Alex made sure the displacement of the rock was correct by hitting each side of the rock with the submarine, and reading the textual output from the *display_panel*, which told him the state of the submarine after each collision with the rock. Brian saw that his submarine was flipping the wrong way and realized that he had displaced the submarine in the wrong direction: "In this case, I displaced in the wrong direction." The visual feedback informed him that the displacement was incorrect; he was able to see the bug, and figure out how objects were positioned on the screen.

When confronted with the same problem, Kyle thought that "it's intersecting too much." That is, *intersect()* was being called repeatedly when it should only be called once. Feedback directed Kyle to the place in his code where he believed the problem lay. In Task 2, his submarine turned to face right when he hit the rock from the bottom instead of flipping to face down as it should. Since this problem arose from interaction, he concluded that the code he wrote in the *intersect()* method was incorrect.

Kyle: Something needs to be changed.

Interviewer: What needs to be changed?

Kyle: When it turns 180, it's still trying to come back up.

He started looking at the code in *keyboardInput()* and *move()* since he noticed that 180 is a case in the switch statement. He proceeded to add a similar switch statement to his *intersect()* method.

In Task 4, George noted that the submarine kept losing in its battle with the squid. When the submarine was at full points, it was not mathematically possible for it to lose to the squid. Based on the textual output of each object's state, he was able to determine where his errors were and fix them appropriately. For example, he noted that "the squid should attack one more time" upon reading each character's attack.

The Game Over screen was an important cue for participants in that it was the end-point for the game play. If a player cleared the ocean of dangerous sea life and items, the Game Over screen congratulated him. If the player died, the Game Over screen announced a mission failure. Participants played their games expecting one of these outcomes, but the Game Over screen did not always appear.

Chris, Henry, and Kyle deliberately ran their submarine into the rock to see if they could trigger the Game Over screen. For Chris and for Henry, the Game Over screen did not appear, which started the debugging process for each of them. Chris again had this issue in Task 4: he was not getting any output when the submarine and the squid intersected each other, and so he decided to print out information to the Output window using a *trace* statement. "The only way to test that, is to put this trace method right in here," he said. "That should tell me what I'm doing wrong." This was akin to creating output that is only visible to the developer within Flash and would never been seen in the final product of the game; it showed a necessity for information that could assist in debugging and ensure that he got some more informative feedback.

Though Ethan was able to get the Game Over screen in Task 4, there were some errors with the textual output from the fight sequence, and he was able to use these for debugging purposes. The submarine's *hit_points* went to -1 a few times, which Ethan explained this way:

> Let's see.... Oh! This needs to be... if AP is greater than other AP — no... It seems that while statement would control the… Oh! — because we didn't actually end the game. Like all we have — oh, no, the game should end. I'm not sure why it's doing that.

He used the game output and tried to match it with the equations he implemented in the sequence to make sense of what was happening in the game. He was also determining how his *while* loop affected the values in terms of when the fight sequence should continuing iterating. The non-re-activeness of the visual feedback was not enough to let him know how he could proceed in debugging.

### *Debugging*

Experimentation was one method of performing the debugging process. Participants were aware of the unexpected behavior of their games, yet they were unsure where the problems originated. In such cases, they experimented with hypotheses they had as to why their games functioned incorrectly. They did this by exploring, tweaking code, or just testing the movie repeatedly to see if they could fix the problem by chance. Jared admitted that he was "usually a trial and error type of guy" and he adopted a "compile it and see if it works" strategy when asked if his solution to Task 1 worked, which relates to a dependency on output and feedback from the game. Another quote that represents the essence of experimentation using CSNüb comes from Chris, who, when testing his code for Task 2, said, "Okay, I'm just going to test out the hypothesis real quick. I'm just going to die."

In Task 2, Chris said he was "just going to test out the hypothesis really quick" and proceeded to hit the rock several times to kill the submarine on purpose. He was testing to see how the game should end in preparation for the potential destruction of the submarine. His hypothesis was that the game engine was not detecting that the submarine's *hit_points* had fallen below 0. Through his hypothesis testing, Chris saw that the *hit_points* were going into the negative numbers. He asked, "Was I supposed to write something that makes it die?" — even though he knew that the game engine would detect this situation. Another hypothesis he generated was that the game engine did not automatically check for game over, and that he might have to write the code himself: when looking for code in CSNub_Character de said, "There's nothing to die."

Similarly, when Game Over was not triggered, Henry believed that the *intersect()* method was not being called: "Maybe the intersection is not being called? Is there is a check for the hitpoints? I wonder why it's related to… not work for the sides at all because it was still getting caught in a loop." Here Henry was intentionally damaging the submarine to try to cause the Game Over. Likewise, when Louis was trying to figure out the true nature of the *_rotation* property in Task 2 he originally coded the turn-around part as `_rotation = 180`, but began to think that *_rotation* was a "heading" and was absolute: "So, I define like how the logic is behind… like no matter where it's coming from, flipping it 180 degrees. So if it's 0, it's going to go 180. If it's 180, it's going to go back vertical," he said, pointing upward. He tried to test out this theory, even though he only tested one case, that of hitting the rock from the right side.

Experimenting was one method by which some participants attempted to solve the tasks without any bigger or full picture thinking. Instead, they experimented using CSNüb with a solution knowing that it might not be correct and that they were just testing to see if it would work. When the rock interaction was not working in Task 2, for instance, Ethan struggled to guess what could possibly go into CSNub_Rock.

Interviewer: What do you think should go into a rock class with the information you have here?

Ethan: Maybe the rock's position, where it is.

Interviewer: What else?

Ethan: The current HP?

Interviewer: Of who?

Ethan:  Well no, that would be in submarine.

Originally, Ethan did not write a constructor for CSNub_Rock, but eventually he put one in there when he saw that nothing happened when the submarine ran into the rock. Incidentally, omitting a constructor from a class definition yields no errors in Flash. Ethan came to the conclusion that CSNub_Rock just has get and set methods — possibly due to his belief that there were no real behaviors for the rock, only behaviors to get/set the rock's position.

Likewise, Kyle's original CSNub_Rock class had no constructor and only a method for decrementing *hit_points*. With respect to the rock object, he said, "I guess it just sits there." Isaac thought there was nothing wrong with having an empty class since "[y]ou're just making a class for rock," and thus he removed the constructor all together. One plausible interpretation was that since the rock did not actually do anything, no other code is really needed for CSNub_Rock. In these cases, participants were experimenting with ideas about how a class should be designed, and were trying to make sense of what the CSNub_Rock class actually did. As Brian observed,

> The constructor for obstacle doesn't initialize anything, so the rock does not need to initialize anything except its own properties, which would be the hitpoints. […] And since it appears that there's only one particular type of rock, every rock placed on the board will take away one HP and refers to itself when it hits it. It's the general parameters for every object created of that type [...] and rock doesn't do anything else.

The participants, thus, were negotiating the purpose of a constructor for a class that represents a rock, which has no real state — only behaviors. Therefore, they experimented with their basic understanding of class design. The debugging process also gave illustrated how consistent students were with their thinking, problem solving, and implementation of their solutions.

### ***Consistency***

Consistency refers to a similar pattern of thought when dealing with design and coding, and it affected how participants debugged and optimized their code. Consistency can be categorized as writing correct or incorrect code, but only incorrect or inconsistent coding will be discussed here. Observations of consistency were drawn from the final source code that participants created in the activity.

Some participants were consistent with incorrect coding behavior. Ethan was consistent in his redundant point variables across all classes he had to create or modify. Alex and Jared copied and pasted redundant variable declarations across classes. Isaac consistently failed to define constructors in CSNub_EnergyBarrel and CSNub_Rock and left the class definitions empty, even without a constructor. These various forms of incorrect coding may be due to the fact that variables were accessed directly in existing code. These cases showed a consistent train of thought, which may indicate a lack of scaffolding or lack of adequate guidance from CSNüb when students were writing the wrong code.

Several participants were inconsistent with coding throughout various tasks. Consistency was an issue mainly when participants were dealing with encapsulation. Some of this could be explained as participants' having a poor or incomplete sense of coding style — that is, their code still compiled and was syntactically correct, but failed to take advantage of encapsulation.

Ethan used *this* when dealing with *_rotation*, but never used it anywhere else. Ethan declared his own point variables across classes, but used inherited methods such as *setHitPoints()*, which could have been due to his perception of the time constraints leading him to feel that he had no time to refine his code. Additionally, he said that constructors were good for "initializing instance variables," but he did not do so in any of the constructors — he did not even have a constructor in the CSNub_Rock class.

Alex repeatedly called modifier methods, despite making one statement in which he referenced *this.hit_points* directly, though this may have been merely a minor oversight. Brian only used *this* in Task 4 when calling methods of the same name from two different objects, which indicated that Brian needed additional scaffolding to help him organize two similar objects. Frank mainly used *super* to call methods defined in the parent class, but used *this* in Task 3. In some parts of Task 4, he returned to using *super*. Chris was inconsistent with using *this*, using it in the *intersect()* code, but nowhere else.

Daniel knew to inherit and use the *points* property from CSNub_EnergyBarrel, but did not do so with CSNub_Rock; he may have assumed that any obstacle will always inflict 1 HP of damage to the submarine. Since the value of damage is constant and always 1, there is no need to have a variable.

Henry redeclared variables in CSNub_Squid, but not in CSNub_Rock or CSNub_EnergyBarrel; he wrote all setters/getters for CSNub_Squid, but only wrote *getHitPoints()* for CSNub_Submarine. He did not use *this* in the constructor initially for CSNub_Squid and CSNub_Submarine, nor did he for other class constructors. George redeclared point variables in CSNub_EnergyBarrel, but did not repeat this error for other classes; he also extended CSNub_EnergyBarrel and CSNub_Rock from CSNub_Object, which was not the intended hierarchy. Isaac's variable declarations were in the constructor for CSNub_Squid, but same variables in CSNub_Submarine were declared outside the constructor and had class scope. Louis did not use *this* in variable initializations for CSNub_Rock and CSNub_EnergyBarrel, nor for CSNub_Squid or CSNub_Submarine.

These observations of inconsistent behaviors and consistent incorrect behaviors were drawn from the final source code; they indicate how CSNüb can be improved with respect to OOP in terms of guiding cognition and problem solving. Participants must be consistent in their coding, not only for coding style purposes, but to ensure the accuracy of their code (e.g, using *this* as opposed to using *super* or nothing at all). It should be noted that some participants were still unaware of the hierarchical relationships between classes even though there were many resources from which they could have discerned them. Finally, participants needed to conform to the data-hiding premise of encapsulation.

### *Summary of Refinement*

Refinement is a process of making the game more correct or efficient. It involves the sub-processes of debugging to make the code syntactically correct so it will behave in an expected manner.

*CSNüb facilitates students' conceptual understanding of OOP by supporting the refinement process in which students receive feedback through interaction (e.g., debugging, experimentation, and exploration).* Debugging was facilitated by the visual and textual feedback from the games' output. Participants played with their games to see if their code was correct, which involved experimenting with different use-cases and states of the objects. Experimentation also occurred when participants formed hypotheses about the code based on the visual and textual evidence from the game. An important sub-process that guided the refinement process was their consistency in code design. Similar to awareness, consistency affected the nature of the participants' code in terms of how well it was designed, coding style, and how well it took advantage of OOP with respect to encapsulation and inheritance.

## Summary of Findings

The findings of the present study implied that moments of disequilibrium (evoked by visual and textual feedback that was contrary to what students expected) were often the catalyst for further problem-solving processes involving exploration and refinement. Exploration allowed students to gain a wider and deeper perspective of an object-oriented system and exposed them to potential resources that could be used in designing their solution. Thus, exploration led to a larger degree of awareness and the gaining of more scaffolding. Refinement was the process in which students were making their games better, using the multimedia-based feedback. They realized something was wrong with their programs and re-analyzed their designs in order to identify what was wrong and how they could correct the errors, and they found other resources that could assist them. Similar to exploration, in refinement students dove deeper into their existing code while gaining a fuller understanding of the behaviors of their existing games (through experimentation and playing their games), which also affected their degrees of awareness of the object-oriented system. The feedback students received from refinement, moreover, served to scaffold their understanding of the system overall.

# Chapter 5: Conclusions

## INTRODUCTION

The purpose of this study was to investigate how CSNüb facilitated novice students' conceptual understanding of OOP. The data analysis resulted in the identification of five categories or factors that affected conceptual understanding: exploration, awareness, scaffolding, refinement, and disequilibrium. Each of these cognitive processes and factors is also related to the others. In attempting to answer this study's research question, these relationships have been explicated and investigated further. At this point, the larger issue of how the findings of this study can inform the theoretical design multimedia-based cognitive tools like CSNüb can be addressed.

## Research Question

The research question proposed in Chapter 1 was as follows:

*How does using CSNüb affect the conceptual understanding of OOP for students who are novices to object-oriented programming?*

Implementing the key role-playing game features in CSNüb cannot be achieved without having a bigger-picture understanding of the architecture as well as an in-depth understanding of the code involved with class composition and object interaction. Students needed to know the relationships between the classes, how to integrate new classes into this network, and how objects interact. Students had to look at other files and examine their contents. Exploring had to be self-initiated; that is, the task descriptions only told students the immediate files they needed to use, but they had to realize that a proper solution must take advantage of pre-existing code and the object-oriented design. The only way for students to learn this was for them to explore the resources provided. Otherwise, their awareness of the object-oriented system would be limited to only the files they were asked to use. Further, students were in charge of their own learning and directed their own problem-solving strategies; they used tools and resources they thought were important, searching wherever they wanted, and took an active role in constructing their conceptual understanding of the object-oriented architecture of CSNüb. Due to the laboratory environment of the study, participants only had access to the materials that I provided to them. They were not able to receive additional help from the Internet, from books, or from me; confining them to the resources they had (the CSNüb template and handouts) promoted the exploration process. It was the interaction with CSNüb that facilitated the exploration process, which led to the bigger picture understanding of an object-oriented (OO) system. As students explored more classes within the tool, they became aware of the relationships between objects and use them accordingly to solve their problems. The role of multimedia was to provide feedback for students' implementation of these relationships. In the case of inheritance, properly inherited code would result in correct behavior in the game. The multimedia characters in the game also facilitated students' discovery of the relationships between classes — for example, a hierarchical relationship exists between a character and a shark, or between an item and an energy barrel.

Another cognitive process that involves direct interaction with CSNüb is refinement. One part of refinement is the debugging process — trying to find out what is wrong with the game and trying to fix it. Within this context, refinement does not refer to debugging syntactical errors but rather to debugging runtime errors that occurred when students were testing their games. At this point, a student's game was imperfect — that is, the code syntax was correct, but the design of his solution was incorrect. This may be due to limited awareness of what students could see of CSNüb's object-oriented architecture. Through the debugging process, students were identifying what might be wrong and exploring other files to see what new (and more correct) information could be adapted — both on the code and cognitive levels. Thus, their understanding of OOP was continually refined and reworked to correct old information and adapt new information. Another component of refinement was optimizing code; this is a higher-level process, in that students were trying to improve their designs, even though the designs were already correct and working. Refinement was usually accomplished by taking further advantage of OOP's affordances, improving performance, and adhering to coding style. Drawing from constructionism, refinement of the resulting video game can be seen as a reflecting the refinement of the students' conceptual understanding (Papert, 1991).

CSNüb is comprised of foundational source code, a game template, a graphics library, and documentation. These resources were used to create a game, but also served as cognitive scaffolding for understanding its OO design and OOP in general. As students explored the source code, they began to realize the inherited relationships between classes and to take advantage of inherited code. Looking at the game engine's source code showed students how objects interact with each other and how the game play depended on the objects' states. The source code also served as models for syntax, class design, and object interaction. Students were able to use these examples by adapting them for their use. Moreover, the participants in this study all had familiarity with video games and were familiar with role playing games, characters and their properties, different items that can appear in such games, and so forth. Thus they were able to use their prior knowledge in games to further scaffold their understanding of how classes such as CSNub_Character, CSNub_Obstacle, and CSNub_Item relate to each other and game play. CSNüb, therefore, provided yet another form of scaffolding with respect to the authenticity of a game environment. Using more real-world objects such as the submarine, rock, and energy barrel further grounded the abstractness of the aforementioned classes. A requirement of Flash is that class files must be linked to the symbols: this gave students the ability to see the connection between a class — a piece of code that represents an abstract object — and an actual object (the Symbol) in the game.

As students explored and refined their code, their view of CSNüb's OO system widened and increased. The degree of awareness changed as students worked with the template and delved further into the code. Students became more aware of the OO system and affordances of OOP, and this awareness informed future exploration and the refinement processes.

When students tested their games, they expected that the game would behave correctly, especially if there were no syntax errors. When unexpected and incorrect behaviors occurred, students entered a state of disequilibrium where the feedback did not make sense. As they tried to resolve the errors, they began to correct their code through the refinement process; this was done by testing the game again, so they had a better concept of the incorrect behavior and perhaps received hints on the source of the problem in the code. They could also explore different code and files to look for resources that could assist them through modeling or reuse. Since the processes of exploration and refinement and the factors of scaffolding and awareness were often evoked or affected by this feedback, it is important to note that disequilibrium played a central role in affecting novices' conceptual understanding of object-oriented programming. The data showed that prior knowledge also ignited the exploration process and guided the problem-solving process, but the research question specifically asked about how the MCT itself affected cognition.

In summary, CSNüb affects novice students' conceptual understanding of OOP by allowing students to begin with their own conceptual understanding of an existing object-oriented architecture. When asked to complete tasks of an OOP nature, students devise a plan they believe would work within their conceptual understanding. As they begin to implement their solution, they can see from the multimedia output whether their object-oriented solution is correct or not. To increase the range of what they understand of the OO architecture, students must explore other files, which broaden the conceptual scope of their understanding of OOP through CSNüb. Thus, their visibility of the OO system increases as they explore. Their understandings may also be refined and strengthened as they work more within this environment. Scaffolding, received through exploration, serves to highlight important aspects of students' knowledge. The moments of disequilibrium set off the cognitive processes of exploration and refinement. This is also the point at which students are integrating and adapting new information into their existing knowledge structures. Thus, it is these moments of disequilibrium that bring about changes in students' conceptual understanding of OOP.

## Practice Informing Theory: Constructing a Cognitive Model

Each of the processes and factors discussed in Chapter 4 could be categorized in different levels of interaction with CSNüb ranging from direct use of the tool to resulting cognitive effects after using the tool. As part of a grounded theory analysis, the goal was to construct a generalization or conceptualization of a phenomenon, that is, a model of how novices' conceptual understanding can be facilitated by MCTs. This study has examined how theory (MCT design framework) played out in a practical setting (CSNüb). One of the major implications of this study is that the findings can be used to re-inform the theory.

With disequilibrium as the central category found in the selective coding process, it was possible to anchor the remaining categories around it in a combined model. This model is useful in looking at the different ways MCTs could facilitate conceptual understanding. The model can also help re-inform and update the theories and literature upon which the MCT framework is based. The remainder of this chapter will focus on generalizing each cognitive process and factor found in the data analysis, constructing a theoretical model of how MCTs can facilitate conceptual understanding, and reviewing the MCT design framework.

## GENERALIZATION OF COGNITIVE PROCESSES AND FACTORS

With respect to grounded theory methodology, simple categorization or conceptual ordering is not sufficient (Glaser, 2001; Strauss & Corbin, 1998). Though the present study has offered a detailed discussion and description of each category of behavior, the interrelationships between the categories have not been fully investigated. The categories found in this study — moments of disequilibrium, exploration, scaffolding, awareness, and refinement — are closely interrelated, and a discussion of those relationships adds a greater depth to the findings and furthers the construction of a grounded theory. To add to the conceptual power of the grounded theory, this discussion remains on a higher-order conceptual level rather than being dependent on specifics and minutiae as to the participants, setting, or factual data (Glaser, 2001). In further exploring the ramifications of the MCT framework, through the CSNüb implementation, each category is discussed on a more theoretical and design level while anchoring it into specific aspects of CSNüb.

**Moments of Disequilibrium**

Moments of disequilibrium were the catalyst for many of the cognitive processes and factors found within the data. Figure 5.1 illustrates the concept of disequilibrium with respect to MCTs.

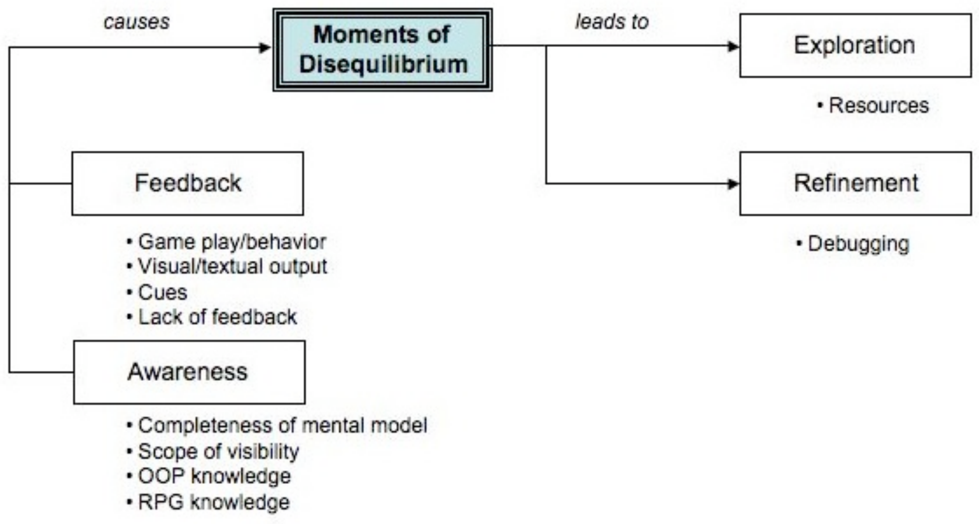


Figure 5.1: Moments of Disequilibrium

Moments of disequilibrium occurred when students' mental models were unable to explain a particular situation; this led to a sense of incorrectness or incompleteness. Students' prior knowledge may not have been complete or sophisticated enough to understand the activity or how to use the MCT. This may stem from students having a narrow sense of awareness of OOP and of how to work within an OO system, since with CSNüb, success was dependent on how well students understood OOP. Students also brought with them an understanding of games, especially role-playing games, but due to the contrast between the complexity of existing games and the simplicity of CSNüb, students had to re-think their knowledge of what a basic role-playing game entails. When students were working with the CSNüb, they were presented with a new way of programming and a new way of thinking. First, they were using a new environment that focuses on multimedia authoring. Though similar applications can be developed in Java or C++, and with even more sophistication, students are used to command line-based programs and applications with simple graphical user interfaces. Using a game template adds yet another level of complexity, since the students were not familiar with this type of programming where it is necessary to truly understand how objects interact, the state of objects, and all the graphical components involved. When moments of disequilibrium occurred, as a solution, students started exploring for any type of resources that might give them examples, hints, or other models to help them with their problem.

A catalyst for disequilibrium was the multimedia feedback from CSNüb during game play. When the visual or textual feedback was unexpected or incorrect, this showed a flaw in a student's design, and thus, in understanding. Students implemented their design with a belief that their code was probably going to be correct, in most cases. They received visual feedback from interacting with CSNüb and seeing how objects interacted and how their states are changed through those interactions. Interacting with CSNüb was necessary when fully testing the system, since object interactions may be dependent on the object state (e.g., the values an object has, the way it is facing, when the interaction occurs, etc.). In this type of interaction, participants could see correct visual output for one case and incorrect visual output for another and all within the same game instance. The textual feedback further informed students on the state of the objects, as that was the main point of the textual output in the activity. When students were confronted with feedback that leads to disequilibrium, they became aware of the problems in their solutions and began to fix them through the debugging process.

### *Disequilibrium and the MCT Design Framework*

Disequilibrium is a state in which students are confused and conflicted. MCTs can bring about such states by providing: 1) multimedia-based feedback that is contrary to the students' beliefs, either coincidentally or incidentally; 2) activities that challenge students at their level of development while taking advantage of their prior experience; and 3) environments that are novel and engaging to students. As MCTs are partners in learning, they must also provide ways of resolving disequilibrium by giving students the appropriate information and feedback needed and adequate scaffolding.

## Exploration

Exploration is the process by which learners search for resources. Figure 5.2 illustrates the cognitive process of exploration.

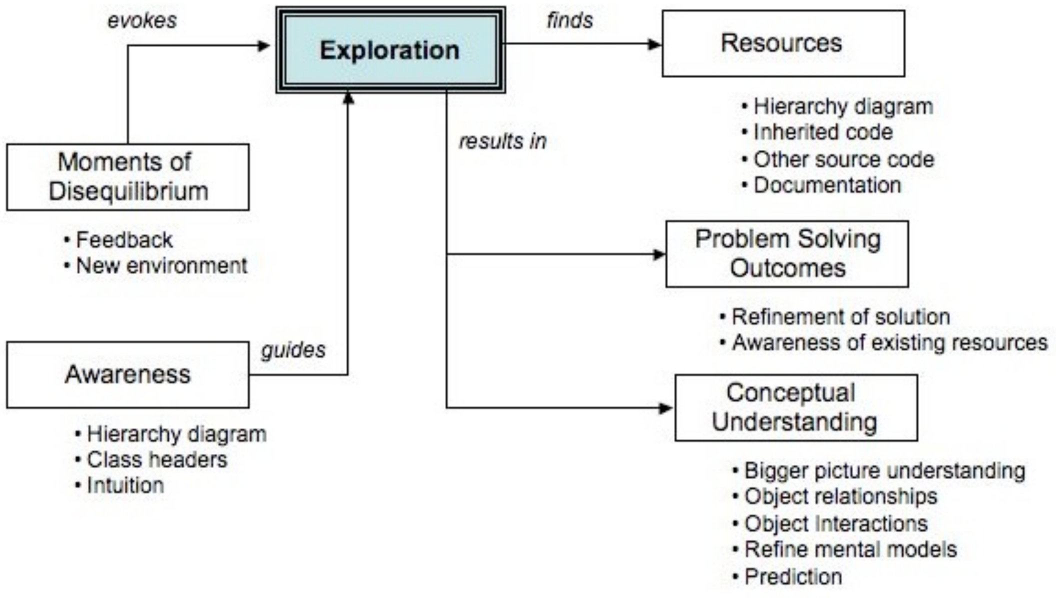


Figure 5.2: Exploration

Exploration was initiated by moments of disequilibrium when students experienced something for which their mental models could not account. Students encountered these moments of disequilibrium when they experienced unexpected or unanticipated behaviors from CSNüb. When faced with such problems, students attempted to refine their solutions, and how to do this was not always obvious. They had to depend on the feedback from CSNüb to guide their debugging process and to figure in what way their solutions were incorrect or incomplete. Exploration and discovery of resources ultimately changed their understanding of the object-oriented system. Another source of disequilibrium was the new environment in which CSNüb placed the students — they were in a new and unfamiliar environment (e.g., development environment, programming language, type of activity, etc.). They were constantly in a state of transition and translation between their prior understanding of OOP and how that might apply within CSNüb.

While exploring, students were able to find resources that helped them understand the object-oriented system and how the different classes related and the objects interacted. It was these relationships that guided exploration; for instance, students who were able to discern a hierarchical relationship between two classes explored those two classes in a hierarchical manner. The ability to discern such relationships depended on the scope of their awareness: for students to explore in a methodical manner, they had to exhibit a wide range of awareness in order to detect such class and object relationships. Students who were unable to see these relationships were also unable to take advantage of OOP's inheritance and reuse features. This failing sometimes resulted in haphazard and random exploration. It was apparent, therefore, that awareness affected how exploration was carried out.

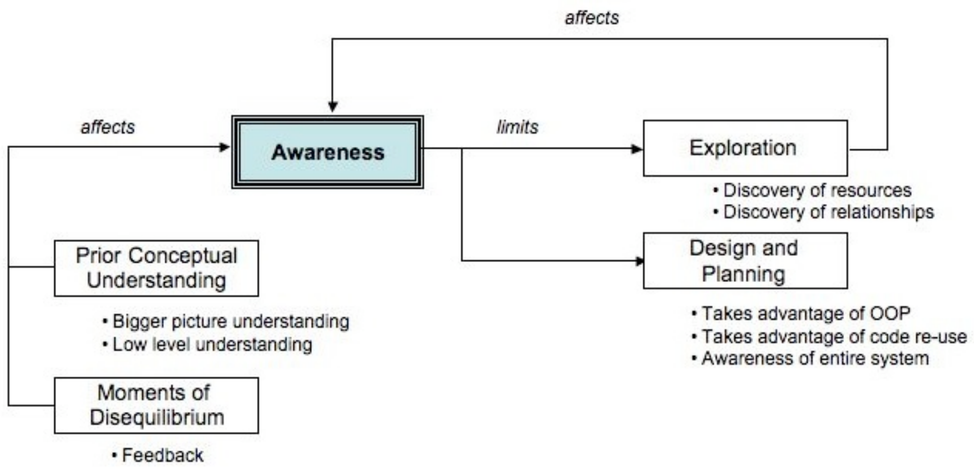
The resources that students discovered guided future problem solving and the refining of previous problem-solving strategies. Such resources were not limited to the CSNüb environment, but included the supporting documentation as well. These resources further informed students on the nature of the classes and the OO system. They began to have a deeper understanding of how classes related to each other, how they interacted with each other, and how the entire system worked together. As they explored each class file, read more documentation, or interacted with their game more, students learned the significance of each class in relation to the entire system and how it might interact with or affect other classes. As this understanding increased and deepened, students' problem solving became more complex, taking in to account the larger system as they developed their solutions.

### *Exploration and the MCT Design Framework*

Exploration was one method by which students tried to resolve disequilibrium. The quality of exploration (methodical versus random) was dependent on students' understanding (awareness) of the OO system. MCTs must provide resources students can access while trying to achieve equilibrium, and the tools and facilities with which they can explore their own understanding. MCTs should guide exploration so that it is methodical and purposeful. Exploration should lead to the discovery of resources (e.g., documentation, feedback, and information) that completes, changes, or challenges students' conceptualizations.

**Awareness**

Awareness refers to the range, in terms of breadth and depth, within which learners can see the object-oriented system. Figure 5.3 illustrates the concept of awareness.



affects
affects
Awareness
limits
Exploration
• Discovery of resources
• Discovery of relationships
Design and Planning
• Takes advantage of OOP
• Takes advantage of code re-use
• Awareness of entire system
Prior Conceptual Understanding
• Bigger picture understanding
• Low level understanding
Moments of Disequilibrium
• Feedback

Figure 5.3: Awareness

When students began working with CSNüb, they started with their prior knowledge. Students brought with them their own level of understanding and problem solving. Higher-order problem-solving, thinking, and abstraction skills are highly desirable, but cannot always be expected of novices. It was expected that students came in with low awareness, which, it was hoped, would change through using CSNüb. The feedback that comes from CSNüb during moments of disequilibrium highlighted flaws in the learners' solutions, which was a reflection of how they came to know the object-oriented system. In addition, CSNüb allowed those students who exhibited high awareness to exercise their understanding of the object-oriented system and to test whether their assumptions of classes were related.

Awareness affected how and what students explores. Students with a low range of awareness were only able to see a small part of the object-oriented system, and this limited their ability to see how classes relate and how objects interact, resulting in a smaller and more limited picture of a potentially large and complex system. Indeed, the relationship between awareness and exploration is bidirectional. As students explored more, they branched out into the different facets of the object-oriented system: they became familiar with ancestor, child, and sibling classes and became cognizant of other source code that connected these classes together. As a result, the range of their awareness widened, and this also guided future exploration. Problem solving and planning were also affected as they became aware of inherited source code and usable resources that might aid in their solutions.

### *Awareness and the MCT Design Framework*

Awareness is dependent on students' prior level of understanding. As they learn through interacting with MCTs, their awareness should increase. MCTs, therefore, need to bring attention to inadequacies in student' thinking — specifically, through multimedia-based feedback, since multimedia-based learning is preferable to single channel learning. Aside from multimedia advantages, MCTs are able to provide the necessary tools, resources, feedback, and information that students need to access to increase their level of awareness. The goal, thus, is to widen and deepen students' awareness through using the MCTs.

## Scaffolding

Scaffolding within CSNüb included resources and multimedia-based feedback that were used for the purposes of supporting and guiding cognition. Figure 5.4 illustrates the concept of scaffolding.

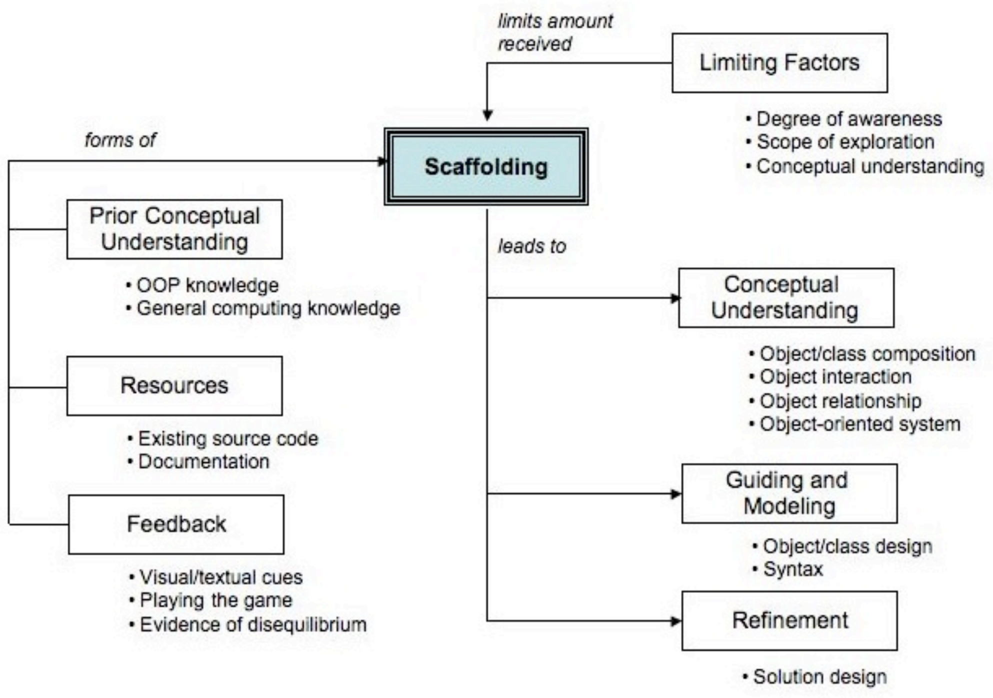


Figure 5.4: Scaffolding

CSNüb provided two forms of cognitive scaffolding: multimedia-based feedback and CSNüb resources.

The multimedia feedback from CSNüb was a form of scaffolding that directed students' attention to a problem or a situation. The code could be syntactically correct but had the potential of behaving incorrectly. It was through interacting with CSNüb and its visual and textual feedback that students could see whether their solution was correct. Every time incorrect and unexpected behaviors emerged from CSNüb, students gained more scaffolding that guided them in refining their solution. CSNüb also drew attention to aspects of their solutions (and code) that they had never taken notice of before — for example, when the submarine would not die after losing all its points, some students realized that their submarine class should have used code from its parent class.

Resources were also used as models when designing solutions, as when students looked for existing examples of class composition, relationships, and interaction. Additionally, resources provided models for programming syntax, aiding in the translation phase to working within a new environment like CSNüb. Nevertheless, even if students are provided with a bevy of cognitive scaffolds, they may be unaware that the information they are receiving is useful. A low scope of awareness, thus, may result in students ignoring the larger system and curtailing the range of exploration before it even begins.

### *Scaffolding and the MCT Design Framework*

MCTs are partners in learning: not only can they evoke moments of disequilibrium, they must also provide the scaffolding students need to achieve their learning goals. Multimedia-based feedback is one type of feedback that MCTs can provide to support conceptual understanding and to guide the problem-solving process. Students must have access to potential scaffolds, both multimedia or single-channel based. In some cases, students must be guided by the MCT to recognize and use supporting information and tools. An example of this situation is when students have low awareness and do not know that the cues they are seeing are for their benefit. In this case, scaffolding is also transformative. Scaffolding must also help relieve some of the cognitive load so that students can spend more time with higher-level thinking and problem solving.

**Refinement**

Refinement is the process of making students' understanding of OOP more correct, complete, and efficient through their solutions. Figure 5.5 illustrates the cognitive process of refinement.

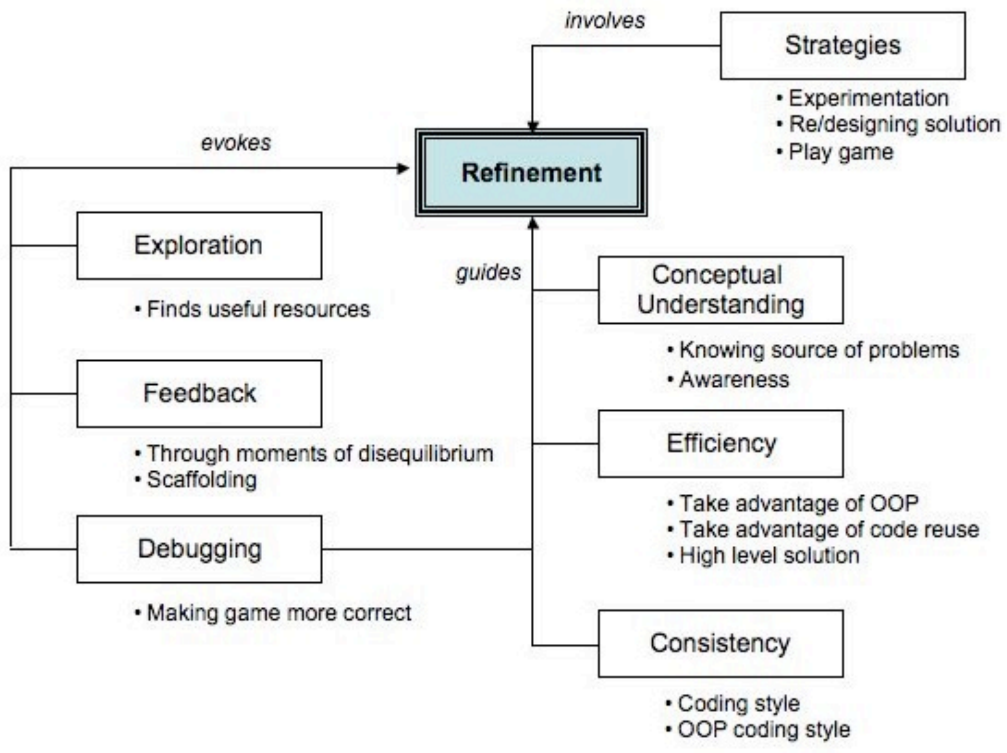


Figure 5.5: Refinement

In the present study, refinement operated on two sides: the cognitive side, in which students were refining their understanding of an object-oriented system and its individual components, and the programming side, in which students were applying their understanding within CSNüb.

Exploration evoked the refinement process when students discovered useful resources (e.g., inherited code, or important information) and adapted their existing solutions using this newfound knowledge. Resources might have informed students that their solutions were not optimal or were incorrect. Multimedia-based feedback that resulted in incorrect and unexpected behaviors from CSNüb also initiated the refinement process. Such errors showed that there were flaws with the design and/or implementation of a student's solution that needed to be fixed. Along with compile-time errors (e.g., syntax errors), feedback started the debugging process. This process was associated more with the programming side of refinement in that students were refining their code while also informing their conceptual understanding of object-oriented programming. Their own understanding, thus, served as the basis for the design of their solution.

The refinement process took place on several levels. When encountering higher level problems, students attempted to resolve their issues at the design level, that is, they reanalyzed their designs. These students took into consideration that their class must be part of the class hierarchy, that they needed to use inherited code, or that they were not taking advantage of object-oriented programming. They employed a top-down refinement process, addressing the problem on the design level first. On the other hand, when problems arose from syntactical, arithmetic, or logic issues, students tended to adopt an experimentation approach, in which they plugged in various numbers (or operators) and tested their solution to see what they would get. This represented a bottom-up approach to refinement, which was on a lower level. However, when the state or behavior of an object was indiscernible, at least on the code level, students depended on experimentation through interaction with CSNüb to figure out the nature of the solution.

Several factors affected the refinement process. Awareness, in terms of seeing the source of problems, was a factor that could initiate or bypass the refinement process.

In other words, if students were unaware of a problem, they assumed that everything worked correctly and no refinement was needed. The feedback from CSNüb brought students' attention to problems in the solutions they had devised. Refinement was not limited to making solutions more correct — it was also about making a solution more efficient and conforming it to standard coding styles. Some students wanted to take advantage of the code reuse affordance of OOP to make their solutions more efficient in terms of both time and design. When some students created new components of the object-oriented system, they attempted to find the place of best fit within the existing system so that all the classes worked in a coherent and logical manner. This raises the issue of how consistent learners are with respect to coding style. Some students, the data showed, consistently coded incorrectly (e.g., duplicating inherited variables), while others were inconsistent as to how they coded (e.g., mixing methods calls and direct variable calls when accessing outside data members). CSNüb, and the development environment on which it was based, was very lenient with respect to encapsulation and syntax, which allowed students to be inconsistent with their coding style.

### *Refinement and the MCT Design Framework*

Refinement of solutions leads to a refinement of mental models. MCTs must provide avenues by which students can refine their understanding through interaction. This occurs when students discover resources during exploration and when they face multimedia feedback that can confirm or reject their understanding. MCTs should constantly engage students in re-thinking and revising their own thoughts. Refinement may involve transforming the scope of awareness, affecting efficiency of students' solutions and their consistency in correct actions, and guiding future problem-solving strategies.

## The MCT Interaction Model (MCT-IM)

A broader theoretical model can be synthesized from the five cognitive processes and factors (moments of disequilibrium, exploration, scaffolding, refinement, and awareness) based on the levels of interaction with the MCT. As stressed in cognitive tool design (Jonassen, 2000; Lajoie, 1993), and throughout this and the previous chapter, MCTs are partners in learning. The present study showed how an MCT can be a cognitive partner and guide. CSNüb enhanced students' learning: it was through interaction with CSNüb on many levels that students were able to complete their tasks while developing a better understanding of OOP. Now these interactions can be applied to the broader category of MCTs; rather than focusing on using MCTs to teach OOP, it is possible at this point to discuss these interactions within a general framework which can represent any concept or topic for instruction. The five cognitive processes and factors can be classified on varying levels of an MCT. The ensuing model can be referred to as the MCT Interaction Model (MCT-IM).

Figure 5.6 illustrates the MCT Interaction Model. Each of the processes and factors within the figure are related to each other. These relationships, already discussed in previous sections, are condensed in the following figure.

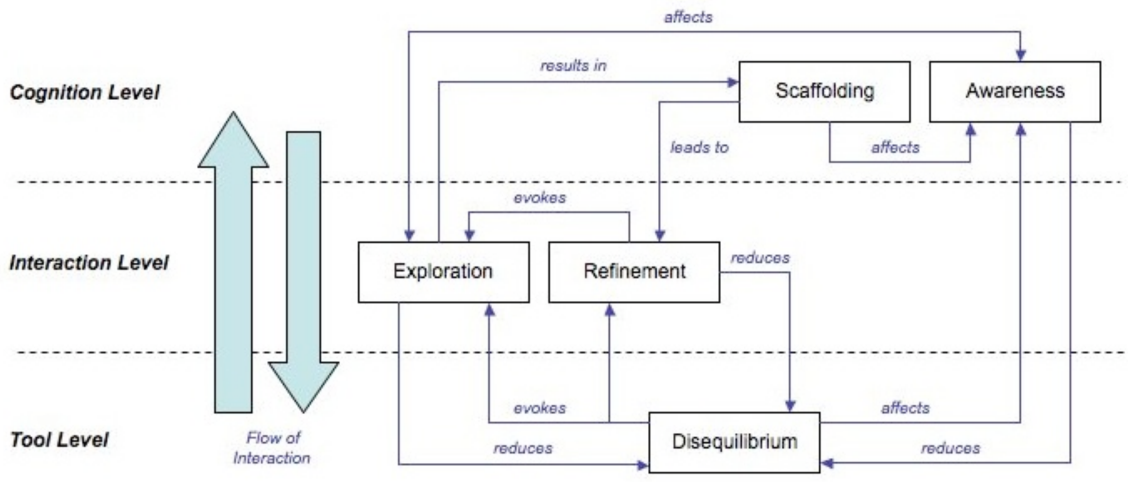


Figure 5.6: MCT Interaction Model (MCT-IM)

Interaction with the MCT occurred on three levels. At the *Tool Level*, students receive output directly from the tool. In this case, interaction is limited to one-direction: from tool to user. Within the MCT-IM, the MCT displays multimedia-based feedback for the student to process. In order for a change or a shift in cognition to take place, this feedback must induce or evoke moments of disequilibrium. Otherwise, students will assume everything is correct or will fail to notice something is incorrect, which is akin to an alpha reaction (Ginsburg & Opper, 1987). If something is incorrect, students' attention must be brought to it. The feedback provides cues that may bring students attention to errors or mistakes in their solutions, and help them in the cognitive processes of selection and organization (Jehng et al., 1999; Lockee et al., 2004; Mayer, 2001).

At the *Interaction Level,* the user is both receiving output from the tool as well as providing input into the tool. In the present study, this was most apparent in the exploration and refinement stage. The Interaction Level is the level at which students are working with the MCT to construct and re-organize their understanding. Students are empowered to seek out resources on their own when confronted with a problem so that they may better understand a larger concept. This empowerment leads students to a sense of autonomy in that they can find and use any resources they think best, design and implement their own solutions, and have a sense of control over how they want to solve problems (Brooks & Brooks, 1993; Marlowe & Page, 2005). Such autonomy may also facilitate a sense of control over their own situations, which may be intrinsically motivating for students to use and work with the MCT (Lepper & Malone, 1987; Liu et al., 2008; Malone & Lepper, 1987; Marlowe & Page, 2005). The refinement process also takes place when students are actively working with the MCT to create a better solution based on the feedback they receive. This involves several iterations of a particular solution, which changes as they receive more feedback and gain a more complete view of the concept as they adapt more information they find. In other words, interacting with the tool not only refines the solution itself, it also refines the students' understanding. The goal here is for refinement to make students' conceptual understanding more expert-like, so they can approach solutions from a higher-level of abstraction, conceptualization, and generalization of the problem (Derry, 1996; Ginsburg, 1997; Winn, 2003; Woody, 2001).

At the *Cognition Level*, students experience the after-effects of interacting with the MCT. Through exploring and refining, students receive cognitive scaffolding from the resources they find, as well as increase the scope of how they view the concept. The degree of awareness, that is, the level at which students see and understand the intricacies and complexities of the concept, presented in the MCT may be changed through the Interaction Level of the MCT-IM. The goal of an MCT is to increase students' awareness as much as possible so they can have a higher-level view of the system, just as experts approach situations from a general perspective and not through the minute or incomplete pieces on which novices tend to rely (Woody, 2001; Yuen, 2007b). As mentioned in the discussion of the Tool Level, the feedback that the MCT provides scaffolds students' understanding through visual and textual cues. Scaffolding was placed on the Cognition Level since it is also an effect of exploring and refinement. The visual and textual cues are used in the refinement process to change student understanding and knowledge structures. Often, students begin with an incomplete picture of the MCT's OO system, and it is through exploring the files and source code that they are able to discern the OO system, how classes are related, and how objects interact. Existing code also serves as a model for students and guides the design and implementation of their own solutions. Such resources are sources of scaffolding, as they help students better understand the state and behaviors of their objects. Another source of scaffolding is the students' prior knowledge.

The levels of MCT-IM are not mutually exclusive; rather, the flow of logic moves from one level to the next and vice versa. For example, on the Tool Level, a student may see that his submarine does nothing when it runs into a rock. He notices that this is a problem, and initiates processes on the Interaction Level. He knows that he has to refine his solution — he needs to make it work correctly — and does this by exploring files to see where the problem may be, and whether other code can help him. The student may find out that he inherited the rock class from the wrong class — he thought CSNub_Rock extended CSNub_Submarine because the two objects interact. He realizes, however, that CSNub_Rock should actually extend CSNub_Obstacle. "Extends" is equivalent to an "is-a" relationship. And since a rock is not a submarine, he is forced to reevaluate his design. This brings him to the Cognition Level of the MCT-IM where his awareness has changed. He now has a better understanding of how some of the classes fit within the hierarchy and how they are related.

The flow of interaction also works in the opposite direction (as illustrated by the arrows in Figure 5.1). With a wider range of awareness, the student may start to look at the bigger picture when trying to solve problems. He may have a better understanding of the OO, and when asked to create a new class/object, may now know where it should be placed within the system/hierarchy. Interaction with the MCT can guide students on where to start exploring for resources that can scaffold understanding or give them hints to solving the problem. This will also inform how a student refines his solutions: does he go with a plug-and-test methodology, or re-analyze his design first? Lastly, the feedback the student receives at the Tool Level can be used to inform a higher level of refinement, that is, students can use this feedback more effectively. Increases in awareness, correctness of solution (through refinement), and amount of exploration can also minimize the moments of disequilibrium a student can encounter.

The MCT-IM explains how an MCT affects student cognition based on looking at novice students working with CSNüb to better understand OOP. The MCT-IM shows three levels of interaction between the student and the MCT. At the Tool Level, the student is only receiving information. At the Interaction Level, the student is receiving and inputting information into the MCT. This is a continuing interactive process in which students are constantly learning. The processes on the Cognition Level are results of interacting with the MCT (Interaction Level). As students work with the MCT, they are receiving multimedia-based feedback that acts as cognitive scaffolding. Such feedback lets students know if their design and implementation were correct. At the same time, the multimedia-based feedback provides additional information as well as another representation of their understanding.

## An Updated Framework of Design Principles for Multimedia-based Cognitive Tools

The MCT-IM showed how learning was facilitated and conceptual understanding was affected through five cognitive processes on three levels of interaction with an MCT. This model can be used to re-inform the design framework on which CSNüb was based: the MCT design framework. The following sections explore the implications of this study's findings and the MCT-IM on each design principle.

### *MCTs should adopt a sensory modalities mode of delivery*

The sensory modalities mode involves the use of visual and audio (Mayer, 2001). Due to resource limitations, a sensory modalities approach was not used in the present study — only visual and verbal channels were used in CSNüb. The feedback from CSNüb, however, was important in getting students to rethink their own thinking and to reinforce what they already knew about their OO solution. Audio information should have the same effect in that students would be able to hear the effects of their code, just as students using the current version of CSNüb see the effects of their code.

As a cognitive tool, an MCT must assist students in cognitive reorganization when it confronts the student with unexpected or incorrect output based on their code. Pea (1985) discusses the use of multiple perspectives on the same piece of knowledge. In this case, the representation of knowledge is presented in both the game and code. When the game's output was incorrect — since it was inconsistent with what they believed they coded — students were thrown into a state of disequilibrium. The game's output was necessary for facilitate beta reactions to make students aware that something wrong was with the behavior of their code (Ginsburg & Opper, 1987). As a result, the feedback enabled them to start refining (and reorganizing) their conceptual understanding and their code based on what they saw from the game's output.

Resolving disequilibrium was based on students' adapting the information they received from their game's output. Piaget's (1952) term "accommodation" can be used to describe this process if it is assumed that the code was a representation of the students' mental models. When students saw that their game was not working, they tried to identify the problems in their solutions and to adjust their code. Thus, they attempted to accommodate their code to account for and avoid the incorrect behavior. From a constructivist perspective, this is why multimedia-based feedback is of utmost importance for supporting students' cognitive processes. Otherwise, students may experience alpha-type reactions in which they may be unaware of problems in their solution (Ginsburg & Opper, 1987).

This design principle can be updated as follows: *MCTs should adopt a sensory modalities mode of delivery to evoke moments of disequilibrium.*

***MCTs should engage students in higher-order thinking and problem solving***

Students are engaged in higher-order thinking and problem solving when they can think outside the box; that is, they must be able to have a bigger-picture understanding of the OO system and the problem itself. This is akin to the expert-novice differences that Woody (2001) described, where novices seemed only to look at the superficial aspects of a situation whereas experts look at the generalizations first. The issue of students' degrees of awareness with respect to the OO system illustrates Woody's point. When students lacked such abilities, they relied on resources (e.g., other source code, diagrams, and documentation) and prior knowledge in attempting to solve the problem. Bickhard (2005) stated that scaffolding makes "construction of competence or knowledge easier, or, perhaps, possible" (p. 169).

Resources such as the multimedia-based feedback, source code, and system documentation served as scaffolds that helped students transcend their own actual levels of development. These resources can also assist students in the active processing of information where important information can be highlighted through visual (or audio) cues for the purposes of decreasing cognitive load (Mayer, 2001). These affordances, transmitted through multimedia, allow students to work within a zone of proximal development (Vygotsky, 1978). In the present study, this was highlighted in the exploration process when students discovered code that helped them increase their awareness of the OO system, and modeled code and functionality. A continual driving point is that MCTs do not teach; students work with them to learn, as Jonassen (2000) explained in his description of Mindtools. If scaffolding is needed, it must be provided by the MCT or students must rely on their prior knowledge as a form of self-scaffolding (Bickhard, 2005). In this respect, this design principle is generally unchanged except that scaffolding was the key factor in keeping students engaged. In other words, *MCTs should scaffold students in higher-order thinking and problem solving*.

### *MCTs should engage students in metacognition*

The updated first principle of the MCT framework suggests that MCTs should strive to evoke moments of disequilibrium. Metacognitive processes (such as re-thinking one's thinking, inner speech, and adaptation) facilitate the resolution of disequilibrium (Greeno et al., 1996; Wells, 1999, 2000). Therefore, the third principle of the original MCT framework can be subsumed by the updated first design principle.

### *MCTs should promote student autonomy*

Student autonomy — that students must have control of their learning process — is a central tenet of constructivist environments (Brooks & Brooks, 1993). In the present study, such control was best illustrated when students would explore different files and source code within CSNüb for help in solving problems. They sought out resources and used what they thought would be best for their own solutions. When students were refining their games, they again applied their own strategies and techniques for improving their solutions.

The open-endedness of the tasks in the present study also contributed to student autonomy. Aside from the confines of the task requirements and the CSNüb architecture, students could initiate any strategy in solving the problems through varying levels of consistency and efficiency. Students had a wide range of choices in how they could solve problems and there were various ways in which their solutions could be implemented. For example, some students made design decisions that were incorrect and did not take advantage of the OO design; however, they implemented a solution that worked for them and even appeared to work correctly. In contrast, some students wanted to improve CSNüb's design to enhance its sophistication, and did so without affecting the outcome. From a constructionist perspective, students were building their own artifacts that represented their own learning (Papert, 1991). This resulted in students taking a personal ownership over their game while maintaining control of their own learning. The goal here is to create a learner-centered environment and, with minimal guidance, give students free reign on their learning process. Hence, *MCTs should facilitate learner-centered environments*.

*MCTs should provide intrinsically motivating experiences*

Though it was hoped that students would exhibit positive or negative emotions when working with CSNüb, they generally did not and simply focused on the tasks. However, several key issues arose that relate to specific components of intrinsically motivating environments. Malone and Lepper (1987) stated that an intrinsically motivating environment must facilitate control, challenge, curiosity, and fantasy. The present study's findings tie directly into the elements of control and curiosity. (Control is related to the issue of student autonomy and was discussed in the previous section.)

Malone and Lepper (1987) described two dimensions to curiosity, the sensory and the cognitive. Sensory curiosity was evoked by the multimedia feedback that students received from testing their games, and served the purpose of scaffolding students' conceptual understanding of their code's behavior and the OO design of CSNüb.

In terms of cognitive curiosity, students were asked to solve a problem and were given the CSNüb template as a partial solution that they could use. The template can be a metaphor for incompleteness, which students resolved by exploring the template further. The exploration process is an example of students trying to gain a more complete and deeper understanding of the OO system in order to help them complete the tasks. This falls in line with the notion that MCTs should create moments of disequilibrium. Most discussion in the literature has focused on multimedia-based feedback and its use to evoke disequilibrium. Additionally, multimedia-based feedback can evoke cognitive curiosity in students as a way to elicit student interest in the problem space.

Since affect was not a main focus of this study, it is hard to judge the effectiveness of this design principle or assess whether it should be included as a design principle at all. The findings did, however, identify cognitive processes and factors that relate to specific elements of intrinsically motivating environments, so it is at least possible to say that control and curiosity can be provided through an MCT based on its constructivist design component with the hope that it intrinsically motivates students. Thus, *MCTs should stimulate sensory and cognitive curiosity*.

## IMPLICATIONS

In addition to theoretical implications for the MCT design framework, the results of this study also inform the areas of instructional technology and computer science education. As the MCT design framework was updated through this study, the results can also be used to update CSNüb.

### MCTs versus Traditional Instruction

I have presented CSNüb to many audiences, and one of the most frequent questions is, "How does CSNüb teach students?" CSNüb does not teach — at least not directly. One principle is that MCTs should promote student autonomy by having students rethink their own thinking, manage their own thoughts, and be in charge of their own learning process. In other words, the distinction is that MCTs are tools with which students can use to learn, but MCTs do not explicitly teach anything.

How do MCTs differ from traditional programming assignments, and what makes them better? A recent study conducted by Lenhart, Kahne, Middaugh, Macgill, Evans, and Vitak (2008) of the Pew Internet & American Life Project reported that 97% of teens (ages 12 to 17) play video games. Their study also found that 36% of teen gamers play role-playing games — the same genre as CSNüb. Furthermore, the study revealed that approximately 30% of teen gamers use code modifications ("mods") to alter the games they play. These teens represent the future student population in university classrooms, and it can be assumed that these students will probably have interests and experiences in game-like environments that are inundated with multimedia and interactivity. MCTs place a focus on using multimedia components to aid in learning. Of course, this effect can also be achieved by having students work with graphical user interface (GUI) kits found in Java and Visual Basic, or look at textual output from a command-line type of program. It is through multimedia components, however, that students can see (or hear) an implementation of their code. Interactivity allows students to manipulate, experiment, and explore their understanding. MCTs operate under the assumption that multimedia learning is better than purely textual learning and that tools are partners in learning. The literature declares that sensory curiosity is a factor in intrinsic motivation, and this can be used to empower students to explore their understanding through visual and textual means (Malone, 1981; Malone & Lepper, 1987; Mayer, 2001). There is also the argument among computer science educators that today's university students, like the teens discussed in Lenhart et al. (2008), are accustomed to multimedia environments (Guzdial & Soloway, 2002). In developing MCTs, the instructional designer must purposefully find ways to use the multimedia components to evoke and take advantage of moments of disequilibrium. The instructional designer must also find and provide resources within the tool that can scaffold students' learning and conceptual understanding. This is not merely giving students a book; rather, students must be able to access information on demand.

Learning is accomplished when students create their own games while gaining a better understanding of object-oriented programming. Students can discover relationships between classes through reusing code and thinking about how the objects in the game can be organized from their real-life associations. With MCTs, students can see (or hear) how objects interact and how objects' states change through the multimedia-based feedback. Since video games are generally interactive, students can also play and experiment with the objects on the screen to see how their actions change the state of the objects, and how that may affect other objects. As an MCT, a game template allows students to work with the programming and the testing sides of video game creation, with both sides being able to facilitate conceptual understanding of OOP.

## Multimedia-based Cognitive Tools in Computer Science Education

Another frequent question about CSNüb is, "How is CSNüb different from the tools already used in computer science education?" This raises the issue of how a video game template can be used as an MCT in computer science education; this can be discussed through describing four differences between CSNüb and typical CS education tools.

1) Though powerful development tools, Alice, BlueJ, Greenfoot, and jGrasp are not development environments widely used in the technology industry, such as Eclipse and Microsoft Visual Studio. CSNüb, in contrast, situates itself within a professional development environment, Adobe Flash, which is an industry-grade multimedia-authoring tool. Students are working within this environment to construct a video game while exercising their understanding of OOP. Game development is an authentic task, and Flash affords the ability to make fully interactive multimedia applications. Students can also delve into the graphic design aspect of game development, if they are interested. Another unique aspect of CSNüb is that Flash is not a widely used tool in computer science education.

2) BlueJ and jGrasp visualize code on a line-by-line basis where students can see how programming logic and data flows through a program. CSNüb is not used for code visualizations; it differs from jGrasp and other code visualization in that it focuses students to think on a higher level, not just on the level of syntax. Syntax and programming logic is important and students will have to deal with it in CSNüb, but the activity focuses on object-oriented design: incorporating changes to and using parts of an existing OO architecture. Different resources in CSNüb (e.g., multimedia-based feedback, previous code base, the metaphor of the game objects for OO relationships) provide adequate scaffolding for students to understand OO better. They are given just enough scaffolding that is either highlighted through feedback or discovered through exploration from CSNüb to enable them to progress.
3) Alice is a popular tool in which students create their own movie with characters and objects. In the most popular versions, students "program" by clicking on objects in the scene. This brings up a menu of possible methods that can be called and variables to change. The programming logic is constructed through a drag-and-drop interface. CSNüb does not offer such guidance; students must manually explore class files to know what methods and variables are available to them. With CSNüb, students are also responsible for their own logic. Aside from animation and graphic design, the engine is not hidden from students. Students have complete access to any part of the game template they wish. It should be noted that a current version of Alice allows students to program through more traditional means (e.g., Java) using the application programming interface (API).

4) CSNüb takes advantage of students' experiences with and interests in video games and other similar interactive multimedia environments. Due to the modularity of video game architecture (e.g., characters, objects, interactions, etc.), it seems to be quite adaptable to object-oriented instruction. The CSNüb template contains a simple game engine that students can learn with a minimal amount of time. The engine is merely an event listener that continuously loops, and therefore there are not as many complexities and limitations as a full game engine (e.g., Quake) would have.

The MCT design framework is not just a framework for designing instructional tools; *it is also a framework for using a tool in a manner that supports a constructivist learning environment through multimedia while intrinsically motivating students to learn.* In the end, it does not matter whether Flash is used, or Java, or C++. The activities or tasks centered around the tools are just as important as the tool itself. It is possible to take an existing tool and adapt it so that it provides multi-sensory information, causes moments of disequilibrium, creates intrinsically motivating environments, and stimulates cognitive and sensory curiosity. The key is to make these tools act as a partner in the learning process and to thrust the student into his or her zone of proximal development through multimedia (Jonassen, 2000; Vygotsky, 1978).

Though the present study focused on object-oriented programming, other topics in computer science require as much abstraction and high-level thinking power as OOP, including algorithms, operating systems, data structures, computer architectures, and programming languages. Many of these concepts are intangible; that is, there is no actual physical representation students can work with to explore and apply their understanding. Some of these topics, such as algorithms and data structures, involve several transactions in which possibly large amounts of data are being shifted around according to specific rules. In many cases, there are no actual graphical representations with which students can work (e.g., operating systems). MCTs can help alleviate the amount of processing and lower-order thinking while allowing students to concentrate on the higher-order issues such as design and optimizing. Whereas programming assignments are largely based on textual programs, MCTs can add another level of feedback on visual and auditory levels.

MCTs also go beyond using graphics just for the sake of illustrating a programming concept. An MCT can motivate students by giving them an environment similar to those many of today's students are familiar with: rich, interactive multimedia environments. As mentioned in Guzdial and Soloway (2002), the students today grew up with multimedia environments (e.g., television/films, video games, the internet). Some students may be disillusioned by computer science when the "Hello, world!" programs they are creating are nothing like the Playstation or XBOX games they have been playing their entire lives. And to reiterate, Lenhart et al. (2008) found that that 97% of teens ages 12 to 17 play video games. MCTs can help engage and motivate students to apply their knowledge within a fun and familiar context.

## Future of CSNüb

Based on what was learned in the present study and the updated MCT framework, there is definite room to improve CSNüb, both as an MCT and as a piece of technology. This sections discusses what changes will be made to the new iteration of CSNüb, which will be referred to as CSNüb 2.0. Due to resource constraints, the multimedia aspect of CSNüb was restricted to visual and textual information even though the first principle of such design is to use a sensory modalities perspective. Since sensory curiosity on only the visual level may not be enough to spark interest, it will be useful to add audio. This can range from affixing sound-bytes to each character to having a soundtrack to the game.

The original version of CSNüb was created in ActionScript 2.0. AS 3.0 was released shortly after the study had started and the next iteration of CSNüb will update the code to conform to AS 3.0. This will only affect the code for the game engine and none of the other source code. In addition to the *hit_points, attack_points,* and *defense_points*, the CSNüb 2.0 will include weapons and tools that would augment these point values. Such objects will add a layer of complexity with respect to object interaction and the expanse of the CSNüb architecture.

Task 1 had almost no feedback, so participants were unable to see the results of their code. The CSNüb 2.0 will display of each character's *hit_points, attack_points,* and *defense_points* through mini-bar graphs above each character. Whereas in the current version of CSNüb interaction between the submarine and any other object was described textually (in the *display_panel*), this will be enhanced so that the game player can see an actual interaction going on between the submarine and an object.

CSNüb's RPG features were scaled down for this study so that the tasks were more focused and more environmental variables controlled. Thus, many traditional RPG features were missing. In the study, participants were asked how they would implement two new features, though they were not actually required to implement them. The first new feature is a class that represents a group of CSNub_Character objects. The next new feature is that all CSNub_Characters will be able to carry one or more item. To make the game more like traditional RPG games, game players will have the option of selecting what to do when the submarine interacts with another character. They will have the option to attack the other character, to defend, to use an item, or to run away from the fight. The functionality for this will already be provided, and participants will not have to implement these. This provides further scaffolding through multimedia-based feedback.

## LIMITATIONS OF THE STUDY

Due to its scope and allocation of resources, there were several limitations to this study.

### Applicability of Findings

Since CSNüb was a newly developed tool, it can be assumed to work for everyone or with the intended population. This study was conducted with a small sample size in order to explore in-depth what aspects of MCT design contribute to conceptual understanding so that future iterations of both CSNüb and the MCT design framework can be revised. Therefore, the findings may not be applicable to the entire population of novice students in computer science. Further, the participants of this study were all male. Participation was voluntary: participants were self-selected and no females volunteered for this study. Though computer science has traditionally been a male-dominated major, female students have entered the field, and there is a move to increase their enrollment at the undergraduate level. This may limit the applicability of the present study's findings and the MCT framework in instructional tool design, as the socio-cultural factors of different users and learners were not explicitly explored. The results can, however, be used as a starting point for future students with the MCT-IM and MCT design framework.

## First Iteration

CSNüb was a prototype in two respects. First, CSNüb represented the first implementation of the MCT framework. As such, it may not entirely meet expectations and cannot be assumed to work completely. Next, CSNüb was still regarded as a software prototype. As with any software, despite exhaustive testing, bugs and errors are inevitable. Minor technical errors did occur during the activities. However, there were no technical "blockers" that entirely stopped progress or completion of the study. Participants' usage of CSNüb revealed needed technical improvements. Future iterations of the MCT framework and CSNüb should be informed and improved by the findings of this study.

## Flash and ActionScript in Computer Science Education

Flash is a widely used tool in the graphic design and web development communities. It is not a common tool found in traditional computer science education, and the software is expensively proprietary. Studies have been conducted on Flash as a way to teach CS1/CS2 courses (Crawford, 2006; Leutenegger & Edgington, 2007; Moses, 2006), but this was not the focus of the present study.

In computer science education, researchers and practitioners are always looking for novel ways to teach computing to students. Indeed, that has been described as one of the grand challenges in computing education (McGettrick et al., 2004). It is also the focus of many studies that use multimedia to teach and motivate students in introductory CS courses (Conway et al., 2000; Powers et al., 2007). Game programming has also been used to stimulate student interest in computer science (Chen & Cheng, 2007; Frost, 2008; Sung et al., 2008). With the increasing number of web-based applications appearing, Flash knowledge will be a practical and beneficial skill for computer science students who end up in those fields.

Some tools like BlueJ and Alice have their own development environments that are unique to the tools themselves and are not widespread in the professional communities. In earlier versions of Alice and some versions of LEGO Mindstorms, programming consisted of drag-and-drop utilities, which is not the traditional method of programming. The present study has shown that using Flash is also a viable alternative in environment and programming language choice. Aside from a few syntactical differences between it and Java/C++, the participants in my study seemed to be comfortable with coding in ActionScript. Their prior knowledge in Java contributed greatly to the ease of transferring from one programming language to another.

The main difference between using Flash and a more conventional language (e.g., Java) and IDE (e.g., Visual Studio) is Flash's ability to incorporate interactive multimedia into its applications. Participants in this study did not delve into the multimedia aspects of Flash — their involvement with multimedia came from dragging symbols on the stage and interacting with the game. The animation and graphics aspects of Flash would add another level of complexity beyond the scope of an introductory computer science class, and would be better suited to graphic design or art classes. The multimedia output — the game — that the participants created is in the present study different than the line-based applications common in CS classes, such as the infamous "Hello World!" (Stein, 1998). It is possible to immerse students in a multimedia environment without focusing too much on the graphic design or sound production parts, as it was shown in this study. AS 2.0 and 3.0 fully support object-oriented programming, and the syntax follows the ECMA standards. Transitioning between ActionScript and Java should be transparent, especially since both are within the

OOP paradigm and stress conceptual understanding of programming languages rather than specific syntax. Thus, the use of Flash as a viable alternative in computer science education would need to focus on the coding portions of a program and not the multimedia aspects. The main limitation of Flash is the high cost of the software compared to the wide range of free development tools for Java, C++, and other modern programming languages.

## Exposure Time

The expected exposure time for CSNüb participants in this study was about 3 to 5 hours. Since course curricula and standards are difficult to change, CSNüb cannot be easily incorporated into current courses, especially since Flash is not widely used. Time was required to learn the Flash environment and the CSNüb template. This study was conducted within a very small timeframe (compared to an entire semester or unit) and outside of normal class time.

## Future Study

Now that the MCT framework has been updated and new updates for CSNüb 2.0 have been discussed, new studies need to be done. The sensory modalities delivery of information was not tested thoroughly due to the researcher's resource limitations. Future iterations of CSNüb can take advantage of the full range of multimedia with the inclusion of sound. Students in the present study were able to recognize a problem with their code by seeing and reading the unexpected output; audio will add another sensory level of perception of the problem.

Since MCTs should be applicable to any content area, studies of using the MCT framework in other areas should be conducted. This would require different tools to be developed, but those new tools could still follow the MCT framework. This study was exploratory, but the findings are adequate for revising the MCT framework and give some preliminary findings as to how such tools can be used to support conceptual understanding through the MCT-IM. Due to the small sample size, however, the findings of this study cannot be generalized to the entire population. This requires controlled experiments with a much larger sample size across multiple campuses. CSNüb is still not ready for assessment for a widespread application due to the theoretical revisions to the MCT framework. Another more focused, qualitative approach must be taken before such experiments are conducted.

Another area for future study is the evaluation of the MCT-IM model. Since all the cognitive processes, factors, and levels of interactions are interrelated, it is possible that there are alternative ways in which the MCT-IM components can be arranged. The "chain of events" are presented in the following order: Tool Level, Interaction Level, Cognition Level, and vice versa. This ordering was a result of the way the findings were discovered in this study: the tool was the agent for change in cognition, and was placed as the "middle" layer. Future study of the MCTs may yield a different ordering scheme; it could be argued that the relationships between levels of interaction are not linear. In the current MCT-IM, there are indications that the Tool Level also affects the Cognition Level directly. Future study can create a deeper understanding of how these levels are related to each other.

## SUMMARY AND CONCLUSIONS

Issues of attrition have long been noticed in introductory computer science courses (Doube, 2004; Forte & Guzdial, 2005; Herrmann, Popyack et al., 2003; McKinney & Denton, 2004). The present study was intended to address student attrition as well as some of the grand challenges in computing education, such as improving public perception of computing, providing innovative ways to teach; and establishing a solid foundation for life-long learning (Beaubouef & Mason, 2005; McGettrick et al., 2004). The solution proposed in this study was to develop multimedia-based cognitive tools whose design draws from constructivist, multimedia, and motivation learning theories and computer-based cognitive tool design. The resulting MCT framework included the following design principles:

1. MCTs should adopt a sensory modalities mode of delivery.
2. MCTs should engage students in higher-order thinking and problem solving.
3. MCTs should engage students in metacognition.
4. MCTs should promote student autonomy.
5. MCTs should provide intrinsically motivating experiences.

The MCT framework was developed in the same tradition of visualization tools in computer science education in which researchers and practitioners have developed tools to help make the abstract concepts in computer programming less abstract and more real to novice students while also motivating them to stay in CS (Barnes, 2002; Boyle et al., 2003; Brusilovsky et al., 2006; Conway et al., 2000; Guzdial & Soloway, 2002; Henriksen & Kölling, 2004; Herrmann, Popyack et al., 2003; Jehng et al., 1999; Kölling & Rosenberg, 1996; McNally et al., 2006; Moreno et al., 2005; Powers et al., 2007). The framework was then implemented in the form of a game template, CSNüb, in Adobe

Flash, which students used to create a simple role-playing game without having to deal with much of the graphics or animation aspects of the game. This study investigated how CSNüb affected novice computer science students' conceptual understanding of object-oriented programming. Object-oriented programming is one of the major concepts in computer science education that are regarded as difficult and it has led to many debates as to when in the curriculum it should be taught (Dale, 2006; Kölling, 1999).

Data were mainly collected using process-tracing methods during clinical interviews with a focus on behavioral protocols (Hayes & Flower, 1980, 1983). A grounded theory applied to data analysis identified recurring themes and patterns of behavior using the microanalytical techniques of coding, and resulted in the discovery of five categories of cognitive processes and factors affecting conceptual understanding: exploration, refinement, scaffolding, awareness, and disequilibrium. It was found that CSNüb affected novice computer science students' conceptual understanding of OOP through five cognitive processes and factors: cognitive *disequilibrium* evoked through multimedia-based feedback, *exploring* for resources that *scaffold* understanding, changing the level of *awareness* of the "bigger picture" and ability for higher-level thinking, and consistent *refinement* of solutions and mental models within the problem space.

It became evident that moments of disequilibrium served as the catalyst for the cognitive processes and factors that led to cognitive changes. Such changes occurred on three levels of interaction with CSNüb. On the Tool Level, the multimedia feedback from the game initiated the exploration and refinement processes. At the Interaction Level, exploration and refinement were the processes by which students worked and interacted with the MCT to gain a better understanding of an OO system, and ultimately

OOP. These processes led to the discovery of resources that scaffold understanding, widened awareness, and improved conceptual models, which are at the Cognitive Level. These processes and factors and levels of interaction were brought together under one theoretical model. This model is the MCT-Interaction Model (See Figure 5.1).

These findings answered the research question, but the results also informed the theoretical design of CSNüb, which was based on the MCT design framework. The MCT design framework was revised to take into consideration how the cognitive processes were affected by using CSNüb.

1. MCTs should adopt a sensory modalities mode of delivery to evoke moments of disequilibrium.
2. MCTs should facilitate learner-centered environments.
3. MCTs should scaffold students in higher-order thinking and problem solving.
4. MCTs should stimulate sensory and cognitive curiosity.

This study has shown that using a game engine can spark various cognitive actions that help transform students' understanding of an abstract and complex concept such as object-oriented programming. They were immersed within an OO problem and required to implement tasks that required a bigger-picture understanding. The MCT-IM shows the different levels of effects MCTs can have on conceptual understanding. The new framework uses the study's findings to promote the use of multimedia to facilitate engaging and supportive learning environments. MCTs need to become partners in learning by giving students the ability to take charge of their own learning while giving them the appropriate guidance. MCTs should also shoulder as much cognitive load as possible to make room for students' higher-level thinking.

Though this study has produced a useful tool for computer science education, the larger result is that a theoretical framework for MCTs has been developed and revised. The resulting MCT framework is offered as a guide for instructional designers who can use it to develop computer-based learning tools with the goal of helping students better understand a concept through multimedia.  And as a theoretically-based framework, the MCT design framework should not be limited to the realm of computer science concepts; rather, the power of a theoretically-based design framework lays its versatility in any similar content domains.

# GAME OVER!

Congratulations! You have successfully cleared the ocean floor of dangerous sea life and collected all the items. Job well done, Captain.

# Appendix A – Learning Objectives for OOP

These learning objectives are based on unique features of object-oriented programming as opposed to other programming paradigms (Ben-Ari, 1996; Pratt & Zelkowitz, 1996; Smith, 1991). Encapsulation and inheritance, the topics covered in this study's activity, are also included as topics to cover in a CS102 type course, which is an introduction to the object-oriented paradigm (ACM-SIGCSE, 2001).

## Encapsulation

1. The student shall modularize their solution such that it follows an object-oriented design.
2. The student shall encapsulate properties and methods into a single object (class definition).
3. The student shall make distinctions between which properties and methods are kept private within the object and public to other objects.

## Inheritance

1. The student shall be able to find a hierarchical relationship between similar objects.
2. The student shall extend existing classes to derive new classes.
3. The student shall use properties and methods from inherited classes in derived classes.
4. The student shall override inherited methods in derived classes.

## Appendix B.1 – Participant Survey Form

Pseudonym:

Age:

Gender:

Year/Class in School:

Major and Minor(s):

Previous computing/programming experience:

Why are you taking this computer science class?

What programming languages do you know?

What is your experience with Object-Oriented Programming?

## Appendix B.2 – Preliminary Interview Questions

**General OOP Questions**

- What is object-oriented programming?
- What makes object-oriented programming different from regular programming?
- What do you use object-oriented programming for?
- What makes a program object-oriented?
- How were you taught object-oriented programming?

**Encapsulation**

- What is an object?
- How do you decide what goes into a class definition?
- What does it mean when variables and methods are public or private?

**Inheritance**

- What is inheritance?
- What does extending a class do?
- What happens to the variables and methods when this class is extended?

## Appendix C.1 – CSNüb Tutorial Activity

**Introduction**

The purpose of this tutorial activity is to familiarize you with how to make a simple game using the CSNüb template in Adobe Flash. This tutorial will explain the different tools provided by the template and step-by-step instructions on how to create a game. The prerequisites for this activity include foundational programming knowledge, C++ or Java-type syntax, and a beginner level understanding of object-oriented programming.

**Operation SPLASH**

Operation SPLASH is a two dimensional video game in which the user pilots a submarine around the ocean floor. The submarine must remove objects from the ocean floor. Objects include dangerous sea life (e.g., squid, eels) and items (e.g., barrels). When encountering such objects, the submarine may be attacked by the dangerous sea life or affected by items. The submarine has a set amount of hitpoints that it has to maintain to stay afloat. There are also obstacles on the ocean floor such as kelp and rocks. These barriers block the submarine from going over it while causing some damage. The goal is for the submarine to clear the entire ocean floor without losing all its hitpoints.

**Points**

All characters have a set of points that represents its status. Hitpoints (HP) represent the amount of damage the character can receive. When HP reaches 0, the character is considered dead. Attack points (AP) refer to the amount of damage one character can inflict on another. The defense points (DP) refer to the amount of damage a character can absorb from another character's attack without losing HP.

**Mac versus PC**

The screen captures shown in this tutorial were done on a Macintosh computer; however, the tutorial can be used for either Macs or PCs. When right click is not available (e.g. on a most Macs), hold down the Ctrl key while clicking as alternative.

**Differences between Java/C++ and ActionScript 3.0**

Though the syntax between Java/C++ and ActionScript 3.0 are very similar, there are several differences between the two languages that are important for CSNüb.

| Java/C++ | ActionScript 3.0 |
|---|---|
| float x; *or*<br>double x; *or* | var x:Number; |
| bool x;     // in C++    *or*<br>boolean x;     // in Java | var x:Boolean; |
| String x; | var x:String; |
| float x = 15; | var x:Number = 15; |
| int x = (int) 3.14159; | var x:int = int( 3.14159 ); |
| public Constructor()<br>{<br>} | public function Constructor()<br>{<br>} |
| public void fubar( int x )<br>{<br>        // do nothing<br>} | public function fubar( x:Number ):Void<br>{<br>        // do nothing<br>} |
| System.out.println( “Hello!” ); | trace( “Hello!” ); |

**Additional Issues**

In ActionScript 3.0, Boolean values can only be true or false.   0 and 1 are not suitable alternatives.

Abstract classes cannot be explicitly defined.

Access modifiers such as public and private cannot be assigned to a class.

**Activity 1: The Hierarchy**

Purpose: This activity gives an overview of all the class files provided to you by the CSNüb template. You will explore some of the key classes that are central to making your game.

Objectives: Understanding the general design of the game, the design of each object, and the hierarchical relationships between the classes.

1. This is a chart of all the classes in CSNüb. Each class has its own ActionScript file of the same name. You will be expected to modify some of these classes as well as create your own. The italicized classes are not included with the template. There is a file associated with each class. Each file has the same name as the class and has an .as extension.

```
* CSNub_Object[3]
        |-----> * CSNub_Character
                        |-----> CSNub_Submarine
                        |-----> CSNub_Squid
                        |-----> CSNub_Eel
                        |-----> CSNub_Ogrefish
                        |-----> CSNub_Shark
        |-----> * CSNub_Item
                        |-----> CSNub_EnergyBarrel
                        |-----> CSNub_RadioactiveBarrel
        |-----> * CSNub_Obstacle
                        |-----> CSNub_Rock
                        |-----> CSNub_Kelp
        |-----> * CSNub_GameObject
                        |-----> CSNub_DisplayPanel
                        |-----> CSNub_EventHander
CSNub_ObjectRegistry
```

Abstract classes are starred—none of these classes should be instantiated since they are abstract. The classes that are not starred can be instantiated. Open each of these files and read through the comments found in them as well as the descriptions below. Below, you will see a graphic representation of the classes.

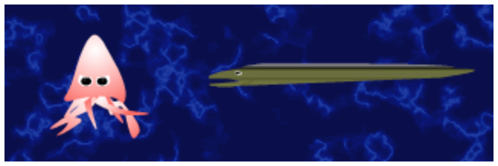
2. *CSNub_Object* is the base class for all objects within the template. CSNub_Object is essentially a wrapper for the Flash data type MovieClip.

3. *CSNub_Character* is an abstract class that represents any characters that exist in the game. All characters in the game descend from *CSNub_Character*.

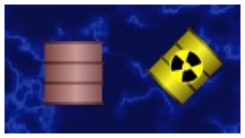
[3] To ensure that code compiles correctly, there are no special characters. The word CSNüb occurs throughout the code, though the 'u' in CSNüb does not contain the umlaut accent mark.

4. *CSNub_Submarine* is a derived class of *CSNub_Character*, which represents the main character, a submarine, for the game.

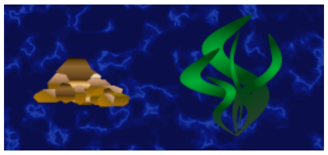
5. There are two defined enemies: *CSNub_Squid* and *CSNub_Eel*. Students can assign these classes to a variety of squid and eels found in the library. Enemy classes are also inherited from *CSNub_Character*.



6. *CSNub_Item* represents items the submarine collects and immediately uses. Using it affects one of the character's properties such as HP, DP, or AP. It is the base class for two classes of items: *CSNub_EnergyBarrel* and *CSNub_RadioactiveBarrel*.



7. *CSNub_Obstacle* is a derived class of CSNub_Object. It is also an abstract class and should not be instantiated. These objects prevent a character from going over or past it. When a character hits it, it loses some HP. *CSNub_Kelp* and *CSNub_Rock* are two examples of obstacles that come with CSNüb.

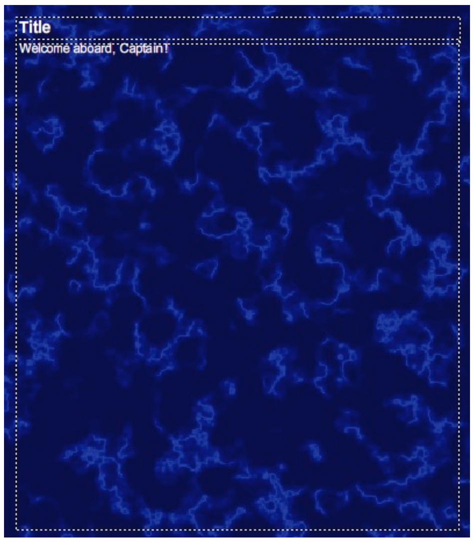


8. *CSNub_GameObject* is an abstract class for objects that pertain to the operations of the game such as an event handler, display and control panels. These objects are discussed below.

9. *CSNub_EventHandler* is an event listener. Thirty times per second, the event handler does the following: 1) checks to see if one character intersects with another character or item, 2) listens for keyboard input, 3) calls the move function for the character. It is also charged with initializing the entire game and checking to see if the player has failed or succeeded in the mission.

10. *CSNub_ObjectRegistry* keeps track of all the items placed on the stage. Essentially, *CSNub_ObjectRegistry* is a list of *CSNub_Object*'s.

11. *CSNub_DisplayPanel* represents a text-based panel that displays information—usually for a fight sequence or when a character runs into something else on the stage. This class only displays clear text, which means that there will be no text formatting (e.g., colors, fonts, sizing). This movie clip for *CSNub_DisplayPanel* is *display_name* in the Library in the *game objects* folder. The outline and text are white and may not appear visible unless it is over a dark area like the background as in the image below.

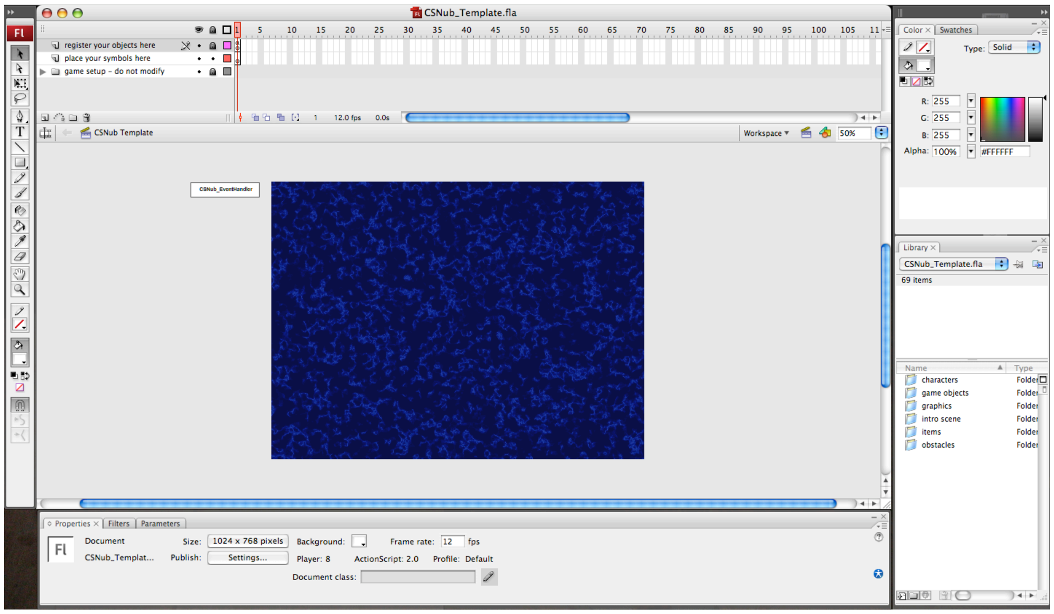


## Activity 2: Setting up the Submarine

Purpose: This activity familiarizes you with the Adobe Flash environment and the CSNub_Template by creating a basic game where the main character, a submarine, moves from the center of the stage upwards.

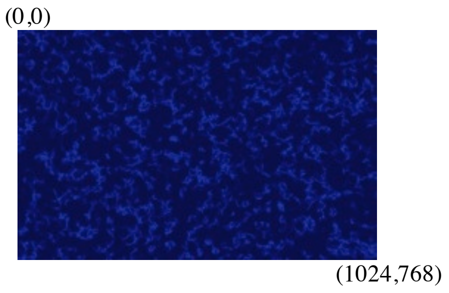
Objectives: Opening Adobe Flash and the CSNub_Template file, adding characters to the stage, assigning names to characters, adding characters to object_registry, and compiling and running your game.

1. Open the file CSNub_Template.fla in the CSNub folder in Adobe Flash. At the top of the window is the timeline. Notice the blue background that makes up the ocean floor below. This is the *stage* area. All objects need to be on the stage to be visible within the game.

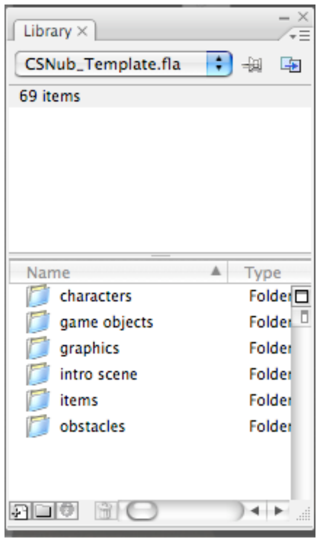


Document CSNub_Templat... Size: 1024 x 768 pixels Background: Frame rate: 12 fps
Publish: Settings... Player: 8 ActionScript: 2.0 Profile: Default
Document class:

When placing items on the stage (the blue background), the upper left hand corner coordinates are (0,0) and the lower right hand corner coordinates are (1024,768).

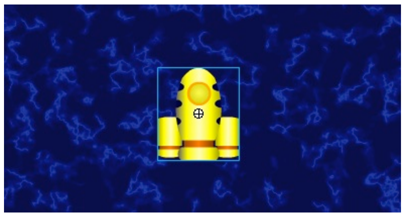


2. To start on your game, you will need a main character. If you do not see the Library window open, go to the Library (Window>Library).



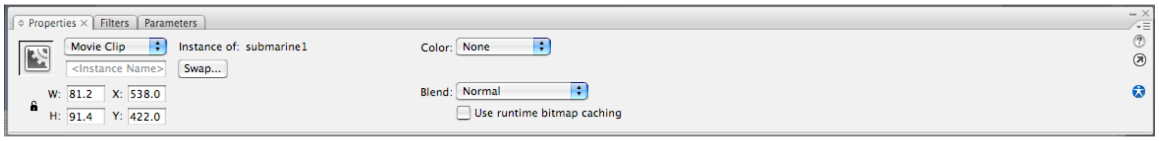
3. Maximize the characters folder by double clicking on it. You will see a list of characters you can put on the stage. There are three submarines from which you can choose. Select submarine1. For reference, submarine1 is the name of the movie clip graphic.

4. Click and drag the icon for submarine1 in the library to anywhere on the stage. It is preferable to place it as center as possible.



A yellow submarine character will now appear on the stage. This submarine is now your main character.

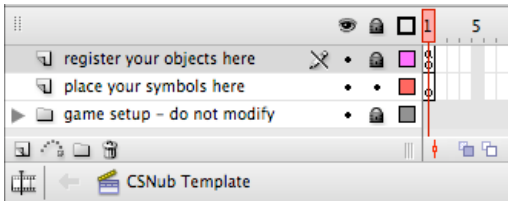
5. Although you now have a submarine on the stage, there still needs a way for the game to refer to this particular character and differentiate it from the other characters and objects that will later appear on the stage. This requires giving the submarine a name. Click on the submarine on the stage to highlight it (a blue bounding box will appear). Go to the Properties window (Window>Properties>Properties).



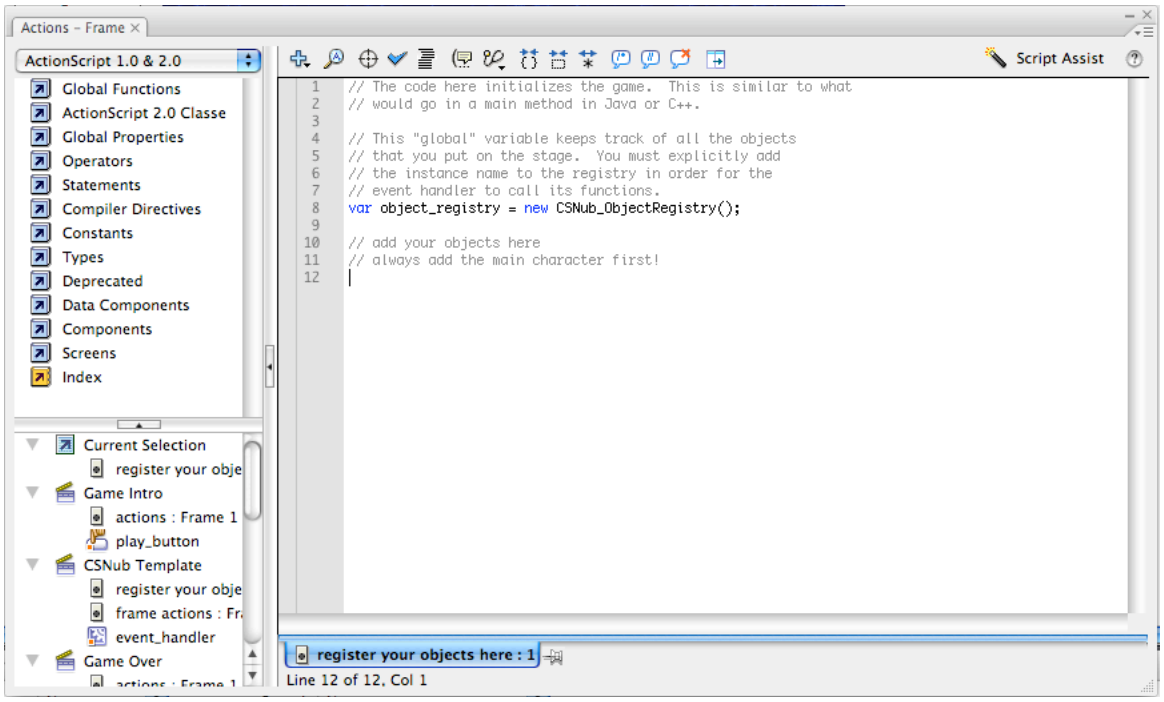
Movie Clip
Instance of: submarine1
<Instance Name>
Swap...
W: 81.2 X: 538.0
H: 91.4 Y: 422.0
Color: None
Blend: Normal
Use runtime bitmap caching

In the text box labeled Instance Name, give the character a unique name. For this example, enter *submarine*. This means that no other character will have the name submarine. So, submarine becomes the instance name for the yellow submarine character.

6. Next, you need to register your character with the game's *object_registry*. This requires some programming. The *object_registry* is a list of all the characters and objects that will be in your game. At every moment the game is running, specific methods are called within every registered item. Go to the timeline and click on Frame 1 on the *actions* layer in the timeline.



Open up the Actions window (Window>Actions).



```
// The code here initializes the game.  This is similar to what
// would go in a main method in Java or C++.

// This "global" variable keeps track of all the objects
// that you put on the stage.  You must explicitly add
// the instance name to the registry in order for the
// event handler to call its functions.
var object_registry = new CSNub_ObjectRegistry();

// add your objects here
// always add the main character first!

```

Line 12 of 12, Col 1

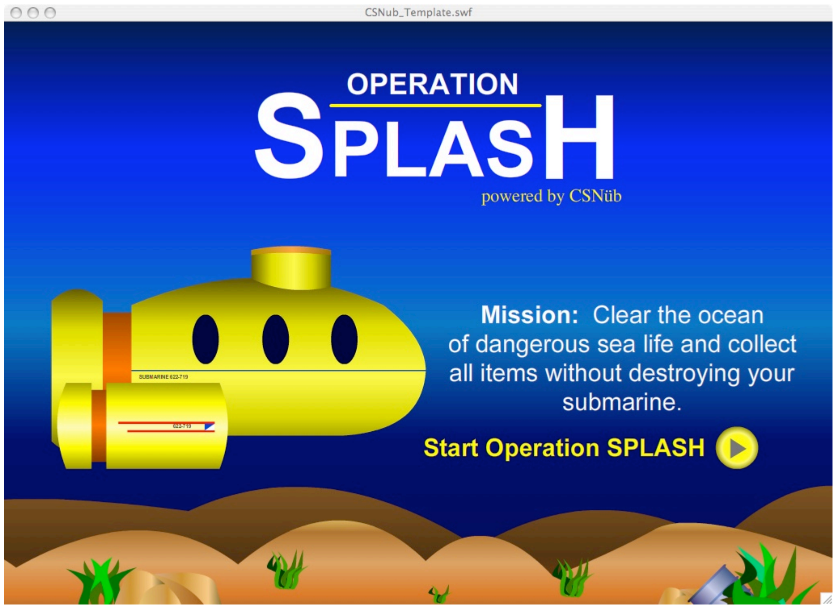
7. Note the following line. It initializes the object_registry.

```
var object_registry = new CSNub_ObjectRegistry();
```

Add the submarine to the registry by using CSNub_ObjectRegistry's addObject method.

```
object_registry.addObject( submarine );
```

8. Test your game by going to Control>Test Movie. A new window will pop up. There is an intro screen for the game. Go to the Control>Test Movie option every time you want to test your game. The beginning of your game should like this.


CSNub_Template.swf
OPERATION
SPLASH
powered by CSNüb
Mission: Clear the ocean of dangerous sea life and collect all items without destroying your submarine.
Start Operation SPLASH

If there are no errors, your game will run. If there are errors, your game may still run in a limited fashion. In this case, the Output window will appear with a list of errors.

9. Click the button labeled Start Operation SPLASH to start the game. You will see the submarine in the center for your screen. This is the "game" you have created so far.

10. You may now close this game window to return to the original Flash file.

11. Save your work (File>Save).

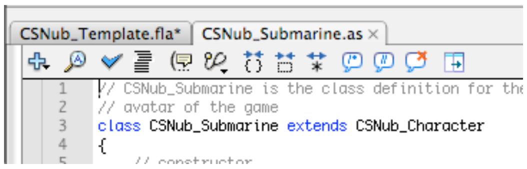
**Activity 3: Assigning Class Files**

Purpose: This activity shows you how to edit the code in the game's class files. Each character, item, obstacle, or object has its own respective class file. Use the CSNub_Template file you edited in the previous activity.

Objectives: Opening and editing class files

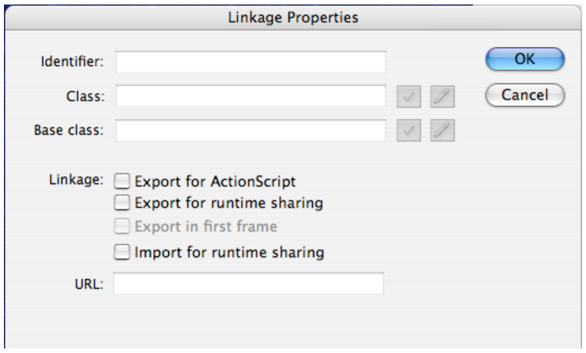
1. All characters and items in the library have an ActionScript class file associated with it. Open the file CSNub_Submarine.as (File>Open). This will open a new tab above the timeline.

2. Click on CSNub_Template tab to return to the actual game file.



```
// CSNub_Submarine is the class definition for the
// avatar of the game
class CSNub_Submarine extends CSNub_Character
{
    // constructor
```

3. The association between the class file and its movie clip representation must be explicitly linked. Go to the Library and select submarine1—the movie clip representation of the character. Right click on submarine1 and select Linkage... This will bring up the Linkage Properties window.



Linkage Properties

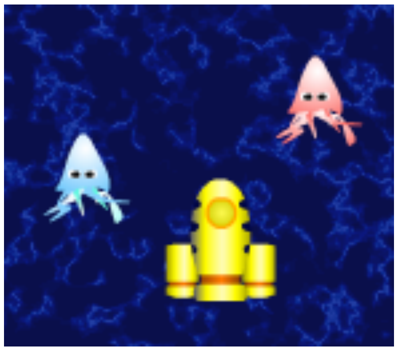
| | |
|---|---|
| Identifier: | |
| Class: | |
| Base class: | |
| Linkage: | ☐ Export for ActionScript |
| | ☐ Export for runtime sharing |
| | ☐ Export in first frame |
| | ☐ Import for runtime sharing |
| URL: | |

4. Check the Export for ActionScript checkbox. This will automatically check the Export in First Frame checkbox as well. In the AS 3.0 Class textbox, enter CSNub_Submarine. Click Ok.

5. From now on, the code from the class CSNub_Submarine will control the submarine character on the screen. Every movie clip in the Library that is instantiated on the stage must be linked to a class file. More than one movie clip can be assigned to the same class file.

6. Test the movie to show that the class CSNub_Submarine is linked with the submarine graphic on the screen by seeing the submarine graphic move upwards. Examine the *move* method to see that it changes the y-coordinate value by -5.

7. We will practice assigning class files to another movie. Go to the Library and find the movie clip *squid1*. Select the squid and drag it the left of the submarine. Add another squid to the stage at the top of the screen in the path of the submarine. In the example below, we used *squid2*.



8. Set the Instance Name of the two squids to redsquid and bluesquid.

9. Add these two new objects to the object registry.

10. Link the class file CSNub_Squid to the movie clips squid1 and squid2. Open CSNub_Squid.as file to view the class file for all squid.

11. Save your work (File>Save).

**Activity 4: Controlling the Submarine**

Purpose: This activity shows you how to add keyboard input to control the submarine. It explains how the CSNub_EventHandler is used as the driver of the game. Use the CSNub_Template file you edited in the previous activity.

Objectives: Understanding the role of CSNub_EventHandler, using and processing keyboard input

1. Locate the CSNub_EventHandler, which is outside of the bounds of the upper lefthand corner of the stage.

2. CSNub_EventHandler is the main driver of the game. Thirty times per second the game is running, it will go through all objects in the *object_registry* and call their individual *move, keyboardInput,* and *intersection* methods. The *move* method is responsible for moving the character around the screen. It does not handle the keyboard input. At any time, should CSNub_EventHandler detect keyboard input or an intersection between the submarine and another character/item on the stage, the methods *keyboardInput* and *intersection* are called, respectively.

3. Open the file CSNub_Submarine.as.

4. Before adding controls to the submarine, we need to keep the submarine from moving off the screen when the game starts. Locate the *move* method and comment out the body.

5. Save the file CSNub_Submarine.as.

6. You can run the game to see that it the submarine does not move. Note that you will need to return to the CSNub_Template.fla file to run the game.

7. Return to the CSNub_Submarine.as file. The next step is to accept and process the keyboard input. If the CSNub_EventHandler detects that the user is holding down a key, it will pass that key's code to the *keyboardInput* method. For this activity, the *keyboardInput* method in CSNub_Submarine will only deal with the arrow keys (up, down, left, and right). This should make the submarine face the direction of the arrow. Rotating the submarine through its *_rotation* data member does this.

8. Enter the following code in the body of the keyboardInput method.

```
switch( keycode )
{
case Key.UP:
      this._rotation = 0;
```

```
            break;
    case Key.DOWN:
            this._rotation = 180;
            break;
    case Key.RIGHT:
            this._rotation = 90;
            break;
    case Key.LEFT:
            this._rotation = -90;
            break;
    default:
            break;
    }
```

The rotation values range from -180˚ to 180˚ with 0˚ being the original orientation of the movie clip. Save this file and test the game to see keyboard input in action.

9. Test your game to see that your submarine now accepts and responds to keyboard input.

10. The *move* method had a single line:

```
    this._y = this._y - 5;
```

Every character object has _y data member, which represents its y coordinate. The stage has its own x-y coordinate system with the origin (0,0) being in the upper left hand corner. The lower right hand corner is (1280,768). Recall that the *move* method is called 30 times per second, and so the y-coordinate decrease by 5 at each method call. See what happens when you change this equation to

```
    this._y = this._y + 5;
```

Experiment with different values as well as with the _x data member.

11. Replace the body of the *move* method with the below code. This code moves the submarine depending on which way it is facing (depending on the *_rotation* value as determined by the *keyboardInput* method.

```
    switch( this._rotation )
    {
    case 0:      // each value is in degrees
            this._y = this._y - 5;
            break;
    case 180:
            this._y = this._y + 5;
            break;
    case 90:
            this._x = this._x + 5;
            break;
      case -90:
            this._x = this._x - 5;
            break;
      default:
            break;

      }
```

12. Test your movie to see the submarine move in the direction it is facing.

13. Save your work (File>Save).

## Appendix C.2 – Clinical Interview Activity

For this activity, you will implement the interaction between the submarine and the other objects on the ocean floor.

1. Add a display panel to the stage. Remember to link the appropriate .as file. Give it the instance name *display_panel*.

2. Set the following values for the characters:

   Submarine: HP = 10, AP = 3, DP = 2
   Squid (both squids): HP = 5, AP = 3, DP = 1

3. Implement a rock encounter. Do this in the submarine's *intersection* method. Add rock to the stage. The movie clips for the rocks are in the *obstacles* folder in the Library. You will need to create the CSNub_Rock.as file and write the CSNub_Rock class. To create a new file, go to New… and select ActionScript File. Save as CSNub_Rock.as. When the submarine encounters a rock, it automatically turns around 180˚. Deduct 1 HP. Display a message saying that the submarine hit a rock and has lost an HP.

4. Implement an item collection sequence. Do this in the submarine's *intersection* method. When an item is collected, it is immediately used. Add an energy barrel to the stage. The movie clips for the energy barrel are in the *items* folder in the Library. You will need to create the CSNub_EnergyBarrel.as file and write the CSNub_EnergyBarrel class.

   Set the following values for the barrel.

   Energy barrel: points = 3

   The character property affected will be the hitpoints. When the submarine encounters an energy barrel, it gains 3 HP.

5. Implement a fight sequence between the submarine and a squid. Do this in the submarine's *intersection* method. The submarine always attacks first. Use the following equations to calculate the damage to each character

   Damage to squid:
   If $AP_{submarine} > DP_{squid}$ Then
   $HP_{squid} = HP_{squid} - ( AP_{submarine} - DP_{squid} )$,
   Else

```
        No damage to squid

Damage to submarine:
    If $AP_{squid} > DP_{submarine}$ Then
        $HP_{submarine} = HP_{submarine} - (AP_{squid} - DP_{submarine})$
    Else
        No damage to submarine
```

Each character takes turns to attack. Display each character's status in the display panel with each attack. The fight continues until one character has an HP of 0. When the battle is over, display a message saying if the submarine won or not. If the submarine wins, make the enemy disappear by setting it invisible. If the submarine loses, CSNüb will automatically take the user to the Game Over screen.

## Appendix C.3 – Extension Questions

These questions are after the clinical interview activity is over and are an extension of the previous activity. They are similar to tasks covered in the clinical activity interview, but are slightly more difficult. Here, participants are asked to conceptualize their answers rather than to implement them.

1. (Inheritance) How would you design a new object that represents a group of characters (e.g. a group of enemies-each taking turns to attack the submarine in battle)? How would you incorporate this into the class hierarchy?

2. (Encapsulation) How would you change the code such that each enemy character can carry one item? When it is defeated, the submarine immediately collects the item and uses it. How would you implement this fight sequence into the intersection?

## Appendix D.1– Behavioral Protocol Log Review

**Exploration**

Definition: The process of seeking out other resources within CSNüb

Subcategories: lookup, inheritance, planning

**Disequilibrium, Moments of**

Definition: When something does not match with the students' mental model or understanding and includes resolving the problem

Subcategories: feedback, assimilation, syntax, redundant code, superfluous code

**Awareness**

Definition: The scope or range of what students can see

Subcategories, high-level, low-level, bigger picture, higher-order thinking, visibility

**Scaffolding**

Definition: Resources that support conceptual understanding or learning

Subcategories: resources, reuse, assimilation

**Refinement**

Definition: The process of trying to make their solution better or more correct

Subcategories: debugging, efficiency, consistency, experimentation, testing

**None of the Above/Unknown**

## Appendix D.2 – Source Code Review

### CSNüb Hierarchy of Classes

```
CSNub_Object
          |-----> CSNub_Character
                              |-----> CSNub_Submarine
                              |-----> CSNub_Squid
                              |-----> CSNub_Eel
                              |-----> CSNub_Ogrefish
                              |-----> CSNub_Shark
          |-----> CSNub_Item
                              |-----> CSNub_EnergyBarrel
                              |-----> CSNub_RadioactiveBarrel
          |-----> CSNub_Obstacle
                              |-----> CSNub_Rock
                              |-----> CSNub_Kelp
          |-----> CSNub_GameObject
                              |-----> CSNub_DisplayPanel
                              |-----> CSNub_EventHander
CSNub_ObjectRegistry
```

### Differences between Java/C++ and ActionScript 3.0

Though the syntax between Java/C++ and ActionScript 3.0 are very similar, there are several differences between the two languages that are important for CSNüb.

| **Java/C++** | **ActionScript 3.0** |
|---|---|
| float x; *or*<br>double x; *or* | var x:Number; |
| bool x; // in C++ *or*<br>boolean x; // in Java | var x:Boolean; |
| String x; | var x:String; |
| float x = 15; | var x:Number = 15; |
| int x = (int) 3.14159; | var x:int = int( 3.14159 ); |
| public Constructor()<br>{<br>} | public function Constructor()<br>{<br>} |

| | |
|---|---|
| `public void fubar( int x )`<br>`{`<br>`        // do nothing`<br>`}` | `public function fubar( x:Number ):Void`<br>`{`<br>`        // do nothing`<br>`}` |
| `System.out.println( "Hello!" );` | `trace( "Hello!" );` |

**PTA Rubric**

**Class Design**
3 – Students design appropriate classes and make appropriate decisions about the use of composition and inheritance.
2 – Most design decisions are appropriate.
1 – Several design decisions are inappropriate.

**Documentation**
3 – The program contains a comment for each public class and for each public member of a public class. The comments are correct and unambiguous, explaining "what" not "how". Spelling and grammar are correct.
2 – Most necessary comments are present, correct, and unambiguous.
1 – Several comments are missing or wrong.

**Correctness**
3 – The program implements all required features. The program behaves correctly for both typical and unusual (but correct) input. The program also handles bad input appropriately.
2 – The program fails to implement some minor feature in the specification. The program behaves correctly for all typical input and most unusual input.
1 – The program does not behave correctly for some typical input or fails to implement a major feature or two or more minor features in the specification.

**Re-use**
2 – The program makes appropriate use of the CSNüb's predefined classes and object-oriented framework to create new classes
1 – The program re-implements classes, variables, and/or methods available in the CSNüb template or the program uses CSNüb classes incorrectly.

## Appendix E – Participant Consent Form

*IRB#* 2007-04-0100

**INFORMED CONSENT TO PARTICIPATE IN RESEARCH**
**The University of Texas at Austin**

You are being asked to participate in a research study. This form provides you with information about the study. The Principal Investigator (the person in charge of this research) or his/her representative will provide you with a copy of this form to keep for your reference, and will also describe this study to you and answer all of your questions. Please read the information below and ask questions about anything you don't understand before deciding whether or not to take part. Your participation is entirely voluntary and you can refuse to participate without penalty or loss of benefits to which you are otherwise entitled.

**Title of Research Study:** Using a multimedia cognitive tool to facilitate novices' conceptual understanding of object-oriented programming

**Principal Investigator:** Timothy T. Yuen, Graduate Student, 949-533-4508
**Faculty Sponsor:** Min Liu, Ed.D., Associate Professor, 512-471-521

**Funding source:** Not applicable

**What is the purpose of this study?** The purpose of this study is to examine how CSNüb, a tool based on the design principles for multimedia cognitive tools, helps novice computer science students understand and comprehend the fundamentals of object-oriented programming. CSNüb is a template written in Adobe Flash for a simple role-playing game. Twenty (20) participants are sought for this study.

**What will be done if you take part in this research study?** There are two rounds to participation. The first round involves an information/tutorial session in which CSNüb and the Adobe Flash environment are introduced. Consent forms and demographic data will be collected at this time. The second round is done individually with the researcher and involves solving a problem using the CSNüb template. You will be asked to explore and reflect on your understanding of object-oriented programming and your solution to the problem. All sessions are videotaped and computer actions recorded, but filming will be done from behind the participants.

**The Project Duration:** The first round lasts 2 hours. The second round is expected to last 4 hours and occurs within a few days to three weeks of the first phase. All efforts will be made to find times that are most accommodating to each participant's schedule.

**What are the possible discomforts and risks?** In the first round, you may feel discomfort associated with attending a discussion section or small lecture. In the second round, you may experience possible discomforts associated with test anxiety, as you will be asked to demonstrate your knowledge of object-oriented programming while trying to solve a programming-based problem. Additionally, all sessions are videotaped, but only the backs of the students are filmed. If you wish to discuss further what participation entails, please ask questions now or call the Principal Investigator listed on the front page.

**What are the possible benefits to you or to others?** Possible benefits of participating in this study to you include having a better understanding of object-oriented programming and experience with an interactive multimedia authoring tool like Adobe Flash and its ActionScript language. Others that may benefit from this study include the computer science education community, designers of instructional tools, and the educational technology research.

**If you choose to take part in this study, will it cost you anything?** There are no costs to participate in this study.

**Will you receive compensation for your participation in this study?** Each participant will be paid $40 at the end of his or her individual interview.

**What if you are injured because of the study?** If injuries occur as a result of study activity, eligible University students may be treated at the usual level of care with the usual cost for services at the Student Health Center, but the University has no policy to provide payment in the event of a medical problem.

**If you do not want to take part in this study, what other options are available to you?** Your participation in this study is entirely voluntary. You are free to refuse to be in the study, and your refusal will not influence current or future relationships with The University of Texas at Austin.

**How can you withdraw from this research study and who should you call if you have questions?**

**If you wish to stop your participation in this research study for any reason, you should contact the principal investigator:** Timothy Yuen **at** (949) 533-4508. **You should also call the principal investigator for any questions, concerns, or complaints about the research. You are free to withdraw your consent and stop participation in this research study at any time without penalty or loss of benefits for which you may be entitled. Throughout the study, the researchers will notify you of new**

**information that may become available and that might affect your decision to remain in the study.**

**In addition, if you have questions about your rights as a research participant, or if you have complaints, concerns, or questions about the research, please contact Jody Jensen, Ph.D., Chair, The University of Texas at Austin Institutional Review Board for the Protection of Human Subjects at (512) 232-2685 or the Office of Research Support and Compliance at (512) 471-8871.**

How will your privacy and the confidentiality of your research records be protected? Each participant will be given an ID number as a pseudonym. All data gathered for this study is kept confidential and will be secured in a private location. Please note: (a) that the interviews or sessions will be audio or videotaped; (b) that the cassettes will be coded so that no personally identifying information is visible on them; (c) that they will be kept in a secure place (e.g., a locked file cabinet in the investigator's office); (d) that they will be heard or viewed only for research purposes by the investigator and his or her associates; and (e) that they will be erased after they are transcribed or coded. If the results of this research are published or presented at scientific meetings, your identity will not be disclosed.

**If in the unlikely event it becomes necessary for the Institutional Review Board to review your research records, then The University of Texas at Austin will protect the confidentiality of those records to the extent permitted by law. Your research records will not be released without your consent unless required by law or a court order. The data resulting from your participation may be made available to other researchers in the future for research purposes not detailed within this consent form. In these cases, the data will contain no identifying information that could associate you with it, or with your participation in any study.**

**Will the researchers benefit from your participation in this study*?*** The researcher will not receive any benefits from your participation in this study beyond publishing and presenting at conferences.

**Signatures:**

**As a representative of this study, I have explained the purpose, the procedures, the benefits, and the risks that are involved in this research study:**

| | |
|---|---|
| **Signature and printed name of person obtaining consent** | **Date** |

**You have been informed about this study's purpose, procedures, possible benefits and risks, and you have received a copy of this form. You have been given the opportunity to ask questions before you sign, and you have been told that you can ask other questions at any time. You voluntarily agree to participate in this study. By signing this form, you are not waiving any of your legal rights.**

| | |
|---|---|
| **Printed Name of Subject** | **Date** |
| **Signature of Subject** | **Date** |
| **Signature of Principal Investigator** | **Date** |

## References

ACM-SIGCSE. (2001). Computing curricula 2001. from http://www.sigcse.org/cc2001/

Aly, M., Elen, J., & Willems, G. (2005). Learner-control vs. program-control instructional multimedia: A comparison of two interactions when teaching principles of orthodontic appliances. *European Journal of Dental Education, 9*, 157-163.

Aslop, S., & Watts, M. (2003). Science education and affect. *International Journal of Science Education, 25*(9), 1043-1047.

Baddeley, A. (1992). Working memory. *Science, 255*(5044), 556-559.

Baddeley, A. (2001). Is working memory still working? *American Psychologist, 56*, 851-864.

Bandura, A. (1994). Self-efficacy. In V. S. Ramachaudran (Ed.), *Encyclopedia of human behavior* (Vol. 4, pp. 71-81). New York, NY: Academic Press.

Barnes, D. J. (2002). *Teaching introductory Java through LEGO MINDSTORMS models*. Paper presented at the SIGCSE Technical Symposium on Computer Science Education, Covington, KY.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: Some thoughts and observations. *inroads - The SIGCSE Bulletin, 37*(2), 103-106.

Ben-Ari, M. (1996). *Understanding programming languages*. Chichester, West Sussex: John Wiley & Sons.

Bennedsen, J., & Caspersen, M. E. (2006). Abstraction ability as an indicator of success of learning object-oriented programming? *inroads - The SIGCSE Bulletin, 38*(2), 39-43.

Bennedsen, J., & Caspersen, M. E. (2007). Failure rates in introductory programming. *inroads - The SIGCSE Bulletin, 39*(2), 33-36.

Bickhard, M. (2005). Functional scaffolding and self-scaffolding. *New Ideas in Psychology, 23*(3), 166 - 173.

Boyle, T., Bradley, C., Chalk, P., Jones, R., & Pickard, P. (2003). Using blended learning to improve student success rates in learning to program. *Journal of Educational Media, 28*(2-3).

Brooks, J. G., & Brooks, M. G. (1993). *In search of understanding the case for constructivist classrooms*. Alexandria, VA: Association for Supervision and Curriculum Development.

Bruce, K. B. (2005). Controversy on how to teach CS 1: Discussion on the SIGCSE-members mailing list. *inroads - The SIGCSE Bulletin, 37*(2), 111-117.

Brusilovsky, P., Grady, J., Spring, M., & Lee, C.-H. (2006). What should be visualized? Faculty perception of priority topics for program visualization. *inroads - The SIGCSE Bulletin, 38*(2), 44-48.

Cable, A. M. (2001). Classroom-based assessment in an object-oriented programming course. *Journal Computing Sciences in Colleges, 17*(1), 259-264.

Chen, W.-K., & Cheng, Y. C. (2007). Teaching object-oriented programming laboratory with computer game programming. *IEEE Transactions on Education, 50*(3), 197-203.

Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement, 20*, 37 - 46.

Collins, A., Joseph, D., & Bielaczyc, K. (2004). Design research: Theoretical and methodological issues. *The Journal of Learning Sciences, 13*(1), 15-42.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., Christiansen, K., Deline, R., et al. (2000). Alice: Lessons learned from building a 3D system for novices. *CHI Letters, 2*(1), 486-493.

Cooper, S., Dann, W., & Pausch, R. (2000). *Alice: A 3-D tool for introductory programming concepts*. Paper presented at the CCSC '00: Proceedings of the fifth annual CCSC northeastern conference on The journal of computing in small colleges, Mahwah, NJ.

Crawford, S. (2006). ActionScript: A gentle introduction to programming. *Journal Computing Sciences in Colleges, 21*(3), 156 - 168.

Cross II, J. H., Hendrix, D., Jain, J., & Barowski, L. A. (2007). *Dynamic object viewers for data structures*. Paper presented at the SIGCSE 2007 Technical Symposium on Computer Science Education, Covington, KY.

Crotty, M. (2003). *The foundations of social research*. London: SAGE Publications, Inc.

Csikszentmihalyi, M. (1990). *Flow: The psychology of optimal experience*. New York, NY: Harper Collins Publishing.

Dale, N. B. (2006). Most difficult topics in CS1: Results of an online survey of educators. *inroads - The SIGCSE Bulletin, 38*(2), 46-53.

Deci, E. L., & Ryan, R. M. (1993). The initiation and regulation of intrinsically motivated learning and achievement. In T. S. Pittman (Ed.), *Achievement and Motivation: A Social-Developmental Perspective*. Cambridge: Cambridge University Press.

Derry, S. J. (1996). Cognitive schema theory in the constructivist debate. *Educational Psychologist, 31*(3/4), 163-174.

Derry, S. J., & Lajoie, S. P. (1993). A middle camp for (un)intelligent instructional computing: An introduction. In S. J. Derry (Ed.), *Computers as cognitive tools*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Dey, I. (1999). *Grounding grounded theory: Guidelines for qualitative inquiry*. San Diego, CA: Academic Press.

Doube, W. (2004). *Multimedia delivery of computer programming subjects: Basing structure on instructional design*. Paper presented at the ACM International Conference Proceeding Series, The University of Queensland, Australia.

Driscoll, M. P. (2004). *Psychology of learning for instruction*. Boston, MA: Allyn & Bacon.

Eckerdal, A. (2006). *Novice students' learning of object-oriented programming*. Uppsala University.

Eckerdal, A., & Berglund, A. (2005). *What does it take to learn 'programming thinking'?* Paper presented at the ICER '05, Seattle, Washington, USA.

Ferguson, E. (2003). Object-oriented concept mapping using UML class diagrams. *Journal of Computing in Small Colleges, 18*(14), 344-354.

Fontana, A., & Frey, J. (2005). The interview: From neutral stance to political involvement. In N. Denzin & Y. Lincoln (Eds.), *The SAGE handbook of qualitative research* (3rd ed.).

Forte, A., & Guzdial, M. (2005). Motivation and nonmajors in computer science: Identifying discrete audiences for introductory courses. *IEEE Transactions on Education, 48*(2), 248-253.

Frost, D. (2008). Ucigame, a Java library for games. *ACM SIGCSE Bulletin, 40*(1).

Ginsburg, H. P. (1997). *Entering the child's mind: The clinical interview in psychological research and practice*. New York, NY: Cambridge University Press.

Ginsburg, H. P., & Opper, S. (1987). *Piaget's theory of intellectual development* (3rd ed.). Englewood Cliffs, NJ: Prentice Hall.

Glaser, B. (2001). *The grounded theory perspective: Conceptualization contrasted with description*. Mill Valley, CA: Sociology Press.

Gorard, S., Roberts, K., & Taylor, C. (2004). What kind of creature is a design experiment? *Educational Research Journal, 30*(4), 577-590.

Greeno, J., Collins, A., & Resnick, L. (1996). Cognition and learning. In D. C. Berliner & R. C. Calfee (Eds.), *Handbook of educational psychology* (pp. 15-46). New York, NY: Simon & Schuster Macmillan.

Guzdial, M., & Soloway, E. (2002). Teaching the Nintendo generation to program. *Communications of the ACM, 45*(4), 17-21.

Hadjerrouit, S. (1999). *A constructivist approach to object-oriented design and programming*. Paper presented at the iTiCSE '99, Cracow, Poland.

Hartley, J. (2004). Designing instructional and informational text. In D. H. Jonassen (Ed.), *Handbook of research on educational communications and technology* (pp. 917-945). Mahwah, NJ: Lawrence Erlbaum Associates.

Hayes, J. R., & Flower, L. S. (1980). Identifying the organization of writing process. In E. R. Steinberg (Ed.), *Cognitive process in writing* (pp. 3-30). Hillsdale, NJ: Erlbaum.

Hayes, J. R., & Flower, L. S. (1983). Uncovering cognitive processes in writing: An introduction to protocol analysis. In S. A. Walmsley (Ed.), *Research on writing: principles and methods*. New York, NY: Longman.

Henriksen, P. (2004). *A direct interaction tool for software engineering education*. University of Southern Denmark.

Henriksen, P., & Kölling, M. (2004). *greenfoot: Combining object visualization with interaction*. Paper presented at the OOOPSLA '04, Vancouver, BC.

Herrmann, N., Popyack, J., Char, B., Zoski, P., Cera, C., & Lass, R. (2003). *Redesigning introductory computer programming using multi-level online modules for a mixed audience*. Paper presented at the Technical Symposium on Computer Science Education, Reno, Nevada, USA.

Hood, C. S., & Hood, D. J. (2005). *Teaching programming and language concepts using LEGOs®*. Paper presented at the Annual Joint Conference Integrating Technology into Computer Science Education, Caparica, Portugal.

Jehng, S.-C. J., Tung, S.-H. S., & Chang, C.-T. (1999). A visualisation approach to learning the concept of recursion. *Journal of Computer Assisted Learning, 15*.

Jonassen, D. H. (1991). What are cognitive tools? In J. T. Mayes (Ed.), *Cognitive tools for learning*. Berlin: Springer-Verlag.

Jonassen, D. H. (2000). *Computers as mindtools for schools*. Upper Saddle River, NJ: Merrill.

Jonassen, D. H., Howland, J., Moore, J., & Marra, R. M. (2003). *Learning to solve problems with technology: A constructivist perspective*. Upper Saddle River, NJ: Merrill Prentice Hall.

Jones, A. (2003a). *Grand research challenges in information systems*: Computing Research Associates.

Jones, S. (2003b). *Let the games begin: Gaming technology and entertainment among college students*. Washington, DC: Pew Internet and American Life Project.

Kölling, M. (1999). The problem of teaching object-oriented programming, Part 1: Languages. *Journal of Object-Oriented Programming, 11*(8), 8-15.

Kölling, M., & Rosenberg, J. (1996). *An object-oriented program development environment for the first programming course*. Paper presented at the SIGCSE Technical Symposium on Computer Science Education, Philadelphia, PA.

Krathwohl, D. R., Bloom, B. S., & Masia, B. B. (1965). *Taxonomy of educational objective-The classification of education goals: Handbook II: Affective domain.* New York, NY: David McKay Company, Inc.

Lahtinen, E., Ala-Mutka, K., & Jarvinen, H.-M. (2005). A study of the difficulties of novice programmers. *inroads - The SIGCSE Bulletin, 37*(3), 14-18.

Lajoie, S. P. (1993). Computer environments as cognitive tools for enhancing learning. In S. J. Derry (Ed.), *Computers as cognitive tools*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Lajoie, S. P. (Ed.). (2000). *Computers as cognitive tools: No more walls* (Vol. 2). Mahwah, NJ: Lawrence Erlbaum Associates.

Lancy, D. F. (1993). *Qualitative research in education: An introduction to the major traditions*. New York, NY: Longman Pub Group.

Landis, J., & Koch, G. (1977). The measurement of observer agreement for categorical data. *Biometrics, 33*(1), 159 - 174.

Lave, J., & Wenger, E. (1991). *Situated learning*. Cambridge, UK: Cambridge University Press.

Lawhead, P. B., Bland, C. G., Barnes, D. J., Duncan, M. E., Goldweber, M., Hollingsworth, R. G., et al. (2003). A road map for teaching introductory programming using LEGO(c) Mindstorms robots. *inroads - The SIGCSE Bulletin, 35*(2), 191-201.

Lemos, R. S. (1979). *Teaching programming languages: A survey of approaches*. Paper presented at the Technical Symposium on Computer Science Education.

Lenhart, A., Kahne, J., Middaugh, E., Macgill, A. R., Evans, C., & Vitak, J. (2008). *Teens, video games, and civics: Teens' gaming experiences are diverse and include significant social interaction and civic engagement*. Washington, DC: Pew Internet & American Life Project.

Lepper, M. R., & Malone, T. W. (1987). Intrinsic motivation and instructional effectiveness in computer-based education. In R. E. Snow & M. J. Farr (Eds.), *Aptitude, learning and instruction* (Vol. 3. Cognitive and Affective Process Analysis, pp. 255-287). Hillsdale, NJ: Lawrence Erlbaum Associates.

Lepper, M. R., Woolverton, M., Mumme, D. L., & Gurtner, J.-L. (1993). Motivational techniques of expert tutors: Lessons for the design of computer-based tutors. In S. J. Derry (Ed.), *Computers as cognitive tools* (pp. 75-106). Hillsdale, NJ: Lawrence Erlbaum Associates.

Leutenegger, S., & Edgington, J. (2007). A games first approach to teaching introductory programming. *ACM SIGCSE Bulletin, 39*(1), 115 - 118.

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic inquiry*. Newbury Park, CA: SAGE Publications.

Lister, R., Berglund, A., Clear, T., Bergin, J., Garvin-Doxas, K., Hanks, B., et al. (2006). *Research perspectives on the objects-early debate*. Paper presented at the ITiCSE.

Liu, M., Toprac, P., & Yuen, T. (2008). What factors make a multimedia learning environment engaging: A case study. In R. Zheng (Ed.), *Cognitive effects of multimedia learning*. Hershey, PA: Idea Group Inc.

Lockee, B., Moore, D., & Burton, J. (2004). Foundations of programmed instruction. In D. H. Jonassen (Ed.), *Handbook of research on educational communications and technology* (pp. 545-565). Mahwah, NJ: Lawrence Erlbaum Associates.

Malone, T. W. (1981). Toward a theory of intrinsically motivating instruction. *Cognitive Science, 4*, 333-369.

Malone, T. W., & Lepper, M. R. (1987). Making learning fun: A taxonomy of intrinsic motivations for learning. In R. E. Snow & M. J. Farr (Eds.), *Aptitude, learning and instruction* (Vol. 3: Cognitive and affective process analysis, pp. 223-253). Hillsdale, NJ: Lawrence Erlbaum Associates.

Marlowe, B., & Page, M. (2005). *Creating and sustaining the constructivist classroom*. Thousand Oaks, CA: Corwin Press.

Mayer, R. E. (1992). Cognition and instruction: Their historic meeting within educational psychology. *Journal of Educational Psychology, 84*(4).

Mayer, R. E. (2001). *Multimedia learning*. Cambridge: Cambridge University Press.

Mayer, R. E. (2003). Theories of learning and their application to technology. In R. S. Perez (Ed.), *Technology applications in education: A learning view* (pp. 127-157). Mahwah, NJ: Lawrence Erlbaum and Associates.

Mayer, R. E., & Moreno, R. (1998). *A cognitive theory of multimedia learning: implications for design principles*. Paper presented at the ACM SIGCHI Conference on Human Factors in Computing Systems, Los Angeles, CA.

McGettrick, A., Boyle, R., Ibbett, R., Lloyd, J., Lovegrove, G., & Mander, K. (2004). *Grand challenges in computing education*: The British Computer Society.

McKinney, D., & Denton, L. F. (2004). *Houston, we have a problem: There's a leak in the CS1 affective oxygen tank*. Paper presented at the 35th SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA.

McNally, M., Goldweber, M., Fagin, B., & Klassner, B. (2006). *Do Lego MindStorms robots have a future in CS education?* Paper presented at the SIGCSE Technical Symposium on Computer Science Education, Houston, TX.

Mertens, D. (1998). *Research methods in education and psychology*. Thousand Oaks, CA: SAGE Publications.

Moore, D., Burton, J. K., & Myers, R. J. (2004). Multiple-channel communication: The theoretical and research foundations of multimedia. In D. H. Jonassen (Ed.), *Handbook of research on educational communications and technology* (pp. 979-1005). Mahwah, NJ: Lawrence Erlbaum Associates.

Moreno, A., Myller, N., & Bednarik, R. (2005). *Jeliot 3, an extensible tool for program visualization*. Paper presented at the Koli Calling 2005: 5th Annual Finnish / Baltic Sea Conference on Computer Science Education.

Moreno, R., & Mayer, R. E. (2000). A Learner-Centered Approach to Multimedia Explanations: Deriving Instructional Design Principles from Cognitive Theory. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, 2*(2).

Moreno, R., & Valdez, A. (2005). Cognitive load and learning effects of having students organize pictures and words in multimedia environments: The role of student interactivity and feedback. *Education Technology Research and Development, 53*(3), 35-45.

Moses, L. (2006). *Animation programming: an alternative approach to CS1*. Paper presented at the ITiCSE '06, Bologna, Italy.

Moskal, B., Lurie, D., & Cooper, S. (2004). *Evaluating the effectiveness of a new instructional approach*. Paper presented at the SIGCSE, Norfolk, VA.

Paivio, A., Walsh, M., & Bons, T. (1994). Concreteness effects on memory: When and why? *Journal of Experimental Psychology: Learning, Memory, and Cognition, 20*(5), 1196-1204.

Papert, S. (1991). Situating constructionism. In I. Harel & S. Papert (Eds.), *Constructionism*. Norwood, NJ: Ablex Publishing.

Pausch, R. (2008). *Alice: A dying man's passion*. Paper presented at the SIGCSE '08, Portland, OR.

Pea, R. D. (1985). Beyond amplification: Using the computer to reorganize mental functioning. *Educational Psychologist, 20*(4), 167-182.

Pellegrino, J. W., Chudowsky, N., & Glaser, R. (2001). *Knowing what students know: The science and design of educational assessment*. Washington, DC: National Academy Press.

Piaget, J. (1952). *The origins of intelligence in children* (M. Cook, Trans.). New York, NY: W.W. Norton & Company, Inc.

Powers, K., Ecott, S., & Hirshfield, L. M. (2007). *Through the looking glass: Teaching CS0 with Alice*. Paper presented at the SIGCSE Technical Symposium on Computer Science Education, Covington, KY.

Pratt, T., W., & Zelkowitz, M. V. (1996). *Programming languages: Design and implementation*. Englewood Cliffs, NJ: Prentice-Hall, Inc.

Renninger, K. A. (2000). How might the development of individual interest contribute to the conceptualization of intrinsic motivation. In *Intrinsic motivation: Controversies and new directions*.

Richardson, L. (2005). *Handling qualitative data: A practical guide*. London: SAGE Publications.

Salomon, G., & Globerson, T. (1987). Skill may not be enough: the role of mindfulness in learning and transfer. *International Journal of Educational Research, 11*(6), 623-637.

Schiefele, U. (1991). Interest, learning, and motivation. *Educational Psychologist, 26*(3 & 4), 299-323.

Sebesta, R. W. (1999). *Concepts of programming languages* (4th ed.). Reading, MA: Addison Wesley.

Sicilia, M.-Á. (2006). Strategies for teaching object-oriented concepts with Java. *Computer Science Education, 16*(1), 1-18.

Smith, D. N. (1991). *Concepts of object-oriented programming*. New York, NY: McGraw-Hill, Inc.

Smith, J. P., diSessa, A. A., & Roschelle, J. (1993). Misconceptions reconceived: A constructivist analysis of knowledge in transition. *The Journal of the Learning Sciences, 3*(3), 115-163.

Stein, L. A. (1998). What we swept under the rug: Radically rethinking CS1. *Computer Science Education, 8*(2), 118-129.

Strauss, A., & Corbin, J. (1998). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Thousand Oaks, CA: SAGE Publications, Inc.

Sung, K., Shirley, P., & Rosenberg, B. R. (2008). *Experience aspects of game programming in an introductory computer graphics class*. Paper presented at the SIGCSE '07, Covington, KY.

Tabbers, H. K., Marens, R. L., & van Merriënboer, J. J. G. (2004). Multimedia instructions and cognitive load theory: Effects of modality and cueing. *British Journal of Educational Psychology, 74*, 71-81.

Tennant, M., & Pogson, P. (1985). *Learning and change in the adult years: A developmental perspective*. San Francisco, CA: Jossey-Bass Publishers.

Thomasson, B., Ratcliffe, M., & Thomas, L. (2006). Identifying novice difficulties in object oriented design. *inroads - The SIGCSE Bulletin, 38*(3), 28-32.

van Haaster, K., & Hagan, D. (2004). *Teaching and learning with BlueJ: An evaluation of a pedagogical tool*. Paper presented at the Information Science + Information Technology Education Joint Conference, Rockhampton, QLD, Australia.

von Glasersfeld, E. (1984). An introduction to radical constructivism. In P. Watzlawick (Ed.), *The invented reality: How do we know what we believe we know? Contributions to constructivism*. New York, NY: W.W. Norton & Company.

von Glasersfeld, E. (1987). Learning as constructive activity. In C. Janvier (Ed.), *Problems of representation in the teaching and learning of mathematics*. Hillsdale, NJ: Lawrence Erlbaum Associates.

von Glasersfeld, E. (2005). Introduction: Aspects of constructivism. In C. T. Fostnot (Ed.), *Constructivism: Theory, perspectives, and practice* (2nd ed.). New York, NY: Teachers College Press.

Vygotsky, L. (1978). *Mind in society: The development of higher psychological processes*. Cambridge, MA: Harvard University Press.

Wells, G. (1999). *Dialogic inquiry: Towards a sociocultural practice and theory of education*. New York, NY: Cambridge University Press.

Wells, G. (2000). Dialogic inquiry in education: Building on the legacy of Vygotsky. In P. Smagorinsky (Ed.), *Vygotskian perspectives on literacy research: Constructing meaning through collaborative inquiry*. Cambridge: Cambridge University Press.

Winn, W. (2003). Cognitive perspectives in psychology. In D. H. Jonassen (Ed.), *Handbook of research on educational communications and technology* (pp. 79-112). Mahwah, NJ: Lawrence Erlbaum Associates.

Woody, R. H. (2001). Learning from the experts. In *MENC* (pp. 9-14).

Yuen, T. (2007a). *Clinical interviews and process-tracing methods in computer science education research*. Paper presented at the SIGCSE Technical Symposium on Computer Science Education, Covington , KY.

Yuen, T. (2007b). Novices' Knowledge Construction of Difficult Concepts in CS1. *inroads - The SIGCSE Bulletin, 39*(4).

## Vita – Timothy T. Yuen

Timothy T. Yuen was born on June 22, 1978, in Los Angeles, CA, to Man Wai and Wing Hung Yuen. Tim grew up in the city of Rosemead, CA, approximately 12 miles east of downtown Los Angeles. Tim has been a student since preschool, and did not stop his formal schooling until the completion of his PhD. Destined to be a software developer, Tim pursued degrees in computer science. He received a BS in Information and Computer Science from the University of California, Irvine, and an MS in Computer Science at the University of Southern California. That career plan changed, however, after he was bitten by the "teaching bug" when he was a TA for computer science classes at UC Irvine; his interest in teaching was furthered by his experience as an instructor and curriculum developer at iD Tech Camps. Tim's research interests are in computer science education, multimedia-based cognitive tools, and design-based research. He has presented his research at academic conferences such as ACM SIGCSE, AERA, EDMEDIA, NECC, and SITE. Tim has held a graduate research assistant position at the Vaughn Gross Center for Reading and Language Arts where he served as a software developer for web-based applications that facilitate online communities of practice and learning and information systems for teacher professional development. Tim has also held software development positions Edvance, Inc. and Enspire Learning, where he developed online learning management systems and online interactive multimedia courses.

Permanent address: 5400 W Parmer Ln #1334, Austin, TX 78727

This dissertation was typed by Timothy T. Yuen.