# An ACM 2013 Exemplar Course Integrating Fundamentals, Languages, and Software Engineering

Jason O. Hallstrom, Cathy Hochrine, Jacob Sorber, and Murali Sitaraman

**Technical Report RSRG-13-08**
School of Computing
100 McAdams
Clemson University
Clemson, SC 29634-0974 USA

September 2013

# An ACM 2013 Exemplar Course Integrating Fundamentals, Languages, and Software Engineering

Jason O. Hallstrom, Cathy Hochrine, Jacob Sorber, Murali Sitaraman
Clemson University, School of Computing
Clemson, SC, 29634
{ jasonoh, chochri, jsorber, murali }@clemson.edu

## ABSTRACT
This paper summarizes our experiences in integrating topics in software development fundamentals (SDF), programming languages (PL), and software engineering (SE) knowledge areas of the ACM 2013 curriculum in a single course. It is novel in combining object-oriented programming and software development practices with fundamental analytical reasoning of software correctness. The aim is to integrate and cover the topics in an effective fashion. The course description in this paper represents an approach we have applied successfully for over 5 years. Students tend to consider this course to be one of the more challenging encountered in the first two years of study. Interestingly, the challenge appears to stem equally from mastering object-oriented programming and design pattern components of the course, as it does from learning to use specifications for analytical reasoning of component correctness.

## Categories and Subject Descriptors
K.3.2 [**Computer and Education**]: Computer and Information Science Education – *Computer Science education, Curriculum*

## General Terms
Design, Documentation, Experimentation, Languages, Verification

## Keywords
ACM curriculum, components, design patterns, formal reasoning, object-oriented programming, software engineering

## 1. INTRODUCTION
This paper summarizes our experiences in teaching a course that combines multiple knowledge areas within the ACM 2013 curriculum [1]. As the field of computing continues to evolve, such combinations will become ever more necessary to impart requisite skills within a reduced number of required courses for CS majors. We hope that this effort will serve as a case study for other educators in developing courses spanning multiple areas, as well as a specific course outline useful to educators interested in developing courses that combine fundamental and practical aspects of software development.

Beyond introducing students to the basics of software development, the sophomore-level course, CpSc 215 at Clemson, integrates three major themes: object-oriented programming (in Java), analytical reasoning, and software design patterns. The emphasis on object-oriented programming is motivated by the importance of this paradigm in modern software practice. Analytical reasoning along with math thinking and computational thinking, are among topics that have received regular SIGCSE attention (e.g., [2][3]). The emphasis on analytical reasoning is motivated by the supposition that the next generation of software engineers must be able to reason rigorously about the functionality and performance of the software they develop and maintain. Traditional trial-and-error methods of software development alone are insufficient. The emphasis on software design patterns is motivated by the wide adoption of patterns in modern software practice, both as prescriptive and descriptive aids. Together the topics help students learn the essential elements of building the next generation of high-quality software systems.

The course is taught using a "hands-on", collaborative approach [4], motivated by the success of peer instruction and other active learning strategies in mathematics and science. While the students consider the course among the most challenging, feedback suggests that the approach has had a positive impact on students' performance, self-efficacy, perception of computing as a discipline of study, as well as their attitudes [5].

**Paper organization.** Section 2 summarizes the ACM 2013 curriculum knowledge areas covered in the course, as well as the major topics addressed. Section 3 presents the key software development foundations, programming languages, and software engineering principles taught in the course. Section 4 shows that student performance across the topic areas is uniformly acceptable. Section 5 presents related work and conclusions.

## 2. COURSE SUMMARY

### 2.1 Body of Knowledge Coverage
This section summarizes the topics covered from different knowledge areas prescribed by the ACM 2013 curriculum. Note that only lecture hours are listed below. Of the 42 hours listed, there is some overlap among the hours devoted to knowledge area (KA) topics in software development foundations (SDF), programming languages (PL), and software engineering (SE). The actual coverage of the topics spans about 39 hours. The course covers a few additional background topics (3 hours) and includes a mid-term exam (1.5 hours).

**Table 1: Coverage of Knowledge Area Topics**

| KA | Knowledge Unit | Topics Covered | Hours |
|---|---|---|---|
| SDF | Algorithm Design | Fundamental design concepts and principles | 3 |
| SDF | Fundamental Data Structures | Abstract data types and their implementations, references and aliasing, linked lists, strategies for choosing the appropriate data structure | 9 |
| SDF | Development Methods | Program correctness, modern programming environments, | 6 |

| | | | |
|---|---|---|---|
| | | debugging strategies | |
| PL | Object-Oriented Programming | Object-oriented design, definition of classes: fields, methods, and constructors, subclasses, inheritance, and method overriding, dynamic dispatch: definition of method-call, subtyping, object-oriented idioms for encapsulation, using collection classes, iterators, and other common library components | 9 |
| PL | Basic Type Systems | Type safety and errors caused by using values inconsistently with their intended types, generic types (parametric polymorphism) | 3 |
| SE | Software Design | Design patterns | 7.5 |
| SE | Formal Methods | Role of formal specification and analysis techniques in the software development cycle, program assertion languages and analysis approaches (including languages for writing and analyzing pre-and post-conditions), tools in support of formal methods | 4.5 |

## 2.2 Major Topics/Course Outline

Major topics covered in the lectures in their approximate order of coverage, include the following.

- **Java Basics:** Introduction, interpreted versus compiled languages
- **Java Basics:** Packages, classpaths, the Java compiler
- **Java Basics:** The Eclipse integrated development environment
- **Java Basics:** Parameter passing, shallow versus deep copying, value versus reference semantics
- **Java Classes:** Fields, methods, accessibility modifiers
- **Java Classes:** Constructors, overloading
- **Java Classes:** Static fields, methods, initializers
- **Design Patterns:** Introduction, historical context
- **Design Patterns:** Singleton, Flyweight
- **Java Libraries:** java.io.*, java.util.*, java.net.*, java.math.*
- **Java Interfaces:** Declaring, implementing, using as types
- **Abstract Data Structures:** Stacks, queues, sets
- **Specification:** Introduction to interface contracts
- **Specification:** Review of basic mathematical types (integers, tuples, strings, sets)
- **Specification:** Formal contract specifications
- **Abstract Data Structures:** Partial maps (dictionaries), linked-lists
- **Analytical Reasoning:** Contract-based testing and tracing
- **Design Patterns:** Decorator
- **Analytical Reasoning:** Assertion-checking wrappers (using Decorator)
- **Java Exceptions:** Concepts, declaring, throwing, catching
- **Java Inheritance:** Concepts, type system integration, polymorphism
- **Design Patterns:** Template Method, Strategy
- **Algorithms as Components:** Parameterized sorting implementations
- **Java Generics:** Concepts, syntax, subclassing issues
- **Design Patterns:** Observer
- **Analytical Reasoning:** Introduction to verification
- **Analytical Reasoning:** Software verification with objects
- **Java Libraries:** javax.swing.*, basic Swing development

## 2.3 Summary of Closed Labs

The course is a 4-credit course because there is a weekly 2-hour lab session in addition to lectures. Each principle is reinforced during the laboratory component of the course, where students work in pairs to develop exercise solutions on shared workstations. Full credit is received when students demonstrate their solutions to laboratory assistants.

While each laboratory exercise targets a specific topic, the exercises are designed to integrate the various threads of the course with the objective of creating a unified, coherent view of the course material. In the Decorator pattern exercise, for instance, students are provided with pre- and post-conditions for a new software interface. They are then tasked with developing a decorator class responsible for checking those conditions in *any* implementation of the interface. This not only reinforces Decorator and prior reasoning concepts, but also initiates a discussion on the advantages and disadvantages of "defensive programming" (i.e., pre-condition checking in release code). The Template Method and Strategy exercise requires students to revisit the concept of recasting algorithms as reusable components. In this context, they must define an interface that recasts the fundamental notion of searching as a reusable component. They must then implement this interface using an appropriate search algorithm (e.g., linear search, binary search). This component builds upon students' prior experience recasting sorting as a reusable component; that solution is designed and implemented during class lecture.

We list summaries of closed labs below to show how we manage to weave topics in the three knowledge areas seamlessly. The complete assignments are available at [5].

1. Title: *Hello, World!* Goal: To gain initial experience working with the Java programming language. (SDF)
2. Title: *Card Game, War!* Goal: To introduce Java arrays, classes, and to use Java objects to implement an object-oriented application. (SDF)
3. Title: *War! Peace!* Goal: To use inheritance to extend previously developed classes and applications. (PL)

4. Title: *Linked Lists*. Goal: To introduce interfaces and implementations (along with object references). (SDF and PL)
5. Title: *Echo!* Goal: To introduce Java I/O and networking libraries. (SDF)
6. Title: *Pre-Conditions, Post-Conditions, and Checking Wrappers*. Goal: To reinforce understanding of pre-conditions and post-conditions, and the concept of a wrapper class for runtime checking of those conditions; use of Decorator pattern to implement a checking wrapper. (SE)
7. Title: *The Flyweight Pattern*. Goal: Developing classes that define static methods, fields, and initializers to implement and use a design pattern. (PL, SE)
8. Title: *Formal Specification-Based Unit Testing*: Goal: To implement classes based on understanding of formal interface specifications, and to develop test cases based on those specifications. (SE)
9. Title: *Formal Specification-Based Unit Testing with JUnit*: Goal: To reinforce the goals of #8 by introducing additional mathematical models, and to teach the JUnit testing framework. (SDF, SE)
10. Title: *Checking Un-Wrapped*: Goal: To combine and contrast several of the principles above and reinforce understanding of inheritance, reuse, and checking. (PL)
11. Title: *Sorting and Searching.* Goal: To introduce Java Generics as well as Template and Strategy patterns in the context of a sorting and searching problem. (PL, SE)
12. Title: *Formal Verification*. Goal: To prove the correctness of object-based code using interface specifications and a formal reasoning method. (SE)
13. Title: *Swing!* Goal: To implement a GUI application using Swing, and in the process reinforce understanding of abstract classes, inheritance, and Template pattern. (PL, SE)

The closed labs provide students with a series of hands-on opportunities to apply and reinforce a set of core principles—principles often addressed separately in typical computing courses, but which, when combined, provide a solid foundation for developing high-quality software.

# 3. SUMMARY OF PRINCIPLES

## 3.1 Software Development Foundations (SDF)

Most students in CpSc 215 are new to Java, but not to computer programming. As they learn the basic constructs of a new language and a new programming paradigm (object-oriented programming), the initial lectures and labs also introduce or reinforce techniques and strategies that are fundamental to developing quality software.

For example, in the early labs, students learn language basics and data types like linked lists. Students learn to program using an IDE, acquire strategies for debugging their programs (with or without an integrated debugger), and gain experience with standard libraries for I/O and network communication. As the course progresses, students build on these basics and learn principles of programming languages and software engineering.

## 3.2 Programming Languages (PL)

A key challenge for any Computer Science program is to prepare tomorrow's software developers to solve important problems in an ever-growing assortment of computing environments and application domains, using one or more of the many programming languages that are currently available. Understanding the principles behind programming language design, and learning to identify important differences between different languages enables software developers to learn new languages more quickly, and produce higher quality software.

In CpSc 215, students learn a new programming language (Java). More importantly, this new language is introduced in order to help students understand the fundamental differences between the procedural languages they have previously been exposed to, and object-oriented programming (OOP) languages. Students are shown how encapsulation can encourage flexibility, how modular software design can make complex programs more manageable, and how inheritance can be used to reuse common code. Students are also introduced to different type systems, calling conventions, approaches to memory management (e.g. garbage collection), error handling, and execution environments (compilation, interpretation, and virtualization). These language topics span the entire course as evidenced, for example, by the introduction of classes in lab #2 to generics in lab #12.

## 3.3 Design Patterns (SE)

Over the past two decades, design patterns have had a profound impact on software engineering practice. Their benefits are both prescriptive and descriptive. On the one hand, design patterns codify expert knowledge, providing solution templates for some of the most commonly encountered challenges in object-oriented design. On the other, patterns serve as an extended vocabulary for documenting software. Annotating a system's source base (or UML representations) with the patterns applied in its construction enables new designers to more quickly understand how a system is constructed and why it behaves in particular ways. It is no wonder that patterns play a prominent role in the design of most commercial class libraries. Indeed, based on this trend, many programming languages have been extended to simplify the expression of the most common patterns (e.g., Singleton, Iterator, Observer).

Design patterns are a central component of a well-rounded education in software engineering. The module begins with an introduction to the history of design patterns, beginning with Beck and Cunningham's interface patterns in SmallTalk, through Gamma's early patterns in C++, and finally, the seminal pattern catalog developed by the Gang of Four [7]. This historical overview is followed by a discussion of the benefits of design patterns and the impact that they have had on software engineering practice. Finally, students are introduced to the standard elements of a pattern description.

The course covers a subset of the most commonly applied patterns and they are woven throughout, providing opportunities to showcase the natural integration of design patterns, data abstraction, object-oriented programming, and formal reasoning topics. This is evidenced by noting an earlier lab, such as, lab #6 or a later one, such as lab #13. The pattern content tracks the following topical outline.

- **Singleton:** The intent of the Singleton pattern is to ensure that only one instance of a given class is constructed, and to provide global access to the constructed instance. The pattern is introduced in the context of a simple database

application involving database access code woven throughout multiple classes. In this context, Singleton provides improved code readability by factoring object construction logic into a single method call. Moreover, efficiency is improved by limiting the number of database connections. This leads to a discussion of Singleton-based connection and thread pooling, which will be leveraged in later systems courses.

- **Decorator:** The intent of the Decorator pattern is to decouple behavioral enhancements from a base object hierarchy and enable those enhancements to be applied without the use of subclassing. The pattern is introduced in a hands-on manner, through the collective, in-class design of an I/O library that mirrors the structure of the standard `java.io.*` classes. The resulting classes decouple buffering and encryption "decorations" from the file and socket streams that use them. This leads to an interesting discussion involving the motivation for interface declarations over class declarations (i.e., concrete-to-abstract dependencies versus concrete-to-concrete dependencies) – a fundamental requirement of Decorator.

- **Template Method:** The intent of the Template Method pattern is to defer a portion of an implementation to an abstract *hook* method – to later be implemented by subclass designers. To cover Template Method, students are first introduced to the concept of a *Sorting Machine*, a class that recasts the concept of sorting as a reusable component. This is, by itself, an interesting topic that reinforces the use of reusable components not only for abstract data types, but also for reusable algorithms. In this context, the sorting class implements a simple insertion sort, implemented over a previously specified `Sequence` implementation. The comparison decision (e.g., increasing, decreasing) is deferred to a hook method, allowing subclass designers to control the sort order.

- **Strategy Method:** The intent of the Strategy pattern is similar to that of Template Method, but defers portions of an implementation to an independent class hierarchy. To motivate the utility of this pattern, the sorting scenario is revisited, and multiple sorting implementations are discussed (e.g., insertion, merge, quick). Accommodating each desired ordering using Template Method results in "subclass explosion"; the class hierarchy quickly becomes unmanageable, with significant redundancy across the hook implementations. To address these deficiencies, the comparison method (or "comparator") is factored into a separate strategy object, passed to the constructor of each sorting class. This leads to an interesting comparison of the two patterns, and a discussion of when each is most suitable.

- **Observer:** The intent of the Observer pattern is to maintain an application-specific notion of consistency between a *subject* object and each of its *observer* objects. This is controlled by the subject class, which maintains a set of references to each subscribed observer. The observers are notified via method call whenever the state of the subject is changed. The pattern is introduced in the context of GUI development using Java Swing. More interesting, perhaps, is that the initial subject implementations rely on previously specified `Set` instances, specified to hold *values*. In this context, the set contains *references*, creating an opportunity to reinforce the distinction between value and reference semantics, as well as the associated tradeoffs between them.

## 3.4 Specification and Reasoning (SE)

A key thread throughout the course is the role of formal specification and mathematical reasoning in high quality software development using components. In the process students see important connections between the concepts they learn in their discrete structures course and correctness of computer programs. Specifications discussed in the course include ones for stacks, queues, lists, maps, and sorting machines (an object-oriented version of sorting). These specifications use a combination of discrete structures such as numbers, strings, sets, and functions to provide mathematical modeling of objects.

A discussion of specifications employed in the course is the topic of [8]. Once suitable discrete structures have been identified for modeling objects, corresponding notations from discrete mathematics are used in formally specifying pre- and post-conditions of operations. To ensure students understand specifications, we employ two methods. The first of these is to have them produce specification-based test points (see closed labs mentioned earlier and quiz #3 discussed later), whereby students write valid input points (that satisfy pre-conditions) and expected outputs (that satisfy post-conditions); we have also built and used a tool for this purpose, named Test Case Reasoning Assistant (TCRA) [9]. As another way to ensure students understand specifications, some of the lab assignments and a project also ask students to produce implementations of components, given only their specifications.

A sample Queue specification is given below. In the post-condition of an operation, *#self* denotes the incoming value of an object. Also, <> denotes the empty string, "o" denotes string concatenation, |self| denotes string length.

```
public interface Queue {
  // modeled by: string of object
  // initial value: self = <>

  void clear();
  // post: self = <>

  void enqueue(Object x);
  // preserves: x
  // post: self = #self o <x>

  Object dequeue();
  // pre: self != <>
  // post: #self = <dequeue()> o self
…
}
```

Teaching mathematical reasoning in conjunction with specifications helps students see the tangible benefits of using formal specifications. An example piece of code to reverse a 2-element queue is given below along with expected student answers (also the topic of quiz #4). Some aspects of the methods and proofs are simplified here to a level where the ideas can be communicated to students within one or two class periods.

We use a tabular reasoning method to teach software correctness [10]. In developing verification conditions of correctness, variable values in different states are distinguished by attaching the state number to the variable name (e.g., q0, q1, q2,

etc.). In this method, pre-condition of the operation becomes a fact (or given) in state 0 and post-condition becomes an obligation (that needs to be proved to show correctness) in state 2. When a method is called, in the state before the call, its pre-condition needs to be proved (an obligation) and in the state after the call, the post-condition may be assumed (a fact). The code is correct if all obligations can be proved from the facts and results from mathematics.

```
public static void shortQueueReverse (Queue q) {
        // pre:  |q| = 2
        // post:  q = Reverse(#q)

        Object t;
        t = q.dequeue();
        q.enqueue(t);
}
```

*Tabular Reasoning Method*

| State | Facts | Obligations |
|---|---|---|
| 0 | $\|q0\| = 2$ | $\|q0\| \neq 0$ |

t = q.dequeue();

| | | |
|---|---|---|
| 1 | q0 = <t1> o q1 | |

q.enqueue(t);

| | | |
|---|---|---|
| 2 | q2 = q1 o <t1> | q2 = Reverse(q0) |

*Proofs of obligations*:

Proof of $|q0| \neq 0$ in state 0:
**true** from fact in state 0.

Proof of q2 = Reverse(q0) in state 2:
= q1 o <t1> from fact in state 2
= Reverse(q1) o Reverse(<t1>) because |q1| = 1
= Reverse (<t1> o q1)
from string reversal theorem
= Reverse (q0) from fact in state 1.

We have assessed student learning of these ideas extensively using a Reasoning Concept Inventory (RCI). Detailed results from evaluation along with methods for continuous improvement, including the use of videos [11], are reported in [5].

# 4. EXPERIMENTATION AND ASSESSMENT

## 4.1 Quizzes

The course principles are all reinforced in the labs, programming assignments, quizzes, and exams. In this section, we just discuss quizzes and projects to show how we cover and assess student performance in various knowledge areas. While both quizzes and project assignments cover SDF and PL topics, for SE, the quizzes place more emphasis on formal methods, and assignments place more emphasis on design patterns.

The first quiz requires students to design a simple class that meets a set of specified requirements. The second requires students to apply object-oriented programming principles in order to implement abstract data structures. The third requires students to read and interpret mathematical specifications by selecting input/output test points for a set of methods. Students are also required to develop a JUnit test suite for an abstract data type. The final quiz continues the analytical reasoning path, requiring students to develop the mathematical assertions that must be proven to show that a given code fragment satisfies its specification. For extra credit, students may prove the assertion set to verify the code's correctness.

Table 2, below, summarizes students' quiz performance and the associated knowledge area topics covered for the 2012-13 academic year. Each class has between 25 and 30 students. (Though the course has been taught for multiple years, we present the data for only one year to avoid confounding variations across quizzes and assignments over the years.)

In the table, the following abbreviations have been used; they correspond to the body of knowledge coverage in Section 2.1. Under SDF, AD refers to Algorithm Design, FDS refers to Fundamental Data Structures, and DM refers to Development Methods. Under PL, OOP is Object-Oriented Programming, and BTS is Basic Type Systems. Under SE, SD is Software Design (including design patterns), and FM is Formal Methods.

**Table 2: Consistent Quiz Averages across Topics**

| | Instr. #1 (Fall '12) | Instr. #2 (Fall '12) | Instr. #3 (Spr '13) | Instr. #4 (Spr '13) |
|---|---|---|---|---|
| Q1: SDF(AD), PL(OOP) | 79 | 72 | 78 | 57 |
| Q2: SDF(FDS), PL(OOP) | 85 | 86 | 86 | 72 |
| Q3: SE (FM), specification | 82 | 78 | 82 | 82 |
| Q4: SE (FM), reasoning | 59 | 73 | 81 | 92 |

Some general observations can be made concerning this data. Students have performed uniformly well across the topics, though three different instructors with varying teaching experiences and backgrounds taught the course; in the table, instructor #2 and #3 are the same person. For the fourth quiz, the improvement in spring can be attributed to an intervention introduced to improve the teaching of reasoning; details of the intervention are documented elsewhere [5].

## 4.2 Project Assignments

The programming assignments are designed to provide students with opportunities to design appropriate class hierarchies and combine multiple principles to solve more complex problems. The first course project requires students to develop a basic web crawler. Given a URL as a command-line parameter, the crawler retrieves and stores all embedded images and other documents, and then recursively follows any embedded links, down to a specified maximum depth. First, and most important, students gain experience designing and implementing a small (but interesting) Java application. This reinforces their object-oriented design skills, as well as their ability to effectively apply the Java development tools. More specifically, the project reinforces student understanding of Java interfaces, the Singleton pattern,

Java exceptions, and the Java IO library. It also improves their ability to debug Java-based systems.

The second course project also involves topics from multiple knowledge areas. Students must develop a formal interface specification for a linked-list abstraction given a Java interface documented using natural language. Next they must develop an implementation of this interface and develop a robust test suite to validate the correctness of their implementation with respect to their specification. In the process, students gain experience on the following aspects: understanding a formal interface specification based on an abstract model, implementing a class based solely on their understanding of its specification, and implementing specification-based unit tests using the JUnit testing framework.

Of all the projects, the third project is the most pattern-intensive. Students are required to implement a fully-functional email client using Java Swing, which includes support for a persistent contact database. The suggested architecture is annotated with the required patterns, again underscoring their prescriptive and descriptive benefits. To complete the project, students must understand how Strategy, Decorator, and Observer are used in the Java class libraries, and must be able to implement aspects of Singleton, Strategy, and Observer themselves. This project has been used for many years and continues to be popular with students. It is not uncommon for students to present their work to future employers as evidence of their object-oriented design capabilities.

**Table 3: Consistent Project Averages across Topics**

| | Instr. #1 (Fall '12) | Instr. #2 (Fall '12) | Instr. #3 (Spr '13) | Instr. #4 (Spr '13) |
|---|---|---|---|---|
| A1: SDF(AD), SDF (DM), PL(OOP) | 83 | 82 | 89 | 80 |
| A2: SDF(FDS), SDF(DM), PL(OOP), SE(FM) | 75 | 82 | 96 | 93 |
| A3: SE(SD), PL(BTS) | 82 | 97 | 98 | 94 |
| A4*: PL(OOP), PL(BTS), SE(SD) | 100 | 100 | 100 | 100 |

While all scores are subject to grading variations, among the project assignment scores, there is a notable difference in performance between fall and spring classes on the second assignment. One possible reason for this increase is perhaps due to a better understanding of specifications (from an intervention). Assignment 4 is optional and requires students to develop a straight-line interpreter based on the Interpreter Pattern. The 100% statistic here is misleading; only 2 students submitted it in the fall and 4 submitted it in the spring.

We have also collected extensive data on student attitudes using pre- and post-surveys for the course [5]. The results show a statistically significant positive shift in the perception of students on how to develop high quality software.

# 5. SUMMARY

It is inevitable that new CS courses that combine multiple topic areas have to be developed, because the trend is for institutions to continue to reduce the number of required CS courses for majors in favor of more electives, as computing evolves as a discipline. This paper details an ACM 2013 curriculum exemplar course that interleaves not only three knowledge areas but also object-oriented and design pattern practices with foundations for specification and analytical reasoning. The experimentation with the course over 5 years has shown that the combination is effective in achieving student learning goals.

# 6. ACKNOWLEDGMENTS

This research was funded in part by the NSF grants CCF-1161916, CNS-0745846, DUE-1022191, and DUE-1022941.

# 7. REFERENCES

[1] IEEE/ACM Computer Science Curricula 2013, Ironman Draft V1, 2013. http://ai.stanford.edu/users/sahami/CS2013/ironman-draft/cs2013-ironman-v1.0.pdf

[2] Gries, D., Marion, B., Henderson, P., Schwartz, D., "Panel: How Mathematical Thinking Enhances Computer Science Problem Solving," *Procs. 32nd ACM SIGCSE Conference*, February 2001.

[3] Krone, J., Baldwin, D., Carver, J. C., Hollingsworth, J. E., Kumar, A., and Sitaraman, M., "Panel: Teaching Mathematical Reasoning across the Curriculum," *Procs. 43rd ACM SIGCSE Conference*, March 2012.

[4] Sitaraman, M., Hallstrom, J.O., White, J., Drachova-Strang, S., Harton, H., Leonard, D., Krone, J., and Pak, R. Engaging students in specification and reasoning: "hands-on" experimentation and evaluation. *ACM SIGCSE ITiCSE*, 2009, 50-54

[5] Drachova, S.V., Teaching and Assessment of Mathematical Principles for Software Correctness Using a Reasoning Concept Inventory, Ph.D. Dissertation, Clemson University, 2013.

[6] Hallstrom, J. O., http://people.cs.clemson.edu/~jasonoh/courses/cpsc_215_fall_2012/Home.html, 2012.

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[8] Cook, C.T., Drachova, S., Hallstrom, J.O., Hollingsworth, J., Jacobs, J.P., Krone, J., and Sitaraman, M., "A Systematic Approach to Teaching Abstraction and Mathematical Modeling." *ACM SIGCSE ITiCSE*, 2012.

[9] Leonard, D.P., Hallstrom, J.O., and Sitaraman, M., "Injecting Rapid Feedback and Collaborative Reasoning in Teaching Specifications," *ACM SIGCSE*, 2009.

[10] Heym, W. D., *Computer Program Verification: Improvements for Human Reasoning*, Ph.D. Dissertation, Ohio State University, 1995.

[11] Hollingsworth, J.H., SIGCSE Workshop 2012 Instructional Video Series. http://www.cs.clemson.edu/group/resolve/teaching/ed_ws/sigcse2012/index.html