

APPRENTICESHIP IN UNDERGRADUATE JAVA PROGRAMMING

Dale E. Parson

Kutztown University of Pennsylvania, Department of Computer Science

parson@kutztown.edu

ABSTRACT

Even the best textbooks for Java programming have one substantial deficiency. They use small, isolated programming examples and throw-away lab exercises that do almost nothing to convey the situated application of object-oriented language constructs. An alternative approach is to embed Java programming assignments as encapsulated modules in an extensible application framework built up across semesters and several courses, with instructors serving as lead analysts and architects. Taking its cues from professional software engineering practices, as well as the educational practices of situated learning, cognitive apprenticeships and microworlds, this approach has the fundamental goal of embedding the introduction to Java programming in a realistic software and social environment from which students can learn via reading, observation, incremental exploration, interaction and deployment of their creations, in addition to the writing and testing of code. Students get a feeling for the categories of work and types of problems they will encounter in supporting and extending professional software frameworks created initially by others. They learn to apply object-oriented constructs by the end of a semester, and they may perform more sophisticated engineering on their software framework in subsequent courses. This paper reports the successful results and pitfalls of applying this approach.

KEYWORDS

cognitive apprenticeship, Java, microworld, situated learning

1. Introduction

Object-oriented software engineers spend their working lives exploring, extending and repairing frameworks that they did not create. Entrée into a project often comes with a sense of being overwhelmed by the volume of information that the software engineer feels compelled to absorb. New entities and relationships are everywhere. Some are well-structured and documented, but many are not. There are many skills gained by experience that keep an engineer from becoming overwhelmed. Cognizance of the structural differences between airtight inter-module boundaries on the one hand and amorphous inter-module coupling on the other helps an engineer restrict the scope of exploration to only those modules with potential impact on the tasks at hand. The ability to discern structure and design intent while reading other people's code is a critical skill. Another is the ability to distill specifications into concise

documentation for annotating interfaces and system components. Communication with engineers and other players is perhaps the most important skill because, when employed artfully, it can open paths to the other important skills. A novice software engineer cannot cultivate these skills working in isolation on small projects. It is necessary to embed one's attention in a growing, evolving software system that requires reading and comprehension in order to grow these skills.

This paper summarizes an ongoing approach for teaching Java programming to undergraduate students at Kutztown University in the CSC 243, *Java Programming* course. The approach is one of analyzing and extending a software framework that persists and grows across semesters. The framework serves as a product that the instructor and students extend.

The author initiated this approach to teaching Java programming beginning in the fall 2008 semester for two reasons. First, the author had just entered academia as a full-time assistant professor after working for twenty years as a professional software engineer, including an eight year stint as a senior software architect in a team that designed, built and maintained a user-extensible software tools framework for simulating, debugging, and monitoring embedded communication systems [1-5]. The author had mentored numerous junior software engineers in the practices of object-oriented software engineering of systems implemented using the Unified Modeling Language (UML) [6], C++ and Java, as well as mentoring numerous graduate student interns from Lehigh University at Bell Laboratories in Allentown. The author saw an opportunity to apply industrial mentoring techniques to teaching Java programming at Kutztown University.

The second motivation for initiating this approach was the paucity of realistic examples of Java software systems among the lab exercises in textbooks. The conventional approach taken by introductory Java textbooks [7-15] is that of teaching students the basic mechanics of the programming language using small, concise examples that illustrate the topics at hand, but that do not embody problems and opportunities found in object-oriented systems. The examples illustrate basic language mechanics, but by their nature they cannot inform the students about appropriate contexts in which to apply these mechanics. Some mechanisms of an object-oriented language such as package partitioning, interface and implementation inheritance, and generic class design cannot be realistically illustrated using small, concise examples. These system-level language constructs span multiple Java classes and packages. They require embedding within substantial

example bodies of code before their utility becomes clear. To understand the utility and correct application of the object-oriented features of a language, it is necessary to see them employed and to work with them *in situ*. Students learn about object-oriented constructs by watching them interact in their intended environment, and by interacting with them there.

Novices do not have the experience to employ out-of-the-book abstractions. It is precisely this lack of experience that the present approach looks to address. Rather than teach object-oriented abstractions as concepts, this approach seeks to make them concrete experiences that are acquired incrementally in the process of performing small, concise increments of work inside a larger, object-oriented environment. As with conventional textbook code, student examples and projects are small and concise, but unlike textbook code, these small, concise pieces of code comprise modules that are integrated into a surrounding virtual environment. In teaching Java programming to second-year computer science students, the author does not seek to overload them with object-oriented abstractions on top of the language mechanics that they must necessarily learn. The author seeks, instead, to embed their language-mechanical work in a supporting software framework that leverages their efforts, with the payoff being useful, entertaining and extensible software tools that students can take with them when the classes are over.

This approach is also appropriate for advanced students taking courses in object-oriented analysis and design. Such courses often use hypothetical systems which the students must hypothetically analyze and design. A much more concrete and motivating approach is to have students analyze and extend the framework inside which they worked as novices.

The Java Programming course at Kutztown University sits outside the main sequence of computer science requirements and prerequisites. It is a 200-level elective course with prerequisites including the two-semester introductory computer science sequence, which is taught using C++ for programming exercises, along with a first course in discrete mathematics [16]. Due to the fact that Java Programming can be used as an elective in a computer science major or minor, it attracts sophomores, juniors and seniors with varying degrees of experience in programming. This variety of student background makes designing appropriate lab exercises challenging. Sophomores fresh out of the first year introductory course sequence have worked entirely with small, self-contained lab exercises, making this course their first exposure to object-oriented programming in the large. The intent is for them to learn both Java mechanics and the appropriate application of Java's object-oriented constructs in a surrounding software environment.

2. Apprenticeship in Java Programming

2.1. Related Work in Cognitive Apprenticeship

As noted in the introduction, this approach grew primarily out of the author's experience acting as a mentor for junior software engineers and graduate interns in an industrial team for engineering software tools for embedded systems [1-5]. The modus operandi for that project was for senior software engineers to partition subsystems into encapsulated, documented modules, and to assign junior engineers the tasks of implementing, testing and supporting those modules. Graduate interns performed exploratory, higher risk tasks because such tasks were not entangled with short-term customer demands and deadlines typical of products with periodic delivery dates. Junior engineers helped to make ongoing product plans into reality, while interns assisted senior engineers perform reconnaissance into new software technologies.

The job of senior engineers working with both junior engineers and interns was to partition the work space so that these junior people could work productively as early as possible despite their relative inexperience by letting them focus their attention within well-defined, encapsulated modules. Senior engineers architected modular systems both for the sake of those systems and for the sake of junior people working within them. Over time, of course, junior engineers became seasoned as they learned not only the architecture of the surrounding software system but also the process used to create it. The guiding organizational principle for this project team of about thirty people, including engineers, testers, technical writers and managers, was the *Surgical Team* model of Fred Brooks, guided by a *Chief Surgeon* [17]. There were two levels of hierarchy in the management and technical structure of this team, with the lead architect (the author) working with four senior engineers leading efforts on each of four technical subsystems, with each junior engineer working within one of those subsystems. The lead architect also worked with graduate interns in performing exploratory work such as tools evaluation and rapid prototyping of new capabilities.

There are several strands of research in the field of education that relate to the current approach. *Situated Learning* refers to an approach in which learners are situated in both a physical and social context wherein they can acquire, practice and refine skills [18]. Situated Learning's concept of *Legitimate Peripheral Participation* is the notion that even novices can perform useful if initially peripheral work on a real project or product, as opposed to working on abstracted, throw-away academic exercises. As novices gain experience and expertise they move from the periphery towards the center of their projects. Working within a realistic social environment is a key aspect of this approach.

Both Situated Learning and the related *Cognitive Apprenticeship* approach [19] derive from the study of historical apprenticeships. Both depart from the practices of conventional apprenticeships, in part because such apprenticeships are constrained by economic time-to-

market demands, but mostly because both deal with educational domains that are more general and abstract than the domains of historical apprenticeships. These approaches use apprenticeship as a lens through which to critique more conventional approaches to education [18]. They use apprenticeship-like techniques to guide learners in the acquisition of cognitive skills.

The concrete practices of software engineering allow the present approach to borrow more directly from the practices of conventional apprenticeships. It is possible to work with undergraduate students as apprentices in software engineering. The present approach is a hybrid cognitive / conventional apprenticeship approach. It initiates students into concrete practices from which both computing domain skills and cognitive skills emerge. The author's background in mentoring junior engineers and graduate interns comes into play in engaging undergraduate computer science students in a similar manner. A substantial challenge, clearly, is scaling and adapting this master / apprentice relationship to work with second-year undergraduates, one semester at a time, without overwhelming those undergraduates.

One additional thread from the field of education that relates directly to computing is that of immersing students in *Microworlds* within which they construct knowledge and acquire skills by engineering virtual environments and tools [20,21]. Participation in a sophisticated software environment as proposed by the present approach, as contrasted with performing a series of unrelated, throw-away programming exercises, constitutes creation and extension of a microworld that maps directly to the professional world of software engineering.

Of course, software engineering courses per se have been in existence for decades, and every programming course shares some degree of the constructivist, hands-on perspective of the aforementioned educational threads. What distinguishes the present approach from more conventional software engineering courses are the facts that it uses a growing software framework as an environment in which to embed introductory Java programming skill acquisition by sophomores – this is not a software engineering course for advanced undergraduates – and it extends a framework that persists across semesters. Each semester provides a new experience in extending this framework in new directions. One year's work by the author and four sections of Java programming students has created a novel interactive system for algorithmic musical improvisation that we have performed in public as an ensemble. Two new sections of students are now continuing to extend and to deploy this framework in the spring 2010 semester.

All introductory programming courses embed students in a virtual environment. The first year course sequence that is a prerequisite for this course embeds students in the virtual environment of interaction with editors, compilers, a debugger and an operating system. The present approach extends that embedding to include the very thing that students are learning, Java software constructs. They learn Java by directly exploring and extending the virtual

environment in which they work. While the primary focus is learning a new programming language, students also take a significant first step towards learning how they will ultimately apply skills and interact with peers in their professional lives.

2.2. Goals of this Apprenticeship Program

This subsection lists the main goals of the current approach. A subsequent subsection on accomplishments and pitfalls considers how well these goals have been achieved to date.

- The Java Programming course must include all topics required by the departmental syllabus, which was created before the initiation of this apprenticeship approach. Required topics include Java counterparts to the C++ topics learned in the first year sequence; new topics include exceptions, events, introductory graphical user interface construction and applets. Object-oriented programming constructs such as Java interfaces, abstract classes and generic classes are new topics, because the first year sequence does not explore object-oriented design. Graphical user interface construction is also a new topic, although many students have had experience in this area thanks to an elective course in Visual Basic.

- Given the fact that isolated, throw-away programming projects do not illustrate the use of object-oriented architectural aspects of Java, the current approach embeds programming exercises within modules in an extensible Java framework. The goal is one of realistic illustration of the use of these language aspects. This approach extends the normal embedding of a programming course in an operating systems and set of tools by further embedding the course in a growing framework that is coded and documented using the language being learned. Students are not required to comprehend the framework that surrounds their modular assignments. Initially they are encouraged to maintain focus on their deliverable modules. As they become more familiar with the language and the project framework, they naturally begin to explore and comprehend aspects of the object-oriented framework.

- A professional software engineer who joins a project with a substantial code base initially feels overwhelmed by the complexity of the virtual environment being entered, and one of the goals of the current approach is to duplicate that experience and the methods for its resolution in a programming course. One of the guidelines of the Cognitive Apprenticeship approach suggests immersion into problems somewhat beyond the reach of students.

“Cognitive apprenticeships are situated within the social constructivist paradigm. They suggest students work in teams on projects or problems with close scaffolding of the instructor. Cognitive apprenticeships are representative of Vygotskian "zones of proximal development" in which student tasks are slightly more difficult than students can manage independently, requiring the aid of their peers and instructor to succeed.” [22]

In the present approach the surrounding Java framework and the small, encapsulated modules for student projects partitioned by the instructor / architect constitute the

scaffolding. Some students do initially feel overwhelmed by the complexity of the surrounding framework. Exposing students to this feeling is one of the goals of the current approach, because it is a feeling that they will experience again as professional software engineers. Exposing students to the resolution of this feeling is critical. Resolution comes about by establishing and maintaining focus on the specific tasks at hand. Completing projects on time mandates this focus of attention. As students become more competent, they begin to understand the surrounding environment incrementally as they work within it. Later projects require more comprehension of inter-module interaction and language mechanics of the framework.

- Requiring students to read code and documentation written by others in order to understand their tasks is an important goal in guiding students who are learning professional engineering practices. The current approach requires students to flesh out stub methods and data structures drafted by the instructor within the framework written by the instructor. Students must also write code and documentation that conforms to basic writing standards.

- Construction and extension of an interesting software product that outlives any given semester is a central goal of the current approach. Like industrial apprentices, students help to create interesting artifacts that are useful beyond the setting in which they are created.

The role of the instructor as system architect is critical to the current approach. The instructor must analyze project alternatives in order to guide construction of software modules that both educate students in the desired language constructs and that lead to interesting, useful products. The instructor must partition the design space in order to create programming projects that are reasonable for second year students. Fortunately, this work need not and cannot be done using a monolithic, waterfall process. Curriculum design within this approach is an iterative process. Both the instructor and students learn and refine the plan as they go along. As students gain expertise, their unique contributions enrich the framework beyond the plans of the instructor.

- The instructor must not be a perfectionist in the role of architect. Professional architects guiding the creation of new software systems typically do not hit upon optimal solutions on the first try. Furthermore, given the large amount of work to be done, making small mistakes is natural. The author uses project planning and post mortem sessions as opportunities for design review by students. Given the size of the problem space, it is not unusual for advanced students to see superior solutions to sub-problems such as the easiest way to code a specific set of tests or to structure some implementation data. Typical textbook coding examples are not rich enough to explore the imperfection and iterative development that are intrinsic in non-trivial software development. This goal of realistic imperfection and iterative improvement fits well with the practices of Cognitive Apprenticeship.

“Occasionally, the problems are hard enough that the students see him (the instructor) flounder in the face of real difficulties. During these sessions, he models not only the use of heuristics and control strategies, but also the fact that

one’s strategies sometimes fail. In contrast, textbook solutions and classroom demonstrations generally illustrate only the successful solution path, not the search space that contains all the dead-end attempts. Such solutions reveal neither the exploration in searching for a good method nor the necessary evaluation of the exploration. Seeing how experts deal with problems that are difficult for them is critical to students’ developing a belief in their own capabilities. Even experts stumble, flounder, and abandon their search for a solution until another time. Witnessing these struggles helps students realize that thrashing is neither unique to them nor a sign of incompetence.” [19]

Given the time constraints of a twelve-credit teaching load, selecting a software application with sufficient complexity to be genuinely interesting almost guarantees that it will not at the same time be perfect. Dealing with imperfection and iterative refinement becomes part of the course strategy.

In summary, the goals are 1) teach Java programming constructs required by the departmental syllabus; 2) illustrate use of Java’s system-level constructs by embedding student-assigned modules within an extensible Java framework; 3) give students the feeling of being slightly overwhelmed by the framework and the resolution of this feeling via focus of attention on their deliverable modules; 4) require students to read documentation and code written by others in order to understand their projects by embedding their assignments within existing code, and to write documentation and code for reading by others; 5) construct and extend an interesting software product that outlives any given semester; 6) include imperfection and iterative refinement that is typical of professional software systems in the process.

2.3. History and Upcoming Plans of this Program

This subsection summarizes the activities of two semesters of CSC 243, Java Programming at Kutztown University, with two sections of the course per semester. The author and students initiated and refined this approach in the 2008-2009 academic year. We have resumed this program in two sections of CSC 243 in the spring 2010 semester.

A quick examination of available textbooks [7-15] at the outset of the fall 2008 semester made it clear to the author that typical code examples and student exercises do not address how Java is used to interconnect libraries and build extensible systems. After considering projects that were likely to arouse student interest, the class settled on having students implement a software Scrabble™ game. This game exercises character string manipulation mechanisms and two-dimensional arrays that are the topics of early chapters in the text. Students already had a two-semester working familiarity with C++ programming from having taken the department’s introductory programming courses. They understood basic control and data structuring constructs. At this stage the plan was for nothing more than to build a single, non-extensible game. After playing several games in groups, the author and students constructed an outline on the whiteboard of major entities and activities involved in

the game. A few days later the author handed out the initial assignment with the following two opening paragraphs.

“We have already started looking at requirements for the Scrabble™ game in class. We are going to build an interactive Scrabble game in Java, using Java classes that we have discussed. I have partitioned the problem into the classes diagrammed below, and you are going to implement the methods and fields of one of these classes, *ScrabbleBoard*. I have written partial implementations of all classes, including a skeletal implementation of *ScrabbleBoard*. It presently consists mostly of so-called *stub* methods. These are methods with the correct signatures — parameter types, return types, and exception types — that do not contain the required, fleshed-out application logic or data. You will flesh out this class, including writing Javadoc-compatible comments for all methods.”

“This program is an exercise in the manipulation of Java chars and ints, Character, String and StringBuilder objects, and 2-dimensional arrays. Subsequent programming assignments will build on this code base.”

Figure 1 below is the UML [6] class diagram for this assignment. This was the first exposure to UML graphical notation for most students. Some UML class diagrams would appear in later chapters of the textbook.

At this stage, students were not expected to learn the intricacies of designing object-oriented relationships or expressing them in UML. The goal of Figure 1 was to give them a visual map of entities that they had already outlined while playing the game. Their specific task was to implement several fields and three methods of class *ScrabbleBoard*. Method *validateWord* checks that a proposed word fits on the board and accords with game rules. Method *putWord* re-validates a word by invoking *validateWord* and then places and scores the word on the board. Method *readBoard* returns a matrix of strings representing the board as it appears to a player. Fields within *ScrabbleBoard* including a two-dimensional character array maintain the state of the board.

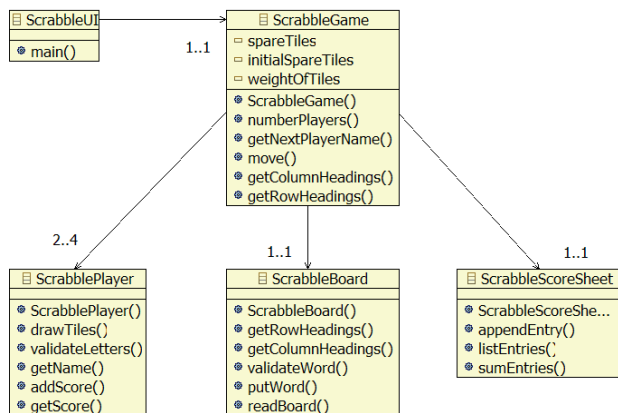


Figure 1: UML Class Diagram for Assignment 1, Fall 2008 -- A concrete game

Even though students were not familiar with object-oriented design mechanisms at this stage, they were

familiar with the rules and entities of a Scrabble game. Walking them through a player’s move scenario is straightforward. A human player enters a textual move command from a keyboard via a *ScrabbleUI* object. This object invokes the *move* method of *ScrabbleGame*, which proceeds in the following steps. First, it invokes *ScrabbleBoard*’s *validateWord* method to test a proposed move; this method returns a string showing exactly which of the proposed tiles (letters) already reside on the board. The game then invokes *ScrabblePlayer*’s *validateLetters* method to ensure that the current player holds the remaining, needed tiles. Either of these validation methods can throw an exception, which sends an error message to *ScrabbleUI*. It is only if validation succeeds that *ScrabbleGame* invokes the student’s *putWord* method to place and score the word, after which it invokes the player’s *addScore* and *drawTiles* methods and the score sheet’s *appendEntry* method.

Students can comprehend the steps that are occurring when presented with the description of the previous paragraph in concert with Figure 1’s UML “map.” Students have already assisted with the analysis while playing the actual board game. Students are not expected to understand mathematically-inclined abstractions of object orientation at this stage. Nevertheless, their work is situated in an object-oriented design. The programming assignment channels students to maintain focus on immediate requirements of the three methods that they must implement within schedule constraints. Students are free to use the UML class diagram, the Javadoc documentation and the source code to explore other regions of the design as their time and talents permit.

Subsequent projects within the first semester extended the first-pass design of Figure 1. Figure 2 shows the substantial architectural enhancement of the second assignment. Replacing the direct reference relationship from *ScrabbleUI* to *ScrabbleGame* of Figure 1 is a reference relationship through the Java interface *TwoDimensionalBoardGame*. The user interface and test driver component *ScrabbleUI* no longer encodes information about the specific game that a player plays. Instead, *ScrabbleUI* now implements a plug-in class loader to load a class by name that implements interface *TwoDimensionalBoardGame*. For readers unfamiliar with UML class diagrams, each open arrow indicates a reference link from one class to another, while the triangular arrow from *ScrabbleGame* to *TwoDimensionalBoardGame* indicates an inheritance relationship. *ScrabbleGame* implements interface *TwoDimensionalBoardGame* in Figure 2.

Student work in the second assignment consisted of documenting the operations of interface *TwoDimensionalBoardGame* using Javadoc comments, and then implementing all new operations that it introduces within class *ScrabbleGame*. The new operations specify methods that query game state. The implementation methods are wrappers on top of existing methods and fields in *ScrabbleGame* and the classes to which it refers. This assignment is one of partitioning and presenting an existing

code base so that it can be used as a plug-in. Students can understand the concept of creating an environment that supports the play of more than one particular game. The architect's contribution consisted of the design refactoring from Figure 1, along with the construction of a name-based loader within class `ScrabbleUI` for loading a specific plug-in class, as well as construction of persistence machinery for saving and restoring the state of a game. All of the classes of Figure 2 except for `ScrabbleUI` implement interface `java.io.Serializable`, making them easy to save and restore together. Students thus became exposed to two practical applications of the Java interface construct. The first is to use it to isolate the game framework from the details of the specific game being played, making the framework reusable among different games. The second is library support for *persistence*, care of the `java.io.Serializable` interface. Students need not understand interfaces and persistence in the abstract. Instead, they get direct working exposure to these constructs as concrete entities. Their own coding, however, consists of the more achievable tasks for novices of writing documentation and implementing wrapper code.

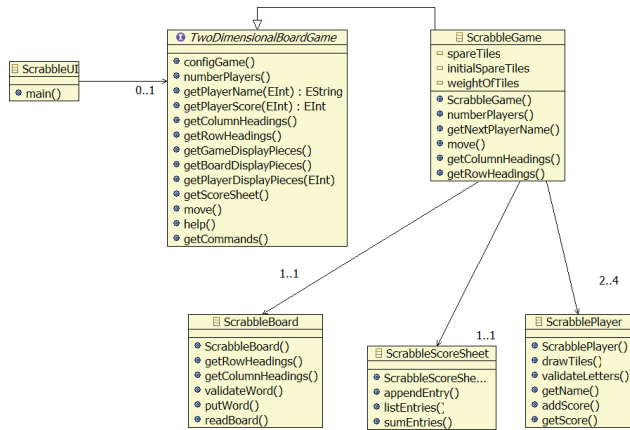


Figure 2: UML Class Diagram for Assignment 2, Fall 2008: An abstract game interface

Assignment three had students temporarily replace the use of container class `java.util.HashMap` for mapping tile characters to their point values, and for mapping tile characters to their occurrence count, with a generic mapping container class that the students wrote using elementary data structures. This assignment was nowhere near a full implementation of the `java.util.Map` interface. Its goals were to expose students to the use of Java generics in writing library container classes, and to give them an opportunity to write a unit test driver as the main method of their library class.

Assignments four and five, illustrated in the class diagram of Figure 3, had students write a subset of the graphical user interface (GUI) code for a stand-alone graphical Java application (`JFrame`) and applet hosted within a web browser (`JApplet`) respectively. This architecture creates another Java interface, `TwoDimensionalBoardUI`, that specifies the game-neutral operations of a game user interface object. Refactoring

partitions the former loader and test driver class `ScrabbleUI` into two parts, the `GameMain` class that implements the loader and test harness, and the `GameTTYUI` class that houses the terminal I/O code of the previous assignments.

There are new classes `ScrabbleSwingFrameUI` and `ScrabbleSwingAppletUI` that provide graphical user interfaces for stand-alone games and browser-hosted applet games, respectively. Design analysis outlined by the instructor for the students reveals a path for minimizing work in going from a `JFrame` application to a `JApplet` applet by placing most of the GUI work into helper class `ScrabbleSwingRootPaneHelper`. The student GUI code went into this class, using for examples a subset of its code supplied by the instructor. Lessons included construction and arrangement of nested graphical components, reaction to user-initiated input events, and interaction with the `ScrabbleGame` object in getting and advancing state. An important object-oriented concept embedded in this class diagram is that of *class covariance* [23]. Game-specific classes that implement `TwoDimensionalBoardUI` must covary with classes that implement interface `TwoDimensionalBoardGame` in order to present users with game-specific GUIs. Assignments three through five have students gain practical experience with run-time type checking to ensure type consistency of the internal structure of generic classes and of covarying classes such as those of Figure 3.

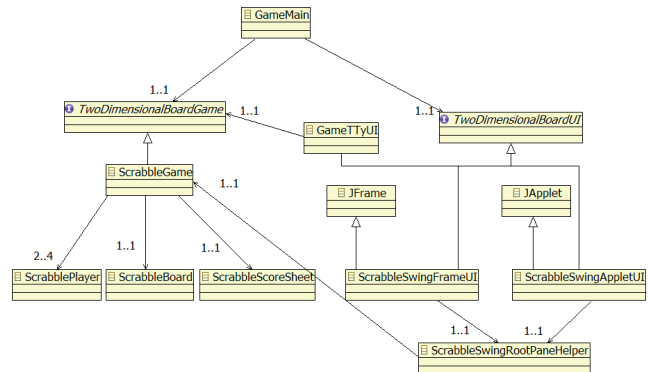


Figure 3: UML Class Diagram for GUI Assignments 4 and 5, Fall 2008: Object-Oriented Covariance

Clearly, a good deal of object-oriented analysis went into partitioning the interfaces and classes of Figure 3 to support a game-neutral framework with plug-in game classes and plug-in game user interfaces. In addition to adding graphical components and their event handlers to `ScrabbleSwingRootPaneHelper` for assignment 4, students added HTML parameter lookup method calls to the `ScrabbleSwingAppletUI` class for assignment 5. While they were not required to understand the overall architecture, the author discussed high-level design decisions with them at the outset of each assignment. When presented with this information in a form related directly to game construction, most students exhibited a clear high-level understanding of the reasons for design decisions.

The reason for going into the first semester assignments in such detail is that it illustrates the suggestion that embedding students within an object-oriented framework

makes object-oriented concepts concrete. Class diagrams are maps of work being done. Students perform only a subset of that work, but they learn object-oriented concepts from exposure to the maps as they develop.

Student work on some aspects of the game environment exceeded the author's contribution. Graphical user interface design in particular is an area where students can apply both technical and aesthetic skills in a way that complements rather than duplicates the work of the instructor.

This subsection concludes with a summary of the steps taken in the second semester to extend the architecture illustrated in Figure 3. The second semester reused all of the design and code of the first semester diagrammed in Figure 3, adding a second Java package that generates MIDI (Musical Instrument Digital Interface [24,25]) music from word paths while a game is played. Student software treats Scrabble words as chords, mapping the letters in these words to MIDI notes and playing these notes via Java's `javax.sound.midi` library classes [26]. The final class diagram with the new package roughly doubles the number of classes of Figure 3. Derived classes in this new MIDI-oriented package use inheritance to implement object-oriented *interception* of game-based method calls in the interest of game-to-MIDI translation. The Scrabble-to-MIDI translator class also uses Java *eventing* to monitor moves in the game. Interception and eventing are both important control mechanisms that students learned by using them concretely rather than by studying them as abstractions. Student code in this package made extensive use of the `javax.sound.midi` library for generating instrument performances during the play of a game.

The first project entailed construction of a depth-first search method for querying the words of a `ScrabbleBoard` as coded in the previous semester. Students wrote a query method to search outward from any tile on the board, along with a helper method and a main test driver. The student method returned an array of `word:location:orientation` triplets.

The second assignment included construction of a game-neutral Java package and a Scrabble game-specific sub-package for listening to move events within a game and mapping game board configurations following each move to music generated via the `javax.sound.midi` library. The student assignment consisted of design completion, coding and testing of mapping class *ScrabbleToMidi* for a `ScrabbleGame`, using the `java.util.TreeMap` library class to store mapping configuration parameters. Each word in the result of a depth-first search of the board maps to a MIDI chord, with each tile within the word mapping to a MIDI note. The initial solution played music mapped from the board using block chords of piano notes. Students also wrote methods to query and update mapping configuration parameters such as the pitch associated with a given tile and the duration of a note.

The third assignment was similar to that of the first semester. Students replaced `java.util.TreeMap` in the second assignment with their own generic mapping class.

For the fourth assignment, the instructor added a significant number of MIDI parameters to the student module of assignment 2 in support of tempo, multiple instrument voices, musical keys, rhythms and melody lines. Students designed most of the graphical user interface of class *ScrabbleToMidiConfigUI* for musical configuration adjustment. This GUI allows game players to change the music generated from the game in many novel and interesting ways as they play. Several students made enhancements to this configuration GUI beyond project requirements. Some students also added non-GUI enhancements to the program, such as the inclusion of a favorite blues scale within the configuration parameters.

The fifth assignment consisted of porting the fourth assignment to run as an applet under a browser, similar to the work of the prior semester.

Project plans in progress for the spring 2010 semester include construction of an undo / redo tree for game state. The plan is for a persistent tree instead of a conventional undo / redo stack, because a tree will allow exploration of alternative game moves.

2.4. Accomplishments and Pitfalls

Both semester offerings of the course have met the previously stated goals. We are currently in our third semester of extending this project, adding a spelling checker and a graphical user interface for a multi-level undo/redo capability that supports exploration of alternative state paths. Besides enhancing the game, this capability allows repetition of generated musical structures from earlier game states, along with exploration of alternative musical structures generated from alternative game states.

As the complexity of the existing framework increases, there are an increasing number of alternative ways to implement this course strategy. 1) Students have learned the Java programming constructs required by the departmental syllabus. 2) The game development and game-to-MIDI framework use Java's system-level constructs by embedding student-assigned modules within an extensible Java framework. 3) Some projects give students the feeling of being slightly overwhelmed by the framework; resolution of this feeling occurs via focus of attention on their deliverable modules. 4) The course requires students to read documentation and code written by others in order to understand their projects by embedding their assignments within existing code, and to write documentation and code for reading by others. 5) Students and the instructor construct and extend an interesting software product that outlives any given semester. 6) Projects include imperfection and iterative refinement that is typical of professional software systems in the process.

A high point for this project was an ensemble musical performance of Scrabble-to-MIDI by students and the author at a public computer audio seminar at Kutztown University on September 30, 2009. This project received second-page coverage in the October 2 *Reading Eagle* newspaper [27], including a photograph of a student

playing the game, a student interview, and a first-page lead-in that included a screenshot of a student GUI. The author has given solo performances of Scrabble-to-MIDI as part of two international musical webcasts, one in June, 2009 and the other on New Year's Eve in December, 2009 [28]. An ensemble performance by the author and students is planned for the annual conference of the Pennsylvania Association of Computer and Information Science Educators at West Chester University in April 2010 [29]. We have also used this software in interactive demos in recruiting fairs with high school students, and we look forward to adding and deploying new capabilities.

This approach has survived some pitfalls. The most serious occurred at the due date of the very first programming project in fall 2008 for the initial Scrabble game methods previously detailed. One section of the course shrank from 9 to 4 students, and the other section from 12 to 9 students, before the due date and before any grading. The author's lack of familiarity with student work habits was to blame. The author allowed three weeks for students to complete their work, anticipating about two weeks of work being required for most students. Unfortunately, most students waited until the final week to begin working on the project, at which time they realized that they could not complete it. Rather than fail a substantial project in what promised to be a course with substantial work, many students dropped the course. The skew between sections is explained by the fact that a larger percentage of students in the more heavily impacted section were sophomores who were not used to the demands of large programming projects.

The solution to this pitfall was to make the first programming assignment more modest in the second semester, to allow less time for its completion, and to provide an exact solution to the problem to be solved – depth first traversal of a Scrabble board's state – in C++ to be used as pseudocode. The author also supplied example code unrelated to Scrabble, written in both C++ and Java, that used the programming constructs required to complete the first assignment. This code served as a Rosetta Stone for students. Mapping from the C++ “pseudocode” to the Java solution could be accomplished by comparing the C++ and Java constructs on the “Rosetta Stone” and then using the appropriate Java constructs. The potential pitfall was averted. In fact, many of the students who had withdrawn from the first semester course completed the second semester without any substantial problems.

The spring 2009 course sections had much lower attrition numbers. One section dropped from 9 to 8 students, and the other from 13 to 9. The course continued to require substantial work from students. Near the semester's end it became clear that it would not be possible to complete the final programming assignments in graphical interface construction without running into final examination week. Rather than water down the project, the author cancelled the exam, replacing it with the final project. We used the two hour final exam time slot for a spirited group testing and debugging session. The subsequent success in deploying the software as a music

synthesis instrument confirms the validity of cancelling an exam in favour of completing the programming project.

Smaller pitfalls occurred at an individual level. In discussions with students and in reading student evaluations of the instructor and the course, the following complaints were made.

Some students did not like reading code and integrating their code into a framework written by someone else. They were used to writing their own assignments completely, based on their experience in the freshman computer science sequence. Fortunately, these students were willing to express this opinion. However, requiring students to read and comprehend existing framework code and documentation is an important dimension of this approach. This requirement gets students ready for their professional working conditions.

A few students felt that the framework dominated their thinking about Java, and expressed a desire for one small “throw-away” programming assignment. Comprehending the framework implementation in detail is optional. It does not contribute to a student's grade. As the framework continues to grow, the plan is to constrain student exposure to portions of the framework that impinge directly on project requirements, providing only the most general overview of the entire framework at the outset, and revisiting this matter later in the course. A few students may never come to understand the framework in a substantive way. That understanding is optional, but it is valuable to make available to the majority of students who can avail themselves of the benefits of exposure to the framework. Exposure to object-oriented constructs in subsequent courses is likely to alleviate this problem for this minority of students, since with repeated exposure they are likely to understand.

Conversely, many students cited working on a realistic application framework as a strength of the course. Work in progress for spring 2010 has students writing one complete program consisting of a re-usable module and a test driver, in order to satisfy this need felt by some and to provide an integration path into group projects. The second project entails integrating this first module into the surrounding framework. We have continued using C++ pseudocode and C++ to Java “Rosetta Stone” examples to help with the initial transition.

Finally, one student in the second semester expressed dissatisfaction at being required to learn to use Java's MIDI code library, considering MIDI to be too far from mainstream applications. While it is true that computer music generation is not a typical application for most undergraduate or industrial Java programmers, it is also true that we have created a novel, interesting product that we can deploy. The instructor did not mention the fact that MIDI is one of the primary mechanisms for creating and storing ring tones in modern mobile phones. Mobile telephony is a mainstream application space! We will integrate a brief overview of this application space into the spring 2010 course.

Given the fact that the author initiated this program without having previously taught the Java Programming

course, and the fact that outcomes for this course have not yet been measured for upcoming ABET accreditation [30] of Kutztown's computer science program, there are no formal assessment rubrics to report at this time. However, enrollments for the two sections of the course in spring 2010 are full with no student withdrawals at midterm, we are designing and constructing two very useful features – spelling checking and multi-level undo/redo with a graphical user interface – that enhance both the game and the game-to-music generation capability, and we are putting our software into the field. We have two ensemble performances planned for April, students from prior semesters are requesting copies of our enhanced software, and in March the International Computer Music Conference (ICMC2010) accepted a research paper by the author on generating musical structures from linguistic structures in Scrabble [31]. It is highly unusual for sophomores to be contributing code to a research project being presented at the premiere conference in its field. This contribution by undergraduates has been made possible by the modular, extensible design of this framework and by the fact that this framework aggregates collectively across semesters. This work is a collective effort of the instructor and students across several years. If assessment includes considering delivery of useful, stable software with ongoing feature enhancements, then this project can be assessed as a success.

3. Conclusion

Structuring the Java Programming course at Kutztown University to work as an apprenticeship in object-oriented software construction using Java has been an incremental process entailing a lot of work on the part of the instructor and students. We cannot simply follow textbooks and download isolated assignments and solutions. In order to achieve an apprenticeship environment, we need to create something novel. Creating and deploying novel software entails hard work.

This approach is meeting the seven goals of embedding Java programming in an object-oriented framework and an apprenticeship-like working environment as already detailed. One aspect of an apprenticeship approach that has been missing so far is multiple-student collaboration on at least one group project. Informal collaboration arose naturally near the end of the spring 2009 semester as students became familiar with the mechanisms and problems of mapping a Scrabble game to MIDI music. The author plans to incorporate group collaboration into a project near the end of the spring 2010 semester.

The author also plans to extend engineering of this framework to a fall 2010 graduate course at Kutztown University, CSC 520, Advanced Object Oriented Programming. Graduate students will serve as architects for the system. Success stories and pitfalls remain to be realized, but it is the author's opinion that application of object-oriented design concepts across several courses at several levels of difficulty within the curriculum is the best way to apply the methods of the current approach. Some of

the students taking this graduate course will have worked on this project as undergraduate Java programming students, and the author is looking forward to helping these students learn to steer the framework that they previously helped to build and power.

This approach is clearly not a typical approach to teaching academic computer science, but it is a typical approach to industrial software engineering for small to medium size projects. It is the author's hope that by integrating this approach into a few courses inclined towards object oriented analysis, design and programming, it will give students powerful new perspectives on how to attack problems in computing, how to create unique software solutions, and how to prepare themselves for professional computing.

References:

- [1] D. Parson, D. Murray and Y. Chen, Object-Oriented Design Patterns for Debugging Heterogeneous Languages and Virtual Machines, *Software—Practice and Experience* 35(3) (March, 2005), 255-279.
- [2] D. Parson, B. Schlieder and P. Beatty, Extension Language Automation of Embedded System Debugging, *Automated Software Engineering* 9(1) (January, 2002), 7-39.
- [3] D. Parson, L. Herrera-Bendezu and J. Vollmer, Distributed Source Code Debugging for Embedded Systems, *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, CSREA Technology Press, Las Vegas, June, 2000.
- [4] D. Parson, U.S. Patent 6053947, Simulation Model Using Object-Oriented Programming, April 25, 2000.
- [5] D. Parson, P. Beatty, J. Glossner and B. Schlieder, A Framework for Simulating Heterogeneous Virtual Processors, *Proceedings of The 32nd Annual Simulation Symposium*, IEEE Computer Society & Society for Computer Simulation International, April, 1999.
- [6] James Rumbaugh, Ivar Jacobson and Grady Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
- [7] Y. Daniel Liang, *Introduction to Java Programming*, Seventh Edition, Comprehensive Version. Pearson / Prentice Hall, 2009.
- [8] Stuart Reges and Marty Stepp, *Building Java Programs: A Back to Basics Approach*. Pearson / Addison-Wesley, 2007.
- [9] Paul Deitel and Harvey Deitel, *Java: How to Program*, Eighth Edition. Pearson / Prentice Hall, 2010.
- [10] Ralph Bravaco and Shai Simonson, *Java Programming: From the Ground Up*. McGraw-Hill, 2010.
- [11] John Dean and Raymond Dean, *Introduction to Programming with Java: A Problem Solving Approach*. McGraw-Hill, 2008.

- [12] Joyce Farrell, *Java Programming*, Fifth Edition. Boston, MA: Course Technology, 2010.
- [13] John Lewis and William Loftus, *Java Software Solutions: Foundations of Program Design*, Sixth Edition. Pearson / Addison-Wesley, 2009.
- [14] D. S. Malik, *Java Programming: From Problem Analysis to Program Design*, Fourth Edition. Boston, MA: Course Technology, 2010.
- [15] C. Thomas Wu, *An Introduction to Object-Oriented Programming with Java*, Fifth Edition. McGraw-Hill, 2010.
- [16] Computer Science Department, Kutztown University of PA, courses, <http://cs.kutztown.edu/courses.aspx>, January, 2010.
- [17] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering*, Second Edition, Addison-Wesley, 1995.
- [18] Jean Lave and Etienne Wenger, *Situated Learning: Legitimate Peripheral Participation (Learning in Doing: Social, Cognitive and Computational Perspectives)*, Cambridge University Press, 1991.
- [19] Allan Collins, John Seely Brown and Susan E. Newman, *Cognitive Apprenticeship: Teaching the Crafts of Reading, Writing and Mathematics, Knowing, Learning and Instruction, Essays in Honor of Robert Glaser*, Lauren B. Resnick (ed.), Lawrence Erlbaum Associates, 1989.
- [20] Seymour Papert, *Mindstorms: Children, Computers, And Powerful Ideas*, Second Edition, Basic Books, 1993.
- [21] Seymour Papert, *The Children's Machine: Rethinking School In The Age Of The Computer*, Basic Books, 1994.
- [22] Virginia Tech Learning Technologies, *Cognitive Apprenticeship*, January 2010, <http://www.edtech.vt.edu/edtech/id/models/cog.html>.
- [23] Bertrand Meyer, *Object-Oriented Software Construction*, Second Edition. Prentice Hall, 2000.
- [24] MIDI Manufacturers Association, <http://www.midi.org/>, January 2010.
- [25] MIDI Technical Fanatic's Brainwashing Center, <http://www.blitter.com/~russtopia/MIDI/~jglatt/>, January, 2010.
- [26] Sun Microsystems, *Java Platform documentation for javax.sound.midi*, January, 2010, <http://java.sun.com/javase/6/docs/api/index.html>.
- [27] G. Cuyler, "KU students spell ingenuity: musical Scrabble," *Reading Eagle*, Reading, PA, Oct. 2, 2009, reprinted in McClatchy-Tribune Info Services <http://www.tmcnet.com/usubmit/2009/10/02/4403193.htm>.
- [28] Dale Parson, two musical performances of Scrabble-to-MIDI on <http://radio.electro-music.com>, at http://electro-music.com/forum/phpbb-files/em_ss09_20june2009_parson_132.mp3 and http://electro-music.com/forum/phpbb-files/dparsonnye20092010_701.mp3.
- [29] Pennsylvania Association of Computer and Information Science Educators, <http://www.pacise.org/>, Annual Conference, April, 2010.
- [30] ABET, *Leadership and Quality Assurance in Applied Science, Computing, Engineering, and Technology Education*, <http://www.abet.org/>, March, 2010.
- [31] ICMC 2010, *The International Computer Music Conference*, <http://www.icmc2010.org/>, June, 2010.