minimUML: A Minimalist Approach to UML Diagramming for Early Computer Science Education

SCOTT A. TURNER, MANUEL A. PÉREZ-QUIÑONES, AND STEPHEN H. EDWARDS
Virginia Polytechnic Institute and State University

In introductory computer science courses, the Unified Modeling Language (UML) is commonly used to teach basic object-oriented design. However, there appears to be a lack of suitable software to support this task. Many of the available programs that support UML focus on developing code and not on enhancing learning. Programs designed for educational use sometimes have poor interfaces or are missing common and important features such as multiple selection and undo/redo. Hence the need for software that is tailored to an instructional environment and that has all the useful and needed functionality for that specific task. This is the purpose of minimUML. It provides a minimum amount of UML, just what is commonly used in beginning programming classes, and a simple, usable interface. In particular, minimUML is designed to support abstract design while supplying features for exploratory learning and error avoidance. It supports functionality that includes multiple selection, undo/redo, flexible printing, cut and paste, and drag and drop. In addition, it allows for the annotation of diagrams, through text or free-form drawings, so students can receive feedback on their work. minimUML was developed with the goals of supporting ease of use, of supporting novice students, and of requiring no prior training for its use. This article presents the rationale behind the minimUML design, a description of the tool, and the results of usability evaluations and student feedback on the use of the tool.

Categories and Subject Descriptors: K.3.2 [**Computers and Education**]: Computer and Information Science Education - *Computer science education*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces - *User-centered design*, *Interaction styles*; D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering (CASE), Object-oriented design methods*

General Terms: Design, Human Factors

Additional Key Words and Phrases: UML, human-computer interaction, education, learning, minimalist design.

1. INTRODUCTION

The Unified Modeling Language (UML) is commonly used in introductory computer science courses as a way to introduce students to object-oriented design. Instructors present UML as a way to talk about design problems and to organize solutions. In turn, the students create a few diagrams to show their high-level concept of a system. In this article we present the design, evaluation, and experiences in using a minimalist tool we call minimUML to support the use of UML early in the CS curriculum.

1.1 Motivation

In choosing a tool to support UML activities early in the curriculum, we must ensure that it meets the educational needs of students at their level. It must provide at least a subset

Author's address: Department of Computer Science (0106), Virginia Polytechnic Institute and State University, Blacksburg, VA 24061; email: perez@cs.vt.edu

of UML that can solve the typical types of design problems in introductory computer science classes. It must provide for abstract and iterative design, support exploration during the design process, and supply an interface that is simple enough to be used during a class or a lab with only a brief introduction. Our goal is to evaluate existing systems against these requirements and to create a tool that will adequately meet them.

Surprisingly, there is a lack of suitable software to support this task. While there are numerous commercial and free systems that have UML diagramming capabilities, they are, in general, too complex for students at an introductory level [Buck and Stucki 2000]. Many of these systems are focused on creating well-defined diagrams and on code generation. There is an assumption that the user is already familiar with good design principles.

Typically, a very small subset of UML is used in the introductory classes. Many textbooks and instructors focus on class diagrams that contain very little detail but show the relationships between the objects through aggregation, association, and inheritance [Fowler and Scott 1997; Lee and Tepfenhart 2001; Barnes and Kolling 2003; QuickUML 2003; Riley 2003; Horstmann 2004; Lewis and Chase 2004]. Since this is the goal, there is little need for tools to support the full UML language. In fact, Alphonce and Ventura [2002; 2003] and Crahen et al. [2002] suggest that limiting the amount of UML taught in this situation is helpful in learning design.

Ease of use becomes an important goal if a UML diagramming tool is to be useful early in the Computer Science curriculum. Features for exploratory learning and error avoidance are needed so that the student can try out different ideas without being penalized. To help do so, the tool should at least support full undo/redo so students can feel free to experiment with alternative ideas. Keeping the interface simple encourages students to focus on learning design instead of how to use the tool.

The tool must support transferring UML designs into another work environment. At the very least, it should support flexible output features, as students might need to print their work or export it in a graphical format (e.g., PNG or JPG) so that it can be included in a report or used in a presentation. Students also might want to transfer their work to other programs (e.g. text editors, compilers, etc.), by converting the UML designs into code. Since, in the early CS curriculum, Java and C++ are the most commonly used OO languages, it is appropriate to provide support for both. In some introductory courses at Virginia Tech, students turned in diagrams created with ASCII art, boxes and arrows in Word, and images created in a paint program or scanned. Considering the complexity of the design solutions, the students probably spent more time formatting the diagrams than designing the solution.

The tool should provide support for feedback and comments within the tool itself. Instructors or peer-reviewers often provide comments to other students, so the tool should have some annotation features [Anderson and Shneiderman 1977; Sullivan 1994; Zeller 2000; Gehringer 2001]. Finally, the tool should support abstract design, not coding, by not requiring students to write Java or C++ in their design diagrams [Alphonce and Ventura 2002]. There is no urgency in enforcing proper class names, specification of return types, and so on; these are details to be worked out later in the design of the system. Tools that enforce these up front are inappropriate in the early stages of the design.

This article presents minimUML, the rationale behind its design, a description of the tool, and the results of usability evaluations and student feedback. We first define the requirements of an UML tool designed for early CS education. Then we review five currently available UML diagramming tools for their usefulness in this particular

educational setting. Next, we present minimUML, our solution to the lack of educationally appropriate UML diagramming tools, and describe its features. We present the results of a usability evaluation to improve the design of minimUML and results of a survey of students on the use of minimUML, their design approaches, and the amount of time they spent on learning to use it. Finally, we discuss future work on the tool and draw conclusions from our work.

2. REQUIREMENTS

When designing a tool it is important to consider the characteristics of the people it is for and the situations it will be used in. In general, first-year computer science students do not have much experience in object-oriented design. Some are accomplished coders, but may have little knowledge of how to structure programs properly; others have no programming skills at all. The majority have little or no idea what UML is.

As part of the course, students may be asked to solve relatively simple design problems by making UML diagrams of a handful of classes [Lee and Tepfenhart 2002;

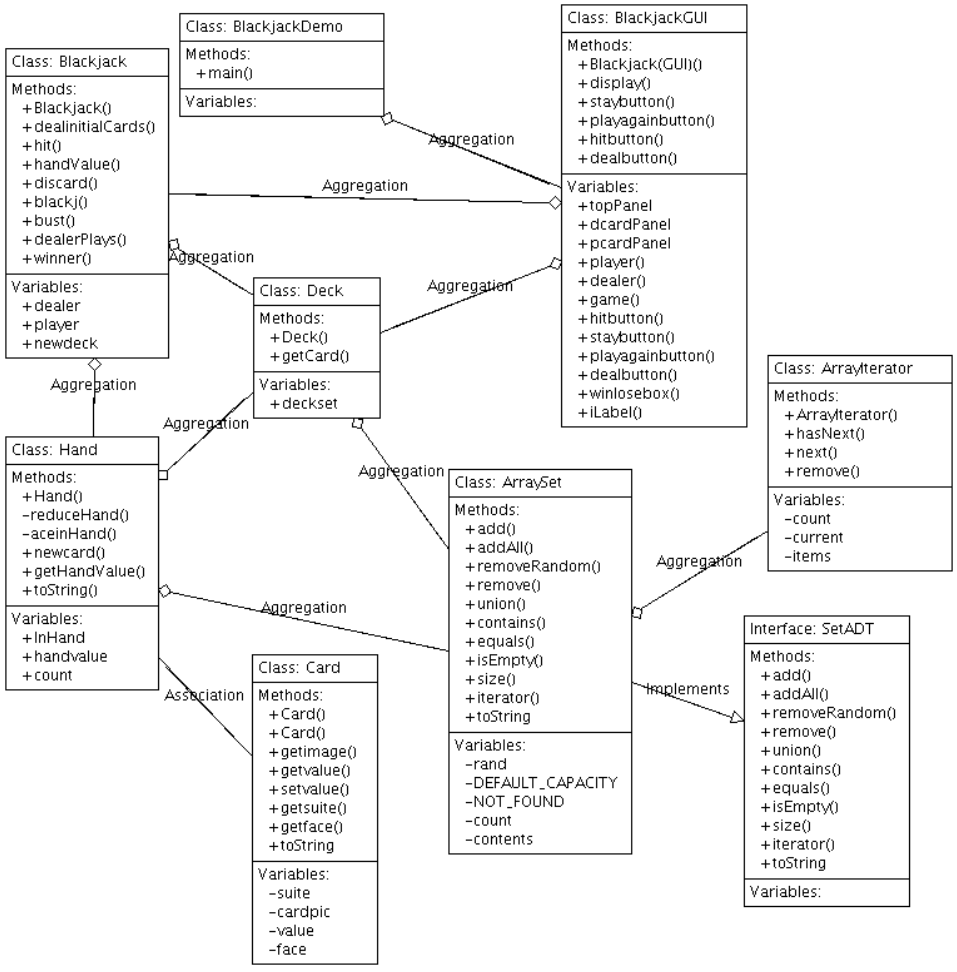

Fig. 1. BlackJack game design from the Lewis and Chase textbook [2005, p.149] as drawn with minimUML

Horstmann 2004; Lewis and Chase 2005]. A typical problem can be found in Lewis and Chase [2005, p.149], shown here in Figure 1 as redrawn with minimUML. To prepare students for these assignments, instructors may spend part of a couple of lectures discussing UML and on how students should represent their designs. The students are then are set loose to complete their work as homework or as part of a lab session. They are faced with the three tasks of quickly learning how to structure a program, how to represent it in UML, and how to operate any tool required for the assignment. This could be quite overwhelming. A good tool should try to limit the amount of effort that goes into the second and third tasks to allow more focus on the first. It should also provide a variety of features that can handle common classroom activities.

In this setting, the goal is not to teach UML, but to teach the basics of object-oriented design with UML. The *entire complexity of UML* is not needed to do this. In fact, very little UML is needed to solve the design problems that the students will face. The solutions typically only involve classes and a few types of relationships between those classes. Nothing else is needed. Here we are making an argument similar to the one used by QuickUML.

It is also important that the tool be useable in a classroom where students have had little or no previous instruction. The interface should be as simple and easy to use as possible and supply the common functionality users expect from programs, such as cut and paste and especially undo/redo. Moreover, students should to be able to try out ideas for their designs without penalty and to be able to design in their own way. No restrictions should be imposed on the design by the syntax of a particular programming language.

Support for exploration is another needed component. It is not necessary to force students to make early commitments; for instance, a student may create a class object and later realize that is should have been an interface. There is little reason to make students explicitly choose a class or an interface when they may only know that they need some kind of object that supports some set of methods. The types of connections between classes are a similar issue. Once again, it is not necessary to insist that the students commit to a particular type of connection when it is created only to be deleted and then to start again if it was the wrong type. A good tool should allow students to make tentative choices and help them make changes without wasting time and effort.

The tool should provide a number of other capabilities to support classroom activities. Diagrams will need to be submitted in some fashion. Good printing features and ways to export the designs into standard formats are necessary; support for electronic submission would also help here.[1] The diagrams will be reviewed by the instructor or by other students, so methods for annotating the assignment should be included. Finally, the creation of the designs could be the first step in a programming project, so the ability to export the classes as code should be included as well. Even better would be to let the student modify both the diagram and the generated code concurrently, so that they can work at different levels of abstraction throughout the entire project without having to be maintained individually.

Together, these features make the tool suitable for the target audience and its learning environment.

[1] minimUML does not currently support electronic submission, but it is being planned for a future release. The latest version of minimUML is available at http://perez.cs.vt.edu/minimuml/.

3. RELATED WORK

In examining diagramming programs for educational work, we found five UML tools, Violet [2003], UMLet [2004], Dia [2003], ArgoUML [2003], and QuickUML [2003] (URLs available in the reference section.). We evaluated them based on the subset of UML they support, the simplicity of the user interface, support for exploratory learning, such as undo/redo facilities, the amount of interface consistency, and the degree to which it allows the user to avoid making errors.

We compared the UML tool subset to the subsets in some of the computer science textbooks intended for introductory object-oriented design. Specifically, we examined the case studies in Horstmann's *Object Oriented Design & Patterns* [2004] and Lewis and Chase's *Java Software Structures: Designing & Using Data Structures* [2004] for examples of typical design problems and the UML used to solve them. Lewis and Chase provide an overview of UML (pages 13 to 17) that describes classes and a few types of relationships between classes. The description provides information about marking classes as abstract or as interfaces and about the formatting variables and methods with their types and parameters. It is interesting to note that although the formatting information is there, most of it is not used in the book's diagrams and case studies. The authors show classes that contain only variable and method names and little other information (e.g., Lewis and Chase [2005, p.149]. This, presumably, is to allow the student to get an idea of high-level design without having to deal with all of the details. Horstmann provides a similar overview and similar diagrams in terms of complexity. This strengthens our claim that it is not necessary to support all of the features of UML, or even all of the formatting options for specific parts of UML, to have a tool that is adequate for learning early in the curriculum.

When appraising how a tool allows a user to avoid errors, we looked for instances where actions could lead to a semantically invalid state or to situations that were difficult or impossible to recover from. (Does deleting a class leave arrows that point at nothing? Can objects be lost off the screen?)

3.1 Comparisons

To better understand the strengths and weaknesses of these UML tools, we compared them across a number of dimensions. The summary of our evaluation is found on Table I below. As mentioned earlier, we evaluated the simplicity of the interface and the amount of undo/redo support provided. We also compared the methods for exporting designs into other formats such as Java or C++ code or as image files. The availability of annotation tools and the possible locations of connections (e.g., between objects or anywhere on the screen) were other aspects evaluated by us. Two of the most important features we looked at were the focus of the tools and the amount of UML they provide, as these features help us gauge their use in learning design, as opposed to creating design. In addition, we noted the presence of features for zooming, multiple selection, printing, and extending the interface. The final column on Table I shows what minimUML, the tool we created to meet the requirements of early computer science education, can do; minimUML is discussed in detail in Section 4.

As can be seen from the table, the tools cover a wide range of functionality. In general, those focused on design have simpler interfaces but lack some of the nicer features, most notably undo/redo, that the more complex programs provide. Conversely, those that are not focused on design and are more general diagramming or productivity tools have more features but do less to support the user in creating valid, abstract diagrams by, for instance, allowing connections to be drawn anywhere or by requiring

Table I. Tool Comparisons

	Violet	UMLet	Dia	ArgoUML	QuickUML	minimUML
Simplicity of Interface	Simple	Complex	Complex	Complex	Simple	Simple
Undo/redo	None	Full	Some	None	None	Full
Printing	To postscript	To PDF	Simple	Simple	Flexible	Flexible
Save as image	Yes	Yes	Yes	Yes	Yes	Yes
Save as code/text	No	No	No	Yes (Java)	Yes (Java/C++)	Yes/Limited (Java/C++)
Annotation tools	Yes	Yes	Yes	Yes	Yes	Yes
Focus on design	Yes	No	No	No	Yes	Yes
UML covered	Most	Most	Most	Most	Limited	Limited
Zoom	Yes	No	Yes	Yes	No	Yes
Connection locations	Anywhere	Anywhere	Anywhere	Between classes	Between objects (includes notes)	Between classes
Multiple selection	No	Yes	Yes	Yes	Yes	Yes
Extensibility	No	Yes	Yes	No	No	No

values to be well- specified. (Note that improvements have been made to several of these tools since they were evaluated. For instance, QuickUML is now an Eclipse plug-in called Green [2005] with new features, including round-trip editing.)

Violet had a wonderfully simple interface but no features for exploratory learning or ways to avoid errors. It did not have undo/redo, so it was harder to recover from mistakes and it permitted actions that are not logical, such as connections that do not connect anything. It was also missing many features, like multiple selection or better print options that would make it more convenient to use. UMLet allowed exploratory learning and for expanding and adapting the user interface, but suffered from an extremely poor user interface. It required users to enter specialized codes to format the objects, such as setting the direction of an arrow on a connection, and was missing some very basic UI affordances like scrollbars for the diagramming area. Dia and ArgoUML provided a great amount of functionality, but their complexity made them difficult to use. Dia, a more general-purpose diagramming tool, allowed the user to control a large number of settings, such as color and line thickness for each object, which made the interface very complex. Since Dia did not focus solely on UML, it did not provide the user with any support for creating valid diagrams. ArgoUML, on the other hand, ensured valid diagrams by requiring users to make Java-specific design decisions early on; but also suffered from a very cluttered interface. QuickUML attempted to meet the requirements of an educational tool, but was missing too many features. Its supported subset of UML and its focus were right for academic use, but its interface was found lacking. Like Violet, it lacked undo/redo support. It also allowed the user to make mistakes from which they could not recover. For instance, objects could be moved off the screen in such a way that they could not be retrieved. The lack of a suitable UML tool for an educational context motivated the creation of one that could meet these goals.

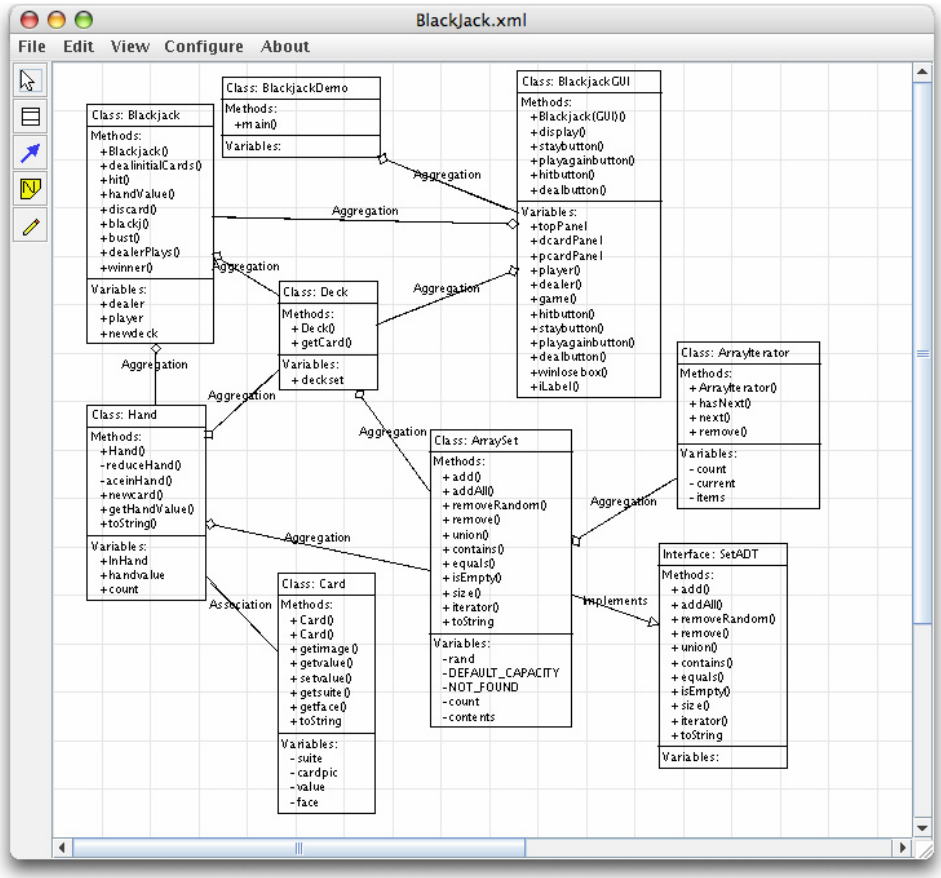

Fig. 2. minimUML interface.

4. MINIMUML

minimUML is a minimalist approach to UML diagramming [Carroll 1997]. Its purpose is to focus the user on creating object-oriented design and not on learning the tool. Learning how to use the tool comes as the users perform real tasks -- in this case, diagramming.

minimUML provides support for classes, connections between classes, and two types of annotations, which should be sufficient for the target audience and for the general types of problems, as described above. To ease the design process, a number of other features such as undo/redo, cut and paste, and drag and drop are supplied by the program. The full interface is shown in Figure 2.

4.1 Classes

Classes provide space for the name of the class, methods, and variables. There are few restrictions imposed on these values other than that the class must be named and that the name must be unique to the diagram. While we could have enforced a particular coding style or language syntax, it would limit the usefulness of the program to do so. Since both C++ and Java are commonly used to teach OO design, it is not prudent to support one over the other. In addition, this kind of enforcement takes time away from the design

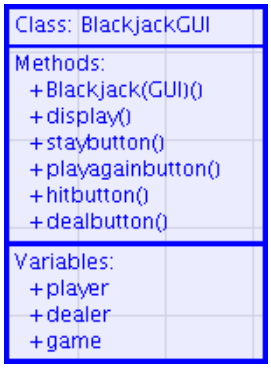

Fig. 3. BlackJackGUI class shown in its selected view

process: if the goal is to create a diagram, then stopping to add a comma or a semicolon is counterproductive. The compiler will catch these problems later as the design is implemented. This has the fortunate side effect of allowing the user to design the class in iterative steps. Variables and methods can be defined vaguely in the beginning and fleshed out as the design coalesces. There is no need to have the types and number of parameters well defined, as items are added to a class. Users are allowed to explore ideas and work in their own way without many restrictions. Figure 3 shows the selected view of class in minimUML.

When a class object is created, the user is given the opportunity to name the class and provide information about the variables and methods. The class is given a default name if one is not supplied. Double-clicking the class object reopens it for editing. The text areas used in the class (class name, methods, and variables) support cut/copy/paste and drag and drop. Figure 4 shows a class being edited. Movie 1 shows an example of how a class is created.

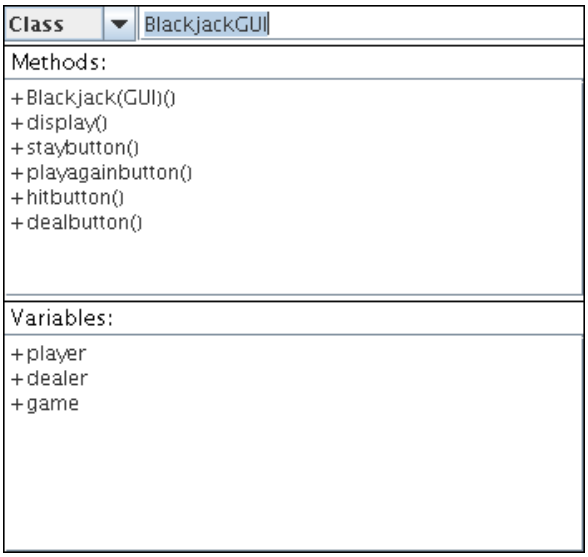

Fig. 4. BlackJackGUI class in edit mode

4.2 Connections

Connections are simple arrows connecting two classes and are dependent objects which must have a source and a destination. As such, the deletion of one of those classes deletes the connection as well. In minimUML, connections only have semantic meaning when they join two objects, so we improve the user's ability to create correct diagrams by only allowing connections to exist when both classes exist. A connection can be given one of four values: generic, association, aggregation, and inheritance. The position of the label and the line itself is automatically determined and is updated as classes are moved about. As connections are drawn, they default to the generic value and can be changed to different values later. This may benefit novice users who may understand that a relationship is needed, but not the type that is needed. Our approach is opposed to a style in many other UML diagramming tools, where separate connection objects are provided for each type. Besides cluttering the interface with extraneous buttons or menu options, it seems to be very unnecessary. The differences between these objects are their appearance and a little internal, programmatic state, so separating them and frequently not permitting conversions between the various types makes little sense. Such a strategy may be useful for those who know exactly what they are doing, but does not help those who are trying to explore ideas. The approach taken by minimUML allows for a dependency to be defined and then lets its nature be determined and redetermined as the design progresses.

Connections are created by simply selecting the connection tool and dragging from the interior of one class to the interior of another. Figure 5 shows a connection between two classes. Like the class object, double-clicking the connection allows the object to be

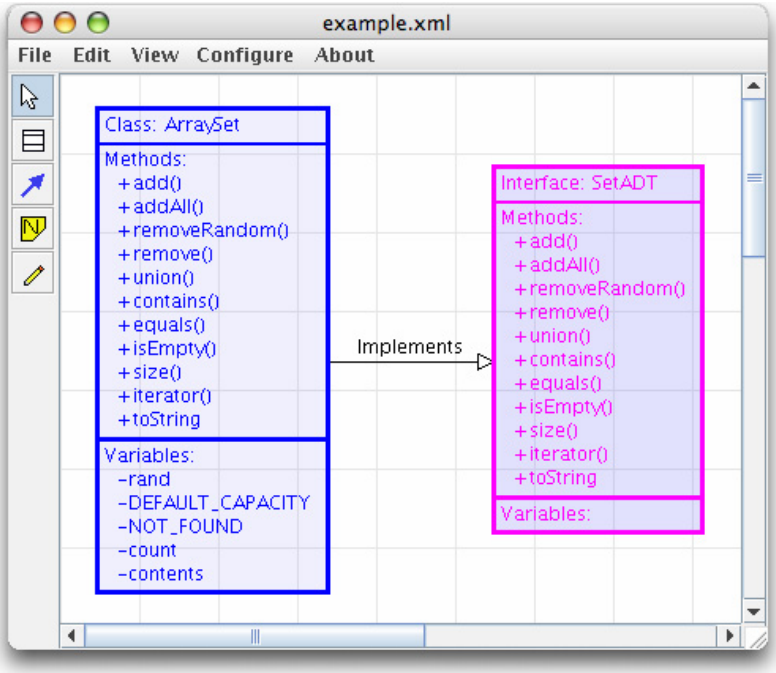

Fig.5. Connection between classes.

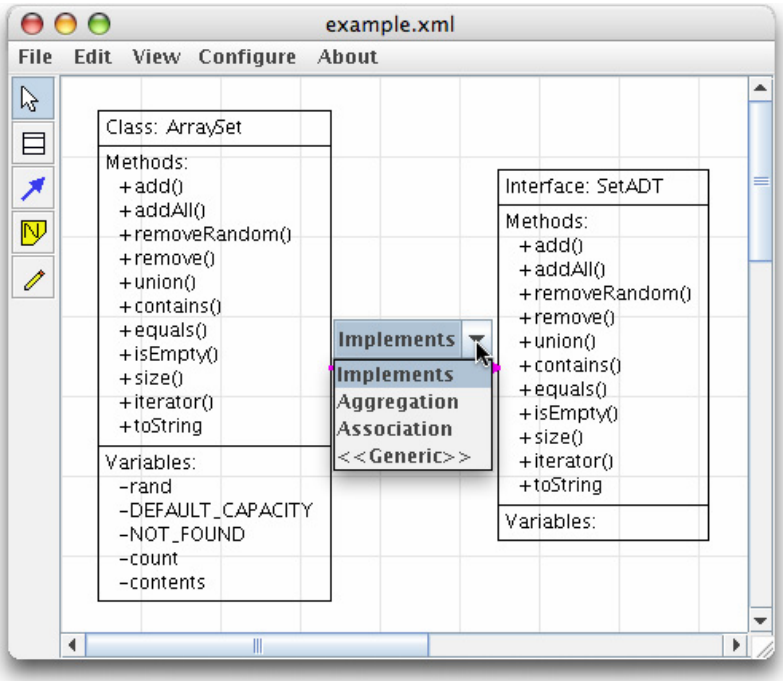

Fig. 6. User editing a connection.

edited via a simple dropdown list. Figure 6 shows a connection while its type is being edited. Currently, there is no way to change the direction of the connection; the student must simply delete the connection and create it in the other direction. Movie 2 shows an interactive example of how a connection is created.

4.3 Annotations

minimUML supports two types of annotations: electronic sticky notes and free-hand drawings, referred to here as *glyphs*. The electronic sticky notes are areas of typed text that automatically resize to show their contents. These two annotation devices provide minimUML a great amount of flexibility. Graders can mark up diagrams using a keyboard or a pen device. Students can use the annotation tools to complement tool functionality by typing or drawing in the needed information.

Sticky notes are created in a manner similar to classes. Choosing the note tool and clicking in the diagramming area gives the user a text area where an annotation can be typed (Figure 7). By default, a sticky note is displayed as an icon in the diagram until it is selected or moused over; but it can be modified so that it is always shown (see Figure 8). Holding down the ALT key (option key on the Macintosh) expands all of the note icons to permit easy browsing. Sticky notes can be edited and moved around to different locations in the UML diagram. Finally, the text area supports cut and paste and drag and drop.

Glyphs are free-form annotations intended to mimic pen-based markings but done with the mouse. They are designed to support any kind of annotation that may be required, such as striking out or circling a section of a UML diagram (see Figure 9).

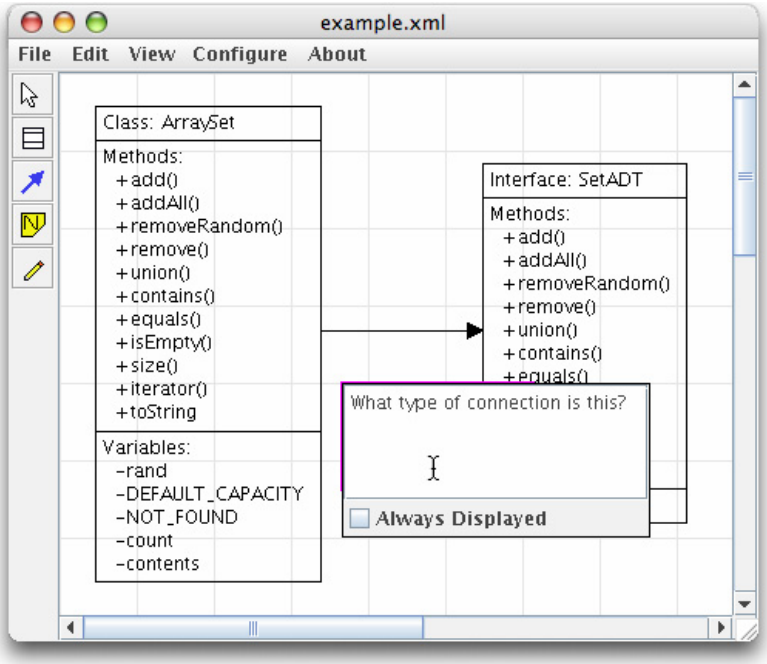

Fig. 7. Creating an electronic note.

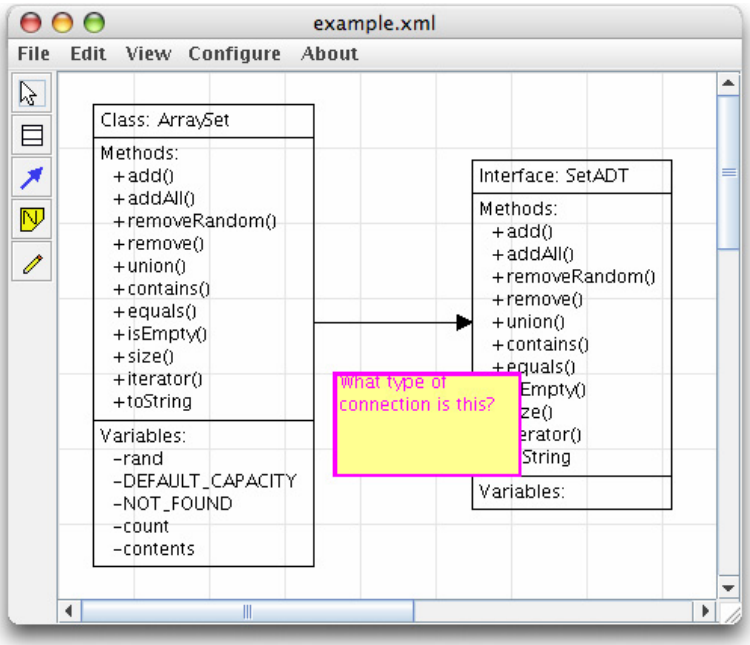

Fig. 8. An electronic note shown in its expanded form

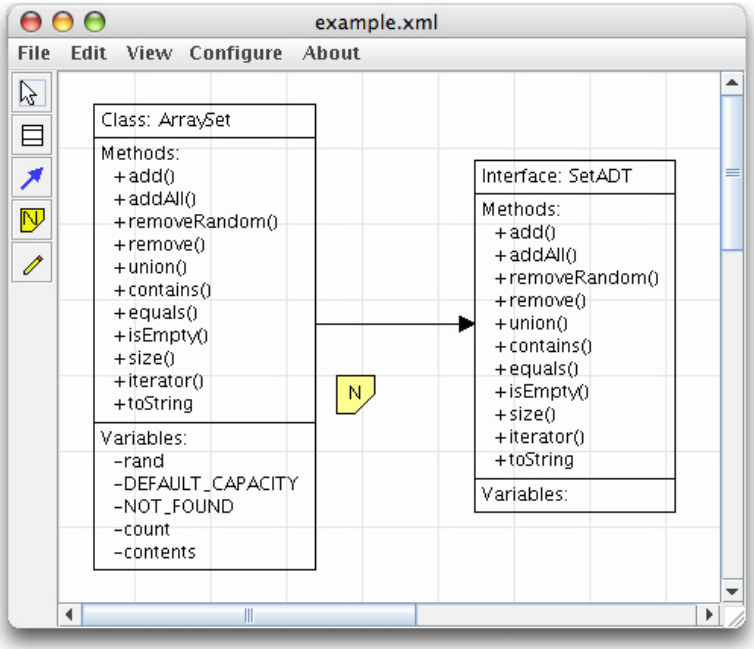

Fig. 9. An electronic note in its collapsed form

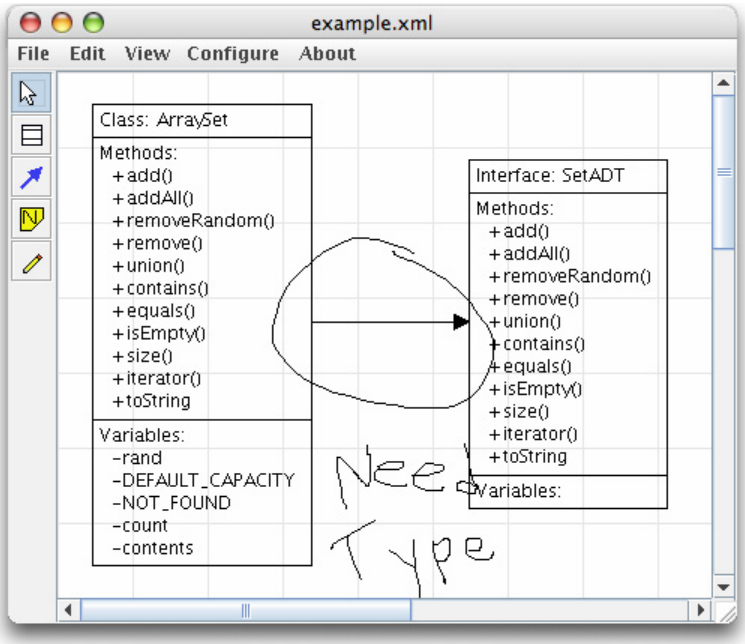

Fig. 10. Use of glyph to provide comments.

Glyphs are made by dragging the cursor across the screen with either a mouse or a stylus. They can be selected, moved, deleted, and otherwise acted upon like any of the other objects, but the glyph's shape cannot be modified once it is created. If a mistake

was made then the glyph needs to be erased and redrawn. Figure 10 shows comments written with the glyph tool. Movies 3, 4, and 5 show these two annotation tools at work.

4.4 Undo/Redo

In addition to the four diagramming objects, minimUML provides a number of other useful functions. In an effort to support exploratory learning, the tool provides undo/redo capabilities for all actions. There is no set limit on the depth of the undo stack, so a user can go back to the point when a window was first opened. This feature is associated with a window rather than the current data file, so it allows the undoing of the loading of a file. Standard shortcut keys are associated with the undo and redo actions. Movie 5 shows an example of how the undo command works. When a class is deleted, the connections to and from that class are also deleted. When the deletion is undone, even the connections are restored; an example is shown in Movie 5.

4.5 Clipboard Support and Drag and Drop

Clipboard support (cut/copy/paste) and drag and drop can be used to move objects among minimUML windows and other windows in the operating system. The standard keyboard shortcuts are supported for the cut/copy/paste commands.

In the basic form of clipboard support, minimUML supports copying of any selected object (except connections) and pasting it in a different location in the same window or in a different one. When multiple classes are copied, all the connections among the classes are copied also. Movie 6 shows an interactive example of several classes being copied to another minimUML document. The movie shows how connections are automatically copied when its two connecting classes are copied.

minimUML makes use of Java's DataFlavor classes, thus supporting multiple data types in the clipboard. Classes copied or cut into the clipboard can be pasted in other applications in either text or graphical form. Pasting data from minimUML into a text editor (see Movie 7) produces a Java source code equivalent of the UML objects copied (see the next section for more information on code generation). Pasting data into a graphical editor produces a graphical representation of the copied UML objects (see Movie 8). These provide support to transfer portions of the UML design to other applications for writing reports, grading, or presentation.

Drag and drop support provides the same functionality as that provided by clipboard support. Classes can be dragged from one window to another within minimUML, and the effect is the same as copy/paste. Classes can be dragged to a text editor window with support for drag-and-drop and the source code for the classes will be inserted into the text editor. Classes can also be dragged into a drawing application and inserted in an image format.

4.6 Code Generation

To support the transfer of UML designs to development environments, minimUML supports simple code generation in C++ and Java, which is meant to provide a stepping off point for actually implementing the design rather than a way to produce reliably compilable code. There is no validation done on the syntax of the variables or methods. We include this mechanism as a way to help students visualize how their diagrams would actually be realized in code.

Code generation is made available in three ways. First, as an export of the full design; students can export the whole design as C++ or Java. Second, parts of the diagrams can be pasted into text editors. Third, parts of the diagram are converted to code as a result of a drop action into a text editor that supports drag and drop. The last two methods support

Java only; but the choice of Java over C++ was purely arbitrary. In a future version this will be controlled via a user-specified preference option.

4.7 Printing

This tool's printing abilities are flexible. Page boundaries can be shown in the diagram and can be oriented in landscape or portrait; the entire document, or an arbitrary rectangular subsection of the diagram, can be printed. To produce an overview, the whole design can be printed as a single page. As an aid for reconstructing the printed diagram, a small key at the bottom of each page shows its relative position to the other pages. Figure 11 shows the page boundary for a large diagram. If the document were to be printed at this stage, only the portion within the darker rectangle would be printed.

4.8 Other Features

Finally, we've included a number of simple, but sometimes forgotten, features to make this a more usable product. Multiple selection is done by dragging a selection box across the diagramming area. The CTRL key (shift key on the Macintosh) can also be used to toggle to select individual objects. Zooming is allowed in increments from 10% to 500% (see Movie 9 for an interactive example). Dragging a minimUML file into a minimUML window will open a new window containing the contents of the file.

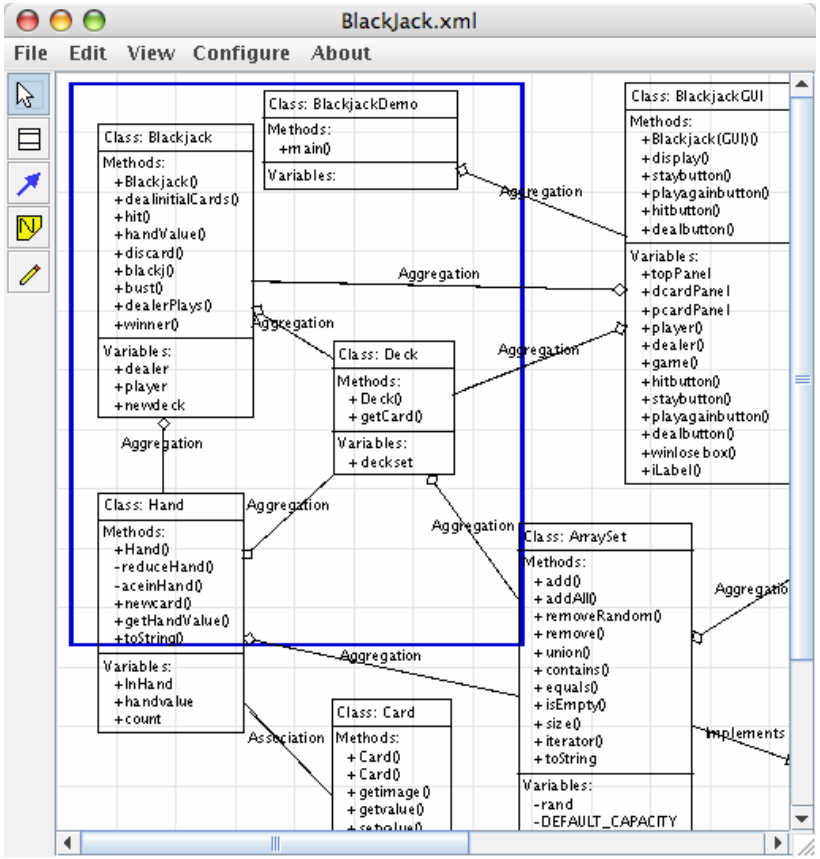

Fig.11. Printing a selected area.

5. EVALUATING MINIMUML

We used two different approaches to evaluate minimUML. First, we evaluated the general usability of the tool by using a typical usability evaluation with a small number of participants. Second, we asked students in a classroom, where they used the tool to prepare UML designs, to fill out a survey to assess how easy it was to use UML without prior training. This section describes both evaluations and their results.

5.1 Usability Evaluation

To validate the design of our tool, we ran a simple usability study with six participants. While this is not a large number, research suggests that it is adequate for this type of evaluation [Nielsen and Molich 1990]. The participants were computer science majors at Virginia Tech, ranging in age from 20 to 22, and were an even mix of juniors and seniors, all male. Most had little UML experience: three rated themselves as beginners, two as intermediate users, and one said he had no experience. Most of the students were from a usability engineering class; all were given a small amount of class credit for their participation. The students were recruited through class email lists.

To begin the session, the students were given a very brief introduction to minimUML, which was intended to be similar to one that an instructor would typically give in a lab session the first time students were introduced to a tool. In this case, the introduction lasted for about one minute and covered minimUML's high-level functionality and the purpose of the buttons on the toolbar. The participants were then given a few minutes to become familiar with the tool's interface. When they had finished they were given a simple task description detailing a movie rental system that they were assigned to diagram (see Appendix A). The problem was specifically designed to provide situations for the use of aggregation, association, and inheritance. A reasonable solution requires approximately ten classes.

The problem is very typical of first-year design assignments; it resembles, in size and complexity, a number of case studies from introductory textbooks. Horstmann [2004, p. 63] provides an example problem about a voice mail system. All told, the UML diagram in Horstmann's book contains six classes. Lewis and Chase [2005] give three additional examples of these types of design problems: (1) a Blackjack program with a 9 class diagram; (2) a diagram for a digital calculator with 11; and (3) a webcrawler consisting of 18 classes. Their general level of difficulty is roughly equivalent to our design problem.

In the interest of time, users were asked not to worry about adding any accessories or mutators to the diagrams. They were given 30 minutes to complete the exercise and were then given a questionnaire (see Appendix B).

Since one of the aspects being measured here is the ease with which the interface is learned, we deliberately gave students cursory instructions and limited the time they could spend looking at the tool before starting the task. Students were also told that they could not ask questions about minimUML during the session. We attempted to create a situation similar to a typical lab session in an introductory course.

5.2 Results

Five of the six students took the full 30 minutes to complete the task. The sixth student finished in approximately 20 minutes and spent the rest of the time exploring some of the tool's other features. The response to the program was generally positive; although a number of issues were raised, the participants found it very easy to use the program and seemed comfortable doing so.

As the students created their designs, there was some confusion over the meaning of buttons and how to interact with the objects (to be discussed later). Despite this,

everyone finished the task and no one appeared to have major difficulties learning the interface. This is encouraging, as it suggests that it is reasonable to introduce the tool and, in the same class or lab session, have the students use it.

On the questionnaire, questions 3 to 23 were 5-point Likert scales with 1 being strongly disagree, 5 strongly agree, and 3 neutral. Questions 15, 20, 21, and 23 were worded negatively, but were inverted in the data analysis for consistency (see Appendix C for full results).

5.2.1 *Usability Evaluation.* Questions 3 to 10 dealt with the usability of the tool for the task the students were given. The participants found it easy to create and modify both the classes and connections, with almost all of the ratings being agree or strongly agree.

Students were more ambivalent to the note and glyph tools; but this is not surprising as the task did not require their use. When asked about the other features of the tool, specifically undo/redo and zoom, students also responded favorably. Overall, they seemed to agree that the tool was adequate for the assigned task. Figure 12 shows a graph with the results.

5.2.2 User Interface Evaluation. The responses to questions 11, 12, and 13 relate to the tool's interface (see Figure 13). Participants found it easy to learn and use, but were

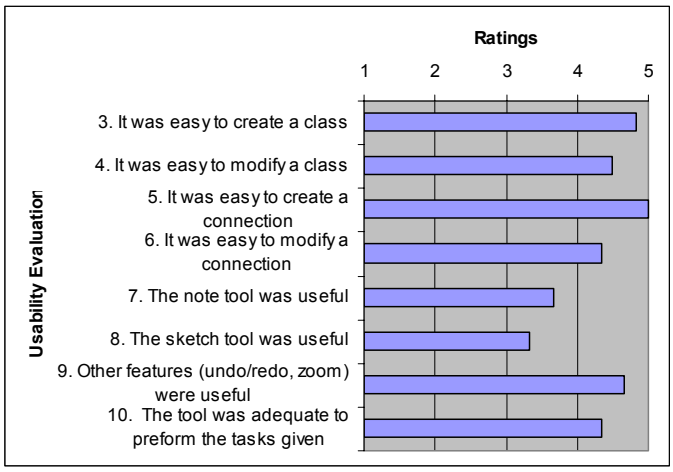

Fig.12. Average ratings for the usability questions.

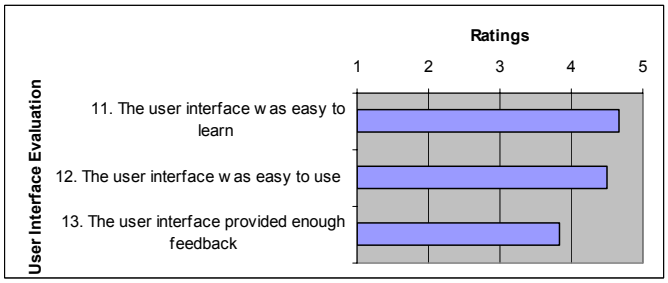

Fig. 13. Average ratings for the user interface questions.

less enthusiastic about the amount of feedback provided by the tool. From the other responses in the questionnaire, this may be attributed, at least in part, to the lack of tool tips for the buttons. Initially, several students seemed to be confused about the purpose of the buttons in the toolbar; tool tips may alleviate this problem, and have been added to the interface.

5.2.3 UML Support. The next four questions revolve around the amount of UML provided by the tool and its usefulness. (Note that in the analysis the scale for question 15 was inverted to match the others.) The participants seemed to be satisfied with the amount of UML and the efficiency of the tool, but they were neutral as to whether more UML functionality was needed. They also agreed only moderately that the tool was useful to them. In part, this data was skewed by the inclusion of a student who appears to have significantly more experience with UML than the others. Since this tool is intended for use in introductory courses, and was designed as such, it is no surprise that people with more UML knowledge do not find it useful. It is more important that the target audience, those with less background in UML, be satisfied with it. Since the task only required the amount of UML provided by the tool, it is not unexpected that the responses fall as they do. It would be more worrisome if the ratings were more negative, as that would indicate a serious flaw in the tool. Figure 14 shows the results for this section.

5.2.4 *Exploration Support.* The final questions, 18 to 23, focus on the exploration of the interface (see Figure 15). (Note that in the analysis the scales for questions 20, 21,

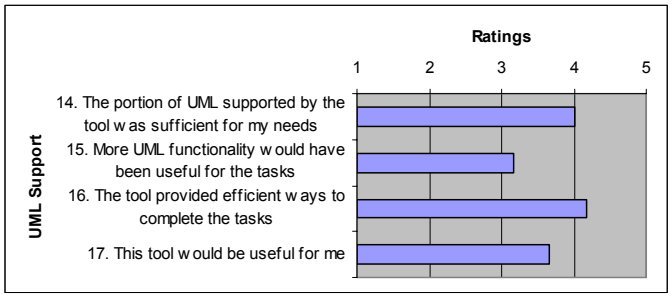

Fig. 14. Average ratings for the UML support questions.

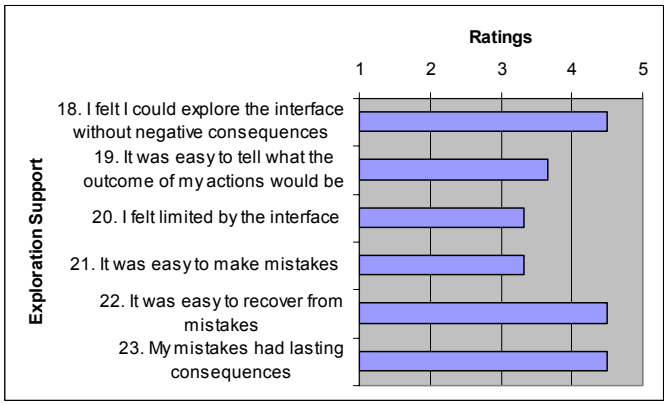

Fig. 15. Average ratings for the exploration support questions.

and 23 were inverted to match the others.) The students reported being comfortable exploring the interface without negative consequences; but were more uncertain about the outcomes of their actions. This may be related to the perceived lack of feedback from the tool. They were also only somewhat in agreement that the interface did not limit their actions. Again, this outcome is partially biased on the response of the one experienced participant who felt very limited by the software. The fact that the rest did not feel restricted indicates that this may not be much of a problem. What is of more concern is that students were close to neutral on how hard it was to make mistakes (to choose or be in the wrong mode, to change things they did not want changed, etc.). This concern is, in some respects, nullified by the positive responses to the ease of recovering from mistakes and to the perceived lack of lasting consequences for mistakes. Although there is not as much error avoidance as may be desired, the facilities for resolving missteps seem to be adequate. As a whole, the tool permits exploration fairly well.

5.2.5 *General Comments.* After the Likert scale questions, the participants were asked to describe what they liked and disliked about the classes, connections, and other features of the tool. They were also asked for their overall impressions and about features that were missing and should be added. Students were almost unanimous in approving the simplicity with which classes could be created and modified. One person liked the fact that the class name, variables, and methods were all immediately available to him. On the negative side, one user wanted to adjust the size of the box manually, and we observed several attempts to resize classes. We chose to use automatic sizing to ensure that each class displayed all its information and that the information was legible. With manual sizing, there is a problem where shrinking a class can obscure information or, if the font size decreases with the class size, render the text unreadable. Since there does not seem to be much advantage in increasing the size of a class, we disallowed it completely.

Another student wanted to specify the type of class as abstract or interface. In response to this comment and to the perceived need for interfaces for certain kinds of designs, support for interfaces was added. When editing a class, the student is presented with the option to select the object is an interface or a class. While there is no explicit support for other qualifiers, such as virtual, static, or abstract, a simple solution is to add any such information to the class name.

As they did for classes, students said that connection objects were very easy to create and use. One person did not like that connections were created and then given a type; he preferred separate tools for each type of connection.

We discussed our reasons for using our method earlier. It is interesting that the student wanting this distinction was the one with the most UML experience. This supports our claim that the more knowledgeable designers may think of the relationships between classes in concrete terms, while beginners may recognize a relationship but not necessarily know what kind. He also disliked our nonstandard method of labeling the arrows; we used text labels and not the standard diamonds, triangles, and so on. Since our tool is intended for novices, we used text labels to avoid any confusion over the meaning of symbols in the diagram. A better approach would have been to use a combination of symbols and text labels, which would remove the ambiguity of the symbols for novices while adhering to the standards (minimUML has since been updated to implement this).

We discovered a more serious problem with the connections, that is, that five out of the six students did not realize that they could be edited. Several students suggested that we add the ability to specify the type of relationship; they did not know that double-

clicking the connection would allow just that. The problem stems from having the connection default to a generic relationship where no text label is shown, so that most users see a simple line that does not have an associated, changeable value. To solve this, we decided to have the generic connection display a value when it is highlighted or selected. This change in state, as the line is manipulated, should provide enough information to indicate that it may be edited.

When asked about the features they liked, most participants replied that they liked the ease with which things were created or manipulated. One participant approved of the cut and paste abilities and the flexibility of the glyph tool. Several people mentioned that there was no way to add a type to the connections, a lack they did not like. Overall, participants said they found the tool useful and easy to use.

Finally, the participants suggested a number of ways to improve the tool: tool tips were an addition mentioned by two people; another person suggested a pop-up edit menu that allows the user to switch modes as an alternative to using the toolbar. Both are simple and effective ideas for improving the usability of the program. The creation of another view of the diagram as a list of classes was also suggested. While this would certainly be useful for larger projects, we are not sure how important it would be for the size of diagrams this tool is designed for. Its value may not justify the screen space it takes up. One solution would be to allow the user to show or hide the list at will or to place it in a separate window; however, this solution has not yet been implemented.

5.3. Classroom Evaluation

For the first time in the spring of 2005, in addition to our usability study, we used minimUML in the lab sections of a first-year object-oriented course; we report on our experiences in this section.

Students were asked to use the tool in a two-hour lab session. They were given a brief introduction to the tool and asked to prepare a small design. As this assignment was part of a lab session, the exact instructions varied with the TA running the section. As the end of the lab, we asked the students to answer a questionnaire to survey the ways they created their diagrams and the amount of time they spent learning to use the tool; we received 34 responses.

Of the 34 students that completed the survey, 29 were freshmen, 4 were sophomores, and 1 was a graduate student. All but one were 18 or 19 years old; the other participant was 23 years old. Their majors were also very homogeneous; there was a single math major among the computer science students; and only one female responded.

Only a very few students reported that they had much experience with UML. Half said that they had no experience with UML at all. Another 41% classified themselves as beginners who had used UML in some small class projects. Of the remaining three students, two were at an intermediate level and one was an expert.

First, the students were asked about what approach they used while designing classes. A majority (71%) said that they frequently or constantly modified their classes as the design progressed (see Table II for details). Some of the modifications were due to the iterative approach taken by some students when adding methods to their classes. While almost half of the students (47%) stated that they completely defined their methods from the start, 38% entered only some of the information for each method, such as a return type and/or some of the parameter information. One person reported initially giving methods a name only; three others used different strategies, depending on the situation. These responses highlight the need for allowing students flexibility in design. Enforcing

Table II. Class Design Strategies

When designing classes, I	
Constantly modify them (They look nothing like they were when they I started.)	21%
Frequently modify them (Most methods or variables change over time.)	50%
Seldom modify them (A few methods or variables change.)	21%
Almost never modify them (Methods and variables look almost the way they were after my initial design.)	6%
no answer	3%
When initially adding methods to a class, I	
Define the method completely including return type and parameter names and types	47%
Define the method partially, by giving it a name and a return type and/or some of the information about the parameters (name or type)	38%
Define the method partially by only giving it a name	3%
Do two or three of the above depending on the situation	9%
no answer	3%
When initially adding variables to a class, I	
Always give them a name and a type	79%
Give them just a name	9%
Give them just a type	3%
Do two or three of the above depending on the situation	6%
no answer	3%

syntactically correct methods at this stage would have negatively influenced half of the class.

The students dealt with the variables in a more uniform manner. Most (79%) always supplied a name and a type for each variable. Three students mentioned that they just put in a name, while one person gave only the type. Two people said that their approach depended on the situation. While there is not as much variance in dealing with variables, there is still enough to argue for a few restrictions on how values are entered.

Next, the students were asked about how they made connections between classes (see Table III). Over half (59%) reported that they made connections between classes and selected a type later. In addition, 35% said that they changed the type of the connection during the design process, sometimes frequently. This behavior supports our rationale for

Table III. Connection Strategies

Typically, the approach I follow when making connections between classes is (Select all that apply)	
Make connections between classes and immediately select a type (Aggregation, association, etc.)	15%
Make connections between classes and select a type later	59%
Change connection types as my design progresses	35%
Frequently change connection types as my design progresses	21%
Frequently add and remove connections	21%

Table IV. Instruction and Usage Times

The approximate amount of time the Instructor or TA spent on explaining the use of the minimUML tool was		**The approximate amount of time I spent on learning how to use the minimUML tool before using it as part of an assignment was**	
More than 10 minutes	6%	More than 10 minutes	3%
5 to 10 minutes	21%	5 to 10 minutes	21%
1 to 5 minutes	24%	1 to 5 minutes	32%
Less than 1 minute	15%	Less than 1 minute	15%
There was no explanation of the tool at all	32%	I immediately used it for the assignment	24%
no answer	3%	*no answer*	6%
I thought that this was		**I thought that this was**	
Too much time spent	29%	Too much time spent exploring	35%
Sufficient time spent	56%	Sufficient time spent exploring	47%
Too little time spent	12%	Too little time spent exploring	15%
no answer	3%	*no answer*	3%

providing a connection object that can switch between types instead of providing separate connection objects. Our method provides for this action directly, while the other method requires connections to be deleted and then recreated.

Finally, we asked about how much time the instructor spent explaining how to use minimUML and how much time the students spent exploring it (see Table IV). Some students, 32%, responded that they received no instruction on how to use the program; an additional 15% said that they were given less than a minute of introduction to the tool; 24% said 1 to 5 minutes; and 21% said 5 to 10 minutes. When asked if that was enough time, 56% said that it was, and, somewhat surprisingly, 29% said that too much time was spent. A third (33%) of the class remarked that they were comfortable with the time spent on instruction, even when they had some problems with the program. An additional third (38%) reported that they were neither uncomfortable nor comfortable with the tool. Of the remaining students who did not feel comfortable, only one said that he had many problems using it. It would be reasonable for an instructor to spend between 5 to 10 minutes demonstrating a new tool to a class at the beginning of a lab session or a lecture, and it seems that minimUML is simple enough for this to be an adequate amount of time for the majority of students.

The time the students spent exploring the program on their own followed a similar pattern. A quarter (24%) of the students immediately started using it for their lab assignment; a number of students,15%, played with the program for less than a minute; 32% used it for 1 to 5 minutes, and 21% spent 5 to 10 minutes. Only one student reported spending more than 10 minutes learning how to use the program. Again, most students thought this was sufficient (47%). or too much time (35%).

Our survey of students using minimUML as part of a lab shows us that it supports some of the basic behaviors that students engage in; minimUML allows for the iterative and sometimes ill-defined nature of student designs, and it allows students to easily manipulate the connections as their design concepts change. In addition, minimUML can be learned and used effectively in a lab session in a fairly small amount of time.

6. FUTURE WORK

While minimUML supports several methods for exporting designs, either to paper or to an electronic format, it does not currently allow for electronic submission to other tools, such a course management system. Ideally, there should be a way, similar to the method used by Allowatt and Edwards [2005] for their Eclipse plug-in, for students to submit and retrieve files directly from the tool. Once submitted, these diagrams could be sent to teaching assistants who would assess them and upload graded versions. Alternatively, the diagrams could be redistributed to the class for a peer review exercise. In either case, the ability to upload and download diagrams directly through the program would streamline the process and allow students to focus on their designs rather than on file management. We are currently exploring how to implement this feature.

We are also looking into a simple and generic method to download diagrams into minimUML. One possibility is to allow the user to drop URLs into the work area to open the files. While this would not allow students to upload files, it does not require server support to handle downloading files.

This program would also benefit from the ability to group and ungroup objects. Some objects, notably a group of glyphs used to write out a sentence, are logically a group. So it would be convenient to act upon them as if they were a single object, to allow for the creation of locational relationships between objects and, in some cases, make the diagram easier to manipulate.

Being able to import code as a diagram would be another useful feature for minimUML. Students would be able to compare their original designs to what is actually coded in their programs, and with a little introspection, they could learn a great deal from the difference. For instructors it could provide a way to gauge how well or badly the students were learning to design, but either way it would supply very useful information. Providing some way to compare diagrams automatically would be a possibility as well.

If students were able to import their code as a diagram, some form of round-trip editing of their code would be an interesting idea to pursue. Alphonce and Martin [2005] have implemented this kind of functionality in their Eclipse plug-in, Green. Ideally, a student would make a design and export it to code; the diagram would then reflect the changes in the code and vice versa. This would give the student two views of his work with minimal cost in moving between them. But there are difficulties to overcome to make this idea work. Synchronizing data between views so that work is not lost is a considerable problem, especially when the student wants to work with both views at once. Handling some of the intricacies of the code with the limited amount of UML provided by the tool is another concern that needs to be explored.

Adding some "style commenting" features may be useful. At Virginia Tech, researchers have been successfully using an automated grader called Web-Cat [Edwards 2003], which among many other tools uses an open-source style-checker that analyzes naming conventions and performs other static checks on code. By integrating a tool like this into minimUML, we could comment on the students' style before they start writing code. For example, we could check variable and method names to ensure that they are not too short or we could check the connections between classes to ensure that situations like circular inheritance do not appear. This might reinforce good programming style, even when working at an abstract level. We are currently exploring how to incorporate such tools into minimUML and whether their potential benefits outweigh the effort to integrate them or the added complexity in the interface.

We are currently planning to implement a preference that would add a default type for any missing variable/parameter types or return values. That is, if the student does not

provide a type for a variable, a default "Object" would be added. Similarly, "void" would be placed in front of a method without a return type, thus providing students with more structure without forcing them into early decisions.

Finally, a help menu would also be a good addition. While our goal is to make the basic usage self-evident, some documentation would be useful to explain the subtler features, such as being able to export parts of a diagram as an image by dropping it into an image editor. As new features, such as the electronic upload and download, are added, explanations may be needed to help both students and instructors.

While for the most part minimUML works well, it does have some flaws. One problem arises when there are a large number of objects in the diagram. Our original design did not include glyphs, so we assumed that if the program was used as intended, diagrams would perhaps have a couple dozen objects and very large designs could have 50 or 60. Thus far this assumption has held true in practice. However, the addition of glyphs changes this assumption radically. As we discovered later, a grader using glyphs for annotations created around 250 glyphs for a diagram originally consisting of 11 objects. In such a case the algorithms currently used to detect focus and selection run rather slowly; they should be replaced with more appropriate algorithms and data structures to take the increased number of objects into account.

7. CONCLUSIONS

In order to support object-oriented design early in the curriculum, we have identified a number of requirements that a good tool should support: it must be easy to learn and use; have features such as undo/redo that support exploratory learning and error avoidance; provide flexible printing options, support some code generation, allow for annotations, and focus on abstract design rather than coding. We have surveyed several available UML diagramming tools and found that they did not meet these requirements to our satisfaction. In response, we created minimUML that tries to meet these requirements. The results of our evaluation of minimUML were positive, and support our claim that the tool is appropriate for early CS education.

For minimUML, the goal is to provide just enough UML to support learning in early computer science education. The subset is the same as the one used in many introductory CS courses. With class and connection objects only, it provides enough power to create reasonably complex designs while not cluttering the interface or burdening users with extraneous features, making the tool easy to learn and use. Since there are very few restrictions on the way in which the classes are designed, it allows users to design at an abstract level. There is nothing to prevent users from starting with just a few high-level details which they can expand upon later. Furthermore, since it is so uncomplicated, it is hard for users to make serious errors. Common and expected features like multiple selection and cut and paste enhance its usability, while the undo/redo allows for exploration. We have taken a minimalist approach to create a UML design tool to make a simple, but useful, tool.

ACKNOWLEDGMENTS

We are grateful to Jaime García-Ramírez who has taken up the job of updating the minimUML tool. We also thank the students in CS 1705 and CS 1706 at Virginia Tech for graciously letting us explore the use of new tools that will benefit future students. The last stages of this tool were developed with support from Microsoft Research University Relations under their Tablet PC and Computing Curriculum Program.

APPENDIX A

Movie Rental Database

MegaMovies Corp. Inc. Ltd. has hired you to create a new program for their movie rental database. After giving you the details of what they want for their software, they ask you to produce an initial design that will show them how the program will be built. (You've noticed that their concept of the database is missing a lot important points, but you'll bring that up when you show them your design. And then ask for more money.)

The database consists of two tables, Movies and Customers, and their associated indexes. For the Movies table, MegaMovies wants it searchable on the movie title, the unique MegaMovies' id number, and any of the actors/actresses that star in the movie. The Customers table should allow lookups on the customer's name and the customer's id number.

For each movie, there need to be some standard information, such as the release date and the producing studio, and it is also necessary to store information about all the store's copies of a particular title and all of the reviews about the film. Each movie copy must have a unique id, some information about its current status, and, if it is rented, who rented it.

For each customer, an address and an account balance are needed. In addition, it must be possible to determine which videos a customer currently has rented. Once the basic functionality has been implemented, MegaMovies wants to allow users to write reviews about the movies they've rented, so you need to design a way to connect the customers to the reviews they have written.

Each movie can be reviewed by a critic, by the studio, or by customers. Each review has a reviewer, a rating, and a text review. The studios also provide some text describing promotional events for the movie and that needs to be stored as well. Unfortunately, this information is transmitted in different ways. The critics' critique comes in the form of a small XML file for each movie. The studios mail a CD with all the data stored in a large table and the customers' views are inputted as plain text. Luckily, you already have a XMLFileClass and a StudioReviewTableRowClass that will help with some of the reviews, so you don't need to worry about implementing or designing them.

With all that in mind, set out and diagram an initial plan for the program in UML.

APPENDIX B

minimUML User Evaluation Questionnaire

Participant Number: _____

Demographics

Gender: Male Female

Age: _____

Class level: Fr Sph Jr Sr

Major: _____

UML Background

1. How would you rate your level of proficiency with UML?

None — Beginner (small class projects)

Intermediate (several large designs) — Expert (used UML extensively)

2. What other UML tools, if any, have you used?

Task	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
3. It was easy to create a class	1	2	3	4	5
4. It was easy to modify a class	1	2	3	4	5
5. It was easy to create a connection	1	2	3	4	5
6. It was easy to modify a connection	1	2	3	4	5
7. The note tool was useful	1	2	3	4	5
8. The sketch tool was useful	1	2	3	4	5
9. Other features (undo/redo, zoom) were useful	1	2	3	4	5
10. The tool was adequate to perform the tasks given	1	2	3	4	5

Interface	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
11. The user interface was easy to learn.	1	2	3	4	5
12. The user interface was easy to use	1	2	3	4	5
13. The user interface provided enough feedback	1	2	3	4	5

UML	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
14. The portion of UML supported by the tool was sufficient for my needs	1	2	3	4	5
15. More UML functionality would have been useful for the tasks	1	2	3	4	5
16. The tool provided efficient ways to complete the tasks	1	2	3	4	5
17. This tool would be useful for me	1	2	3	4	5

Exploration	Strongly Disagree	Disagree	Neutral	Agree	Strongly Agree
18. I felt I could explore the interface without negative consequences	1	2	3	4	5
19. It was easy to tell what the outcome of my actions would be	1	2	3	4	5
20. I felt limited by the interface	1	2	3	4	5
21. It was easy to make mistakes	1	2	3	4	5
22. It was easy to recover from mistakes	1	2	3	4	5
23. My mistakes had lasting consequences	1	2	3	4	5

Impressions

24. When working with a class, what did you like the most? The least?
25. When working with a connection, what did you like the most? The least?
26. What feature(s), if any, did you like the most? Why?
27. What feature(s), if any, did you like the least? Why?
28. Briefly describe your overall impressions of the tool.
29. What should be added to the tool to make it more useful?

APPENDIX C

Questions/Students	S1	S2	S3	S4	S5	S6	Range	Avg	Std Dev
UML Background	B	N	I	B	I	B			
3. It was easy to create a class	5	5	5	5	5	4	1	4.83	0.372678
4. It was easy to modify a class	5	3	5	5	4	5	2	4.50	0.763763
5. It was easy to create a connection	5	5	5	5	5	5	0	5.00	0
6. It was easy to modify a connection	4	4	5	5	5	3	2	4.33	0.745356
7. The note tool was useful	3	3	4	5	3	4	2	3.66	0.745356
8. The sketch tool was useful	3	3	5	3	3	3	2	3.33	0.745356
9. Other features (undo/redo, zoom) were useful	5	4	5	4	5	5	1	4.66	0.471405
10. The tool was adequate to perform the tasks given	5	4	3	5	5	4	2	4.33	0.745356
11. The user interface was easy to learn	5	4	4	5	5	5	1	4.66	0.471405
12. The user interface was easy to use	5	4	4	5	4	5	1	4.50	0.5
13. The user interface provided enough feedback	4	3	3	4	5	4	2	3.83	0.687184
14. The portion of UML supported by the tool was sufficient for my needs	5	4	2	4	5	4	3	4.00	1
15. More UML functionality would have been useful for the tasks (Scale Inverted)	2	3	4	3	2	3	2	3.16	0.687184
16. The tool provided efficient ways to complete the tasks	5	4	3	4	5	4	2	4.16	0.687184
17. This tool would be useful for me	4	3	1	4	5	5	4	3.66	1.374369
18. I felt I could explore the interface without negative consequences	5	4	5	4	4	5	1	4.50	0.5
19. It was easy to tell what the outcome of my actions would be	4	4	4	3	2	5	3	3.66	0.942809
20. I felt limited by the interface (Scale Inverted)	2	3	5	3	1	2	4	3.33	1.247219
21. It was easy to make mistakes (Scale Inverted)	2	4	2	2	3	3	2	3.33	0.745356
22. It was easy to recover from mistakes	5	4	5	4	5	4	1	4.50	0.5
23. My mistakes had lasting consequences (Scale Inverted)	2	2	1	2	1	1	1	4.50	0.5

For UML Background:

N = None
B = Beginner (small class projects)
I = Intermediate (several large designs)
E = Expert (used UML extensively)

APPENDIX D

Installation

Download minimUML.zip.

The unzipped file should create a directory UML with 3 files:

- minimUML.jar – main program file
- jdom.jar – library file
- umldraw.cfg – configuration file (will be created with defaults if deleted)

Both jar files must be in the same directory.

Execution

If Java is associated with jar files: Double click minimUML.jar.

From the command line: [Your Java Path]\java.exe –jar minimUML.jar.

REFERENCES

ALLOWATT, A. AND EDWARDS, S. H. 2005. IDE support for test-driven development and automated grading in both Java and C++. In *Proceedings of the Eclipse Technology Exchange (eTX) Workshop at OOPSLA* (to appear).

ALPHONCE, C. AND MARTIN, B. 2005. Green: A customizable UML class diagram plug-in for eclipse (Poster presentation). *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA). ACM, New York.

ALPHONCE, C. AND VENTURA, P. 2002. Object orientation in CS1-CS2 by design. In *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education* (Aarhus, Denmark). ACM, New York.

ALPHONCE, C. AND VENTURA, P. 2003. QuickUML: A tool to support iterative design and code development. *Companion to the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Anaheim, CA). ACM, New York.

ANDERSON, N. AND SHNEIDERMAN, B. 1977. Use of peer ratings in evaluating computer program quality. In *Proceedings of the 15th Annual SIGCPR Conference.* (Arlington, VA). ACM, New York.

ARGOUML.2003. http://argouml.tigris.org/.Version 0.14.

BARNES, D. J. AND KOLLING, M. 2003. *Objects First With Java: A Practical Introduction Using BlueJ*. Pearson Education.

BUCK, D. AND STUCKI, D. J. 2000. Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. In *Proceedings of the 31 SIGCSE Technical Symposium on Computer Science Education* (Austin, TX). ACM, New York.

CARROLL, J. M. 1997. Reconstructing minimalism.In *Proceedings of the 15th Annual International Conference on Computer Documentation* (Salt Lake City, UT). ACM, New York.

CRAHEN, E. AND ALPHONCE, C. ET AL. 2002. QuickUML: A beginner's UML tool. *Companion to the 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, WA). ACM, New York.

DIA. 2003. http://www.gnome.org/projects/dia/.Version 0.92.

EDWARDS, S. H. 2003. Improving student performance by evaluating how well students test their own programs. *J. Edu. Resources Comput. 3*, 3, 1-24.

FOWLER, M. AND SCOTT, K. 1997. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, MA.

GEHRINGER, E. F. 2001. Electronic peer review and peer grading in computer-science courses. In *Proceedings of the 32 SIGCSE Technical Symposium on Computer Science Education* (Charlotte, NC). ACM, New York.

GREEN. 2005. http://green.sourceforge.net/index.php. Version 2.4.0.

Horstmann, C. 2004. *Object Oriented Design & Patterns*. Wiley, New York.

LEE, R. C. AND TEPFENHART, W. M. 2001. *UML and C++: A Practical Guide to Object-Oriented Development*. Prentice Hall, Englewood Cliffs, NJ.

LEE, R. C. AND. TEPFENHART, W. M. 2002. *Practical Object-Oriented Development with UML and Java.* Prentice Hall, Englewood Cliffs, NJ.

LEWIS, J. AND CHASE, J. 2004. *Java Software Structures: Designing and Using Data Structures.* Pearson Education.

LEWIS, J. AND CHASE, J. 2005. *Java Software Structures: Designing & Using Data Structures,* 2nd ed., Pearson Education.

NIELSEN, J. AND MOLICH, R. 1990. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Empowering People*. (Seattle, WA). ACM, New York.

QUICKUML.2003. http://www.cse.buffalo.edu/~alphonce/QuickUML/. Version 1.3.6.

RILEY, D. D. 2003. *The Object of Data Abstraction and Structures using Java*, Addison-Wesley, Reading, MA.

SULLIVAN, S. L. 1994. Reciprocal peer reviews. In *Proceedings of the 25th SIGCSE Symposium on Computer Science Education* (Phoenix, AZ). ACM, New York.

UMLET. 2004. http://qse.ifs.tuwien.ac.at/~auer/umlet/index.html. Version 1.8.

VIOLET. 2003. http://horstmann.com/violet/. Version 0.14.

ZELLER, A. 2000. Making students read and review code. In *Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE Conference on Innovation and Technology in Computer Science Education* (Helsinki). ACM, New York.

Received February 2006; revised June 2006; accepted August 2006