Integrated Test Cases in Educational Fusion

by

Wesley S. Chao

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 23, 2001

Author [illegible]
Department of Electrical Engineering and Computer Science
May 23, 2001

Certified by [illegible]
Seth Teller
Thesis Supervisor

Accepted by
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Integrated Test Cases in Educational Fusion

by
Wesley S. Chao

Submitted to the
Department of Electrical Engineering and Computer Science

May 23, 2001

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Educational Fusion is a collaborative, Web-based software platform designed to enhance the pedagogical process. It provides a visualization of the presented material, which may be algorithms, simulations, or virtual labs, and an underlying reference implementation. Students are then challenged to explore the reference implementation, recreate the steps manually, and then write and submit code that implements the algorithm.

This thesis details the design and implementation of integrated test suites, which allow students and teachers to run prefabricated and user-devised tests on the students' implementations of code. The system is tightly integrated with Educational Fusion's infrastructure, including the witness detection, collaboration, and visualization components. It also provides test case management, for personalized test suites. Finally, it has a robust reporting mechanism, for quick and easy viewing of test results and group statistics.

Thesis Supervisor: Seth Teller
Title: Associate Professor of Computer Science and Engineering

Acknowledgements

No section of acknowledgements could be complete, or for that matter proper, without recognition of my advisor, Professor Seth Teller. His original vision for the purpose and ultimate goals of Educational Fusion inspired a long line of excellent student Fusioneers, whose work in turn excited and motivated me. He has provided guidance and kept me in sight of the important aspects of Fusion all along the way, while placing his trust in me by giving me a greater degree of freedom and creativity than I could have imagined. While Fusion is not yet ready for widespread use, I have no doubt that Prof. Teller's care will make it so, and soon.

In the same vein, I thank Josh Glazer. His work on visualization and witness detection has been crucial to my own work; without the solid foundation that he left, I can only imagine how much more difficult my task would have been. Even in his rush to graduate, he found time to sit with me and make sure I understood all the little quirks and nuances of Educational Fusion.

Special thanks must also go out to my parents, who have always believed in me. While their trust in my ultimate success in any of my endeavors has occasionally given me pressure to live up to their expectations, it has also provided unwavering and unconditional support for me. Without them, I could never have come to MIT, nor, for that matter, done any of the other things that have defined and shaped my life. Also, I must thank my sister; even though we never fight, I know that if we did, no fight could ever break the bond that we share. I wish her good luck as she begins her own journey through higher education.

In the years that I have spent at the Institute, no group of people has been as supportive of me and as fun to be with as my friends. It is with great pleasure and a wisp of nostalgic sadness that I say goodbye to those people, from Baker, the MIT varsity water polo team, the MIT women's club water polo and men's club ultimate teams, and various other sources. Thank you, to those people, for making my time here as enjoyable as it was. Of my friends, a special place is reserved in my heart for Andrew, Eli, and Elsie, who have always been there for me, and have put up with all my quirks and insanities. How they managed to share a living space with me for a year, to say nothing of four years, is quite beyond my imagination. Also, a separate mention is warranted for Ed and John, to whom I could always turn when the vagaries of Java escaped me.

Lastly, I must thank one very special someone, who has been by my side the entire time as I made my way through this final year. I am convinced that without her emotional support and her constant prodding to work on my thesis, this project would never have become a reality. Jennie, you are my muse.

Contents

Figures and Tables

1 Introduction to Educational Fusion

Since the development of the World Wide Web in the early 1990's and the recent popularization of the Internet, it seems that most of humankind's activities are becoming enhanced by the online experience. Life, one might say, is becoming digitized. But in the rush to create the next e-commerce website or the smallest fully functional handheld organizer, one major aspect of life has been forgotten: education.

One might argue that education is slow to become wired because education is one of those tasks that derives so much from human interaction, that it would be impossible, for instance, to replace a live lecturer with canned video clips. But that argument is missing the point. Technology should not be used to supplant live education, but to supplement it.

In fact, in Professor Seth Teller's NSF Career Development Proposal, submitted while at MIT [Tel94], he proposes "an integrated collection of research and educational activities that will increase the efficacy and use of collaborative, interactive techniques for design & verification, experimentation, simulation, visualization, and pedagogy". It is from that document that the work of Educational Fusion, or simply Fusion, has sprung, and it is that document which continues to be the guiding force in the motivation and development of Fusion.

Up to this point, there have been many contributors to Fusion. Some of the major pieces of Fusion include Nathan Boyd's work on the underlying infrastructure [Boyd97], Brad Porter's work testing the first version of Fusion in a classroom [Por98], Nick

Tornow's work on collaboration [Tor98], Aaron Boyd's work on content development [Boyd99], and Josh Glazer's work on witness detection and algorithm visualization [Gla00].

1.1 What is Educational Fusion?

Originally, Fusion was conceived as a way to enhance the teaching process for computer algorithms through use of Java and the Web. Today, Fusion is a more robust, flexible system that is adaptable to many different kinds of content. Although it is now a much more efficient handler of non-algorithm material, including the hosting of previously computerized simulations that may have been written for a particular platform or lacking collaborative features [Tel+98], it is instructive to examine the underpinnings of the Fusion system and its beginnings as an algorithm teaching tool.

Traditional methodology for teaching algorithms involves lectures, readings, and problems that explore the behavior of specific algorithms. Sometimes, a problem might be to implement an algorithm in code, which ensures that the student understands not only the algorithm, but also associated data structures and memory management. While these are certainly important aspects of learning an algorithm, too often students get bogged down in the details of an implementation. The traditional solution, to provide students with a framework, complete with "your code goes here" comments, can leave students feeling confused and lost as to where their code fits into the big picture.

Fusion's approach to teaching algorithms is multifold. First, the student is presented with a Concept Graph, which shows an orderly grouping of algorithmic *modules*. Modules have inputs and outputs, which may be connected using lines that represent data. In this way, the student gets an immediate view of how each algorithm fits into the big picture.

Once the student chooses which algorithm to work on, a *visualization panel*, or simply *panel*, opens, and he is presented with a visual representation of the algorithm. If he does not quite understand the algorithm, he might choose to give the panel some input and run the algorithm in *reference mode*. This is a coded implementation of the algorithm he is studying. Given input and the command to begin, the panel will then display the workings of the algorithm, terminating with the output of the algorithm still visible.

To test his understanding of the algorithm, the student might then switch to *manual mode*, where he can again set the input, but instead of viewing an automatic running of the algorithm, he is challenged to reproduce the algorithm's actions manually. Taken together, reference and manual mode in a visual setting are powerful tools to aid the student in understanding the underlying algorithm.

The student can then enter *user mode*. In user mode, the panel runs code that the student must write himself. The student can edit this code using a text editor with many of the features of a professional development environment, and he can then compile the code, all within the Fusion environment. Fusion will also track different versions of code, so that the student can always go back to an idea that had promise and see if it leads to a correct implementation.

It is particularly important to note that the code editor automatically loads a *code template*, which can explain to the student what methods and data structures to use. This provides a powerful and flexible level of abstraction, where the student can either be relieved of all the details of the implementation, exposed to them, or any level between those extremes.

While interacting with the algorithm via the visualization panel, the student may choose to enable *witness detection*. Witness detection is a separate piece of code that, given the input to and output from an algorithm, checks that the output has no errors. Note that it cannot check if the code actually is correct, but by running a sufficiently representative cross-section of tests, it can determine if a code is incorrect, or offer a high probability that it is correct.

A further benefit of teaching and learning algorithms in the Fusion environment are the collaboration features. Students who are working in Fusion have access to many of the features found in professional messaging systems, including instant messages, offline messages, group discussions, announcements, persistent archives, and lists of online users. Additionally, students may make use of the ***help queue***, which sends questions to online staff. These requests are then processed in FIFO order.

The last benefit of Fusion is its strong infrastructure. It is Java-based, running in any Web browser with a Java VM, thereby enabling distance learning and platform independence. There is a simple checklist for creating new modules (see Appendix A), which allows any educator with a knowledge of Java to create robust, visually involving algorithm modules within the Fusion framework. And finally, Fusion's robust back end enables the possibility of future extensions. Most recently, the interface was extended to allow students to run interactive simulations and conduct experiments with real physical equipment remotely [Gla00].

1.2 Objectives and motivation

Even with the current feature set, which has been deployed in several classrooms, [Por98], Fusion is still not ready for widespread deployment. The modular approach and the witness detection systems are tailored for teaching material in which a student can demonstrate understanding by reproducing the reference implementations; it would be difficult to take advantage of all of Fusion's features for use in, say, a history class. This is a consequence of the infrastructure, and cannot change.

The ultimate goal of Fusion, of course, is to create a robust and powerful environment for teaching objective material. It would be extremely easy to use, thereby making it a compelling addition to any pedagogue's suite of teaching tools. This would lead to widespread use of Fusion, which would in turn lead to constant module development. With a wide variety of modules being in development by Fusion users everywhere, a teacher choosing curricula could pick from any of the modules which address his needs, or implement a new module.

Fusion's deployment readiness will not happen overnight. Several additions, changes, and tweaks can and should be made to the Fusion system before it should be used extensively [Chao00]. Each of these additions will come, as this one did, in line with several of Fusion's original objectives and goals.

1.2.1 Fusion objectives

Like any major software project, Fusion began with a unique vision. Provided by Professor Seth Teller in his NSF Career Development Plan, this vision is of a ground-breaking software platform that would allow for teachers to utilize the power of the Internet to a greater degree:

> **This proposal addresses the potential for collaborative interactive techniques to improve pedagogy at the undergraduate, graduate, and professional levels, and performance evaluation at the undergraduate level. As increasing amounts of technical data and simulations come 'on-line' and university courses begin to follow suit, we must fulfill the educator's roles of selecting, organizing, and presenting this material to the student. [Tel94]**

Armed with this specification, the original Fusion team then set out to create a set of objectives that the finished product would satisfy. As the system has evolved over the last half-decade, other objectives have been added. The most relevant ones are reviewed here.

1.2.1.1 Platform and location independence

Traditional computing environments have always been platform and/or location dependent, due to the resources needed to support them. As computing power becomes cheaper, it is more and more likely that students will wish to work from their own hardware in their own rooms rather than learning a new operating system or architecture, in some lab across campus. Therefore, Educational Fusion should operate in the same manner regardless of the underlying hardware, operating system, or location of the user.

1.2.1.2 Collaboration

In almost all learning environments, collaboration is at least encouraged, if not required. When the student-to-faculty ratio increases, students are more likely to learn the material if they have easy access to their classmates and to the teaching staff. Also, the Internet has strong foundations in transferring information between two or more parties. Therefore, Educational Fusion should support and encourage collaboration, both between students, and between students and faculty.

1.2.1.3 Visualization

When presented with a difficult or foreign concept, students often find it easier if they can refer to a picture of the problem. This is known as visualization. When working with algorithms, which can be particularly obtuse without visualization, research suggests that students learn particularly well under several conditions, including being allowed to view the visualization while solving problems [KST99] and when they can specify the inputs to the algorithm [LBS94]. Therefore, to maximize student learning, Fusion should provide students with a complete visual display of any material. It should also allow students to refer to this display and work at the same time, and be flexible enough to allow students to specify the data on which their algorithms run.

1.2.1.4 Usability

No matter how many features it has, no tool is useful if its intended audience cannot use it. This is an especially pronounced problem with software; as feature sets have become bigger, users are having increasing difficulty finding the ones that they need. Educational Fusion should have an intuitive interface that is easily understandable by anyone who has a passing familiarity with computers. Additionally, because the faculty and students may not be outstanding programmers, Fusion should provide ways to simplify any programming tasks, so that users can concentrate on the algorithms and their visualizations instead of worrying about the nuances of the programming language.

1.2.1.5 Transparency

Compiling, executing, validating, and submitting code are often causes of headaches for students working on implementing algorithms. The overhead of setting up a development environment and ensuring that it is consistent for all users can be a significant burden for both course administrators and the students trying to focus on learning the presented material. Educational Fusion should have a simple way to access all of these functions.

1.2.1.6 Bounded Creativity

Students learn best when they are genuinely interested in the material. One way to create interest is to allow students to have a high degree of interactivity with the material. To ensure that students are learning the curricula that teachers set for them, this interactivity must have strict boundaries. On the other hand, students should be allowed to exercise creativity and independent thought within these boundaries, else they will quickly exhaust the interactive options. Educational Fusion must strive to provide these boundaries without overly or arbitrarily limiting the student's input.

1.2.1.7 Accessible student work

Teachers cannot evaluate students' progress without being able to see their work. Members of the teaching staff should be able to access students' files at any time during their development process, so that staff can catch error patterns and misunderstandings as they arise. Educational Fusion should provide a simple way for teachers to view the students' work from their own browsers. If there is a large amount of data involved, Fusion should provide means to sort and organize this data, so that it is presented in a meaningful way.

1.2.1.8 Algorithm context

One of the traditional problems in assigning students to write code is ensuring that students focus on the task at hand without losing sight of the importance of that task. Often, algorithms call other algorithms as subroutines, or pass output to other algorithms. It is important for students to know what applications their implemented algorithms can have. Also, they should know what kinds of typical inputs their algorithms will receive. Educational Fusion should provide students with an overview into both these aspects of algorithm development.

1.2.1.9 Error detection

Teachers need to be able to effectively evaluate a student's understanding of material. They should not have to read code or run tests manually, nor should they have to determine by sight whether a student's implementation performs correctly. In order to ensure that students are submitting work that represents the extent of their knowledge, students should be encouraged to test their work before it is submitted to the teacher. Therefore, Educational Fusion should provide a simple way to run many tests, determine the success or failure of each individual test, and report the results.

1.2.1.10 Centralized Administration

When students are assigned to write code, they are often provided with a framework for doing so. Despite the teaching staff's best intentions, there may be bugs in this code. Changes to distributed code, or disseminating course information like announcements, assignments, readings, etc., must be released in a timely and efficient manner. Hence, Educational Fusion should provide ways for teachers to make changes and should automatically distribute those changes in a non-intrusive, non-destructive way to students.

1.2.2 How do test cases satisfy these objectives?

Several of the above objectives can only be satisfied by proper implementation of test cases. This mandates that proper thought be given to test case design and implementation, which is discussed in 3.2 and 3.3. However, the inclusion of test cases is important as a means for meeting some of the objectives.

1.2.2.1 Visualization

While Fusion currently provides a reference algorithm, with which students can interact, a student who is learning an algorithm for the first time may not know what inputs produce interesting output. He may spend a good deal of his time creating inputs and viewing the resulting animations of the algorithm without ever seeing some aspects of the algorithm because his inputs do not demonstrate them. With a properly specified test case suite, students can be given a set of inputs that teachers determine to cover all the interesting aspects of an algorithm.

1.2.2.2 Algorithm context

The Concept Graph is a powerful and informative tool for students wishing to understand how their algorithm receives its inputs. However, the Concept Graph for a particular course probably will provide only a small sampling of the possible input sources. Also, the input that an algorithm receives through the Concept Graph may be limited by the source; for example, a line drawing algorithm might receive input from a clipping algorithm, in which case the line drawing algorithm would never see lines with points outside the clipping area. Finally, the Concept Graph may not provide enough area for students to view all the important parameters of the input. A properly specified test suite, running in a visualization panel, addresses all these shortcomings. It can provide students with inputs that arise from sources not represented on the Concept Graph and that are not limited in any way. It can also allow students to view the parameters in the visualization panel, which is tailored to the algorithm and provides more viewing area than the Concept Graph.

1.2.2.3 Error detection

Perhaps the most obvious objective satisfied by test cases is that of proper error detection. While Fusion currently provides a robust and flexible way to detect errors in a single instance of input [Gla00], it does not detect whether the implementation actually is correct. Nor does it help the student determine what test cases to run or provide a way to store test cases once the user has found them.

The problem is exacerbated on the teaching side. Traditionally, teachers must evaluate a student implementation by reading through the submitted code, or by running test cases manually, once for each student. Not only is this a waste of time, it is prone to error. Also, teachers who know what test cases they wish to run have no place to store them within Fusion.

A testing framework and a properly specified test suite will ameliorate all of these nuisances. For students, they will provide a set of test cases that will help students determine if their implementation is likely to be correct, and they will encourage students to devise and run their own tests. For teachers, they will provide a quick and easy way to run multiple tests on multiple students, and will allow teachers to interpret the results. This will allow teachers to tailor their instructions to the problems with which most students are having trouble. Lastly, a testing framework will provide an integrated place to store test cases for future reference.

1.3 Background

As Fusion is nearly half a decade old, much of the infrastructure required for robust test case implementation is already in place. Indeed, a significant aspect of the challenge involved was discerning the specific functionality of literally hundreds of pieces of interlocking code, then puzzling out where to make changes in order to implement the desired functionality, without introducing new bugs into the system.

There are several independent pieces of functionality required as a basis for test cases, including file handling, client/server interaction, visualization, inter-client messaging, and witness detection. As these features were already present in a form not specific to creating, managing, viewing, running, and reporting tests, the work done involved making changes to them so that they could support test cases, then creating a test case layer to interact with them. This thesis will focus on the specific details of these modifications.

The next chapter will explore work that is currently being undertaken in both the Web-based teaching realm, and in the realm of testing Java code. This work was important in shaping the requirements of Fusion's objectives, both overall and in terms of user code testing. However, there are significant differences between these other works and Fusion, which will be explored as well.

Chapter 3 will go in-depth on the work involved in adding test cases to Fusion. A brief recap of the motivation behind test cases will be given. Then, this motivation will be translated into specific design criteria, which will in turn lead to the changes and additions made in order to effect the implementation of test cases. Finally, the impact that these changes have on designing and implementing new modules will be discussed.

Chapter 4 concludes this thesis by first answering the question of whether test cases make students' lives too easy. It then suggests a variety of other additions that could be implemented in Fusion, reminding the reader that Fusion is indeed a work in progress, but is with each new piece coming ever closer to its goals.

2 Related Work

As the Internet becomes ever more pervasive, educators are slowly developing teaching and learning materials to take advantage of the Web's communication and information distribution powers. At MIT, for example, e-mail is now commonly used to disseminate course announcements, instant messaging is used for students looking for help with problems, and the Web is used for posting readings, assignments, and handouts. This chapter reviews several interesting Web-based teaching and Java code testing software and contrasts them to Educational Fusion.

2.1 Web-based teaching

There have been several efforts at allowing students and teachers to use the power of the Internet to facilitate their interaction. Most of these are Web applications; like Fusion, they run in any Web browser that supports the Turing complete language Java. An overview of some of these applications is provided here in order to highlight some of the features and deficiencies of Educational Fusion.

2.1.1 WebCT

WebCT is a commercially available platform that is marketed to educational institutions. Its stated goal is to provide a "flexible, integrated environment where [one can] use the latest technology to foster inquiry, encourage discourse, and inspire collaboration…at many levels: the course, the department, the campus, and now the entire learning community" [Gol+01]. WebCT is written in Java, Javascript, and HTML, and runs in technologically current Web browsers (Netscape Navigator, versions 4.51 to 4.76 and Internet Explorer, versions 4.0 to 5.5) [Gol+01].

WebCT is built with ease of use in mind. Teachers using WebCT for the first time are given a Wizard-like interface to perform the basic steps required to set up their course, including setting the look and feel of the Web pages that define their course. WebCT also provides teachers with simple file and course management through the intuitive Web interface. Teachers can also draw on the files available in WebCT's communities, so that they need not reinvent the wheel.

Students and teachers using WebCT log in to a homepage with customizable links. These links may include collaboration tools, a calendar of important course dates, or course materials, like a syllabus or actual content. Content modules provide students with text, but allow for inclusion of links to external supplemental material as well as WebCT tools, like assignments and quizzes.

Finally, WebCT allows teachers to automate grading of student submissions. Students complete work that is easily checked, such as a series of multiple choice questions, and the system grades it automatically according to a set of answers that the teacher has provided. This ability is a microcosm of the inherent abilities of the WebCT implementation, in that WebCT provides a very powerful, suite of features that are meant to ease administration and facilitate distribution of *traditional* teaching materials: books, lectures, paper assignments, etc. However, this feature also gives insight into the limitations of WebCT. In order to create automated grading, teachers are limited to objective questions: multiple choice, true/false, limited short answer. Students do not have the freedom to express creativity or reasoning in their responses.

WebCT is an excellent tool for facilitating administration and simple assignments. But compared to Fusion, it does not have enough flexibility. A WebCT user is constrained by the limitations of the WebCT implementation; he would, for example, have difficulty adding an assignment that requires students to implement a sorting algorithm. Also, WebCT is a commercial product, making it difficult for small groups to take advantage of its features. However, WebCT currently enjoys a user base in the millions, including professors and students at several major universities [Gol+01], indicating that it certainly has earned a following among those looking for a Web-based tool to replace traditional administration methods.

2.1.2 CourseWork

Another Web-based teaching tool is the software platform CourseWork, which is very similar to WebCT, in that it provides all the same features, on the same Web/Java platform [Ker01]. However, where WebCT is a generalized tool, CourseWork was specifically developed by Stanford University, for Stanford University. Hence the system has a level of infrastructure that is not seen in WebCT. This translates into two different features.

First, CourseWork is fully integrated with the rest of Stanford's academic support systems. CourseWork uses Stanford's SUNeT authentication system, so that students can use the same account and password for CourseWork as they do for using the Stanford network. CourseWork also is linked to the Stanford Registrar, so that students are automatically given access to courses in which they are enrolled. Finally, CourseWork is integrated with the Stanford Library system, so that students may perform online searches of those materials. They may also take advantage of links to materials that may be stored online.

Second, CourseWork supports a college learning environment. That is, where WebCT is designed for any educational enterprise, CourseWork is specifically designed for teaching and learning in a university setting. This means that it is aware of traditional college teaching methods, including lectures, problem sets, seminars, labs, group projects, and discussions, which can be held online through the PanFora system [Sza+01]. It also means that while it only allows students to do certain things online, like take simple quizzes of the same type found in WebCT, it facilitates assigning and grading those methods. CourseWork also supports distribution of students by group, assignment, project, or section, and the distribution of differing levels of teaching staff (faculty and TA's) to those students.

CourseWork also builds upon some of the features found in WebCT. For instance, when taking a multiple-choice quiz, students are given space to justify their answers, a testing methodology often seen in traditional classrooms. CourseWork then allows staff to sort and view these responses. Also, CourseWork allows for audio submissions, so that it can be used for language classes.

However, insofar as CourseWork shares similarities with WebCT, it also shares the same inherent limitation: it is at its core a tool for "handling much of the time-consuming bookkeeping and communication tasks associated with courses" [Ker01]. As such, when compared to Fusion, the current infrastructure is able to support fewer types of wholly online work, like implementing algorithms in code or running simulations. Still, CourseWork is not fully implemented yet. The development team expects to have the core functionality running by Fall 2001, and they plan to work closely with teaching staff to determine what features should be created or modified to better support Stanford teaching. Hence it will be worthwhile to revisit CourseWork once it is fully realized.

2.1.3 WolfWare

WolfWare is a different type of Web-based teaching tool. Developed "as a joint effort between the College of Engineering and the College of Physical and Applied Mathematical Sciences [at North Carolina State University], the mission of WolfWare is to provide a cohesive Web-based course presentation package which operates within the NC State Realm computing environment" [Harr+99]. Originally conceived as a way to enforce a standard way of storing and managing course materials on an AFS file system, WolfWare now also provides some of the previously mentioned features.

WolfWare differs from the previous tools in that it is intended to make managing the files that represent content easier. Hence it lacks some of the amenities found in WebCT and CourseWork. Its feature set is primarily concerned with actions that are directly related to the storage and retrieval of content, including content archival, Web page templates, homework submission, enrollment management, and content scratch space. It also provides some additional course support features, like course calendar and syllabus creation, mailing lists, and a course message board. WolfWare also differs from the previous tools in that it is written entirely in Perl.

Due to its origins as a content storage standard, WolfWare has a more limited feature set than either WebCT or CourseWork. It maintains the same platform independence by using the Web interface, but it is written entirely in Perl instead of Java. When comparing WolfWare to Educational Fusion, it is again important to note that WolfWare is a tool for administrative tasks, not an interactive teaching platform. Of the Web-based teaching tools reviewed so far, all have fallen into the former category, while only Fusion can be placed in the latter.

2.1.4 Hamlet

The Hamlet Project, from the Computer Science department at the University of Utah, is intended to provide simple capabilities useful for creating simple HTML-based online tutorials [Zac+97]. It differs significantly from the other tools in several ways.

First, Hamlet is neither platform-independent nor wholly Web-based. It is a Unix-based X application which is spawned when students access the Web page that contains the tutorial. This implementation means that any files accessed or modified by Hamlet must be stored locally, on the client's machine. All of the other tools and Educational Fusion store their files on a centralized server.

Second, Hamlet can run local scripts, send commands to other currently running applications, and perform basic file management tasks. This makes Hamlet tutorials ideal vehicles for use of external programs, such as the widely used MATLAB or Maple. Indeed, most of the sample Hamlet tutorials [Zac+97] involve external programs.

Finally, Hamlet is not intended to augment traditional teaching methodology. While the other software packages have generally provided methods of migrating administrative and simple teaching tasks online, Hamlet provides a way to create wholly self-sufficient tutorials. Thus, Hamlet might be used for creating a document that teaches a specific aspect of a course, but not for testing a student's understanding of that aspect through assignments or quizzes.

There are several differences between Hamlet and Fusion. Hamlet is not Web-based or platform independent, which makes it inappropriate for distance learning. Additionally, it is possible that users will have trouble installing the Hamlet viewer. Hamlet's use of local file storage makes it unsuitable for tasks, like grading, whose results must not be accessible to students. Hamlet does not provide any methods for basic administration, like tracking students enrolled in a specific course. Due to this limitation, Hamlet is unable to provide any sort of collaboration features. However, Hamlet does have one very significant advantage over Fusion, which is that it can easily utilize external software programs. Due to Fusion's client/server architecture, any external programs must reside on the server and be called specifically by the client. Currently there is no framework in Fusion for doing so.

2.2 Java code testing

There have been people testing computer programs almost as long as there have been people writing computer programs. As it is an extremely difficult problem to ensure that a piece of code performs exactly as specified, testing code consists of running a sufficiently varied array of glass box (visible parameters) and black box (hidden parameters) tests each time the code is changed. If any of the tests fails, then the code is clearly incorrect. If all the tests succeed, then the probability that the code is correct is high.

Due to the high degree of repetition involved, there are many tools available to testers to help automate the testing process. However, due to the relative novelty of the Java programming language and the extreme demand for quality testing tools, most of these tools are only available commercially. Two software packages that are freely available for download are reviewed here to determine the required functionality of the Fusion testing environment.

2.2.1 JUnit

JUnit is a testing framework that "defines how to structure your test cases and provides the tools to run them" [Gam+01]. It is open source, freely downloadable, and now in version 3.6. JUnit is written for Java developers, and as such requires a fair knowledge of Java programming. It is written in Java and can be run under any operating system that support the Java programming language.

In JUnit, programmers create tests by creating a new class, which subclasses the JUnit package class TestCase. This class can then be outfitted with any number of tests, represented by methods. It can also implement methods setUp and tearDown, which create test fixtures, or objects that will be commonly used for every test. Finally, the class is outfitted with a method to return a suite of all the tests.

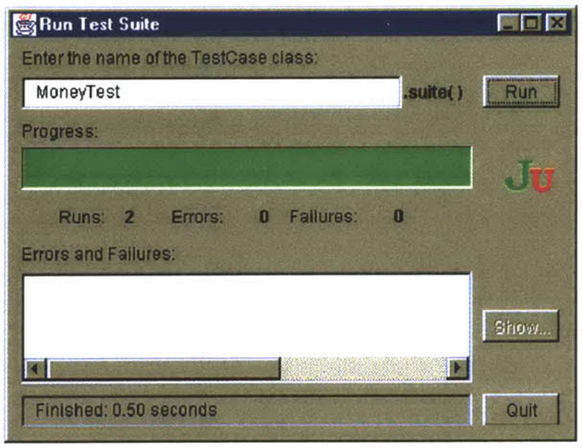


Figure 2-1: The JUnit GUI

JUnit also provides a GUI for running tests, as seen in Figure 2-1. The programmer enters the name of the newly created class of test cases, and clicks Run. JUnit then runs the returned test suite while maintaining a progress bar of the number of tests that have been run. This bar remains green until a test fails, at which point the bar turns red. JUnit keeps track of how many tests are in the suite, how many of those ran successfully and unsuccessfully, and how long the tests took to run.

Because JUnit allows the user to write any method in Java to execute tests, it is extremely flexible. However, this method of implementing tests means that any user creating tests must have a solid understanding of Java, especially if the tests are complicated. While this is not a problem for Java programmers, who are the intended audience of JUnit, the faculty using Fusion are neither required nor expected to have this understanding (see section 1.2.1.4).

2.2.2 Java Test Driver

In 1998, in order to fulfill the need of a client to test all aspects of a system written partly in Java, Altisimo Computing developed the Java Test Driver [Kas99]. Now in version 2.0, the Java Test Driver, hereafter referred to as JTD, and its source code have been made freely available. Like JUnit, JTD requires that the user write tests in Java.

The format for writing tests in JTD is very similar to that of JUnit. Instead of a class file that holds all the tests as methods and returns a suite of them, JTD uses a class file that holds the tests as methods, but is itself the test suite. Like JUnit, JTD allows users to specify methods for initialization and cleanup. Also, JTD and JUnit both have simple binary output. In JUnit, either all the tests succeeded or they didn't; in JTD, each test either succeeds or it didn't.

JTD differs from JUnit in a few significant ways. First, JTD has no GUI. Instead, users run their tests and view their results through a command-line interface. Second, JTD allows users to specify which test cases to run on the command line, whereas JUnit runs every test case every time unless the code is changed. JUnit then runs these tests in alphabetical order, regardless of the order in which they were specified; JTD runs them in that order.

Like JUnit, JTD expects that its users will have a working knowledge of the Java programming language, which may not be true for faculty Fusion users. And while JTD's ability to run a subset of the available test cases is a useful feature, JTD is similar to JUnit in that it is lacking in a rich results reporting mechanism, a shortcoming that is exacerbated by the lack of a graphical user interface.

3 Test Cases

Most programmers are familiar with the concept of the test case, which is simply an input to the tested code that verifies that some aspect of the code is working correctly. Test cases in Fusion serve the exact same purpose, within the context of an education tool: they help students examine their code and refine it until it is error-free and ready for submission, and they help teachers evaluate submitted code so that they can analyze student performance. This chapter revisits the motivation behind adding test cases to Fusion, then explains the rationale used in the decisions made in designing the test case framework. Next, an overview of the implementation is given. Finally, this chapter concludes by discussing the changes that need to be made in order for modules to support test cases.

3.1 Motivation

Educational Fusion was designed as a learning tool. One of the methods that Fusion uses to help students learn algorithms is to have them write their own implementations of those algorithms in Java. In order to increase understanding of algorithmic concepts and enhance the current implementation's satisfaction of several objectives, including algorithm visualization, algorithm context, and error detection, Educational Fusion should include a subsystem for creating, running, and reporting test cases.

3.2 Design criteria

As is true with the addition of any new features into an existing system, it is vitally important that the test case subsystem not interfere with the operation of current features. In fact, the test case subsystem should try wherever possible to use the methods and classes already implemented in Fusion. Further, the implementation of test cases must conform to all of Fusion's original objectives, including platform and location independence, collaboration, usability, transparency, bounded creativity, accessible student work, and centralized administration. For the implementation to meet these constraints, it is important that careful attention be paid to the design of the test case subsystem.

3.2.1 Intuitiveness

A suite of test cases is no good unless students use it. As such, the interface for using test cases must be clear. There should be a single, uniform entry point to test cases for every module, and the placement of this entry point should be consistent from module to module. It should also be intelligently placed, so that students can find it quickly.

Once the entry point is accessed, Fusion should present a simple interface for all test case functionality. The layout of the interface should be easy to understand and the purpose of each of the interface elements should be clear. When a Fusion user chooses to run some set of tests, the system should run those tests while informing the user of its progress, then give useful feedback on the result of those tests.

If the interface is properly constructed, then the Fusion objective of usability will be satisfied. Additionally, if the methods that run tests do so within the Fusion framework, then the objective of transparency will also be satisfied.

3.2.2 Reporting

The test suite should have several methods of reporting, depending on the type of user. Students should be given a general report, which tells them how many and which of the chosen test cases resulted in successful outputs. They should also have case-specific reports, explaining in each case whether the test was a success or failure, and if it was the latter, what went wrong, including such possibilities as incorrect output, exception thrown, or timeout (infinite loop).

Staff should have options for running tests on any arbitrary subset of students. Once a member of the teaching staff selects the test cases and students to run, the system should run the selected tests on the selected students. It should then report the percentage of correct cases for each student, as well as the percentage of students that got each case correct. Finally, the system should also allow staff to examine individual test outcomes.

In each case, the reporting mechanisms should provide the ability to sort the results by outcome and by student (for teachers). If the reporting mechanisms are implemented properly, the Fusion objective of usability will be met. Additionally, the ability to display student output in an organized, easily readable way meets the Fusion objective of accessible student work.

3.2.3 Authoring

Students should be given the ability to add test cases to their personal suite of test cases, with a separate personal suite of cases for each module. This will allow and encourage students to come up with their own inputs, to make their personal testing process more robust. Since the student is most likely to generate test case candidates while creating input on which to run the algorithm, the system should support an easy way to "snapshot" a set of inputs and add it to the test suite as a named test case.

In addition, students should be able to submit their test cases to the teaching staff. Staff should then have a simple way to add an interesting test case to the standard suite of test cases distributed to all students. This allows the staff to modify or add to the test cases that they feel are important for students to see.

Creating the authoring aspects of the test case subsystem as described will satisfy several Fusion objectives. First, allowing students to submit test cases to staff meets the collaboration objective. Second, the one-click "snapshot" of input satisfies the usability criterion. Third, allowing the student to create his own test cases by using only the interface elements provided in the module's visualization panel satisfies the objective of bounded creativity. Finally, if there is a standard suite of test cases, then the centralized administration requirement of Fusion will be fulfilled.

3.2.4 Opaqueness

Sometimes, students can benefit from seeing the parameters that make up a particular test case, so that they can determine exactly why their code is not working.

However, staff may wish to release a set of test cases that will be used for grading purposes. These cases' internal values should not be visible to the student, to prevent students from tailoring code to match the inputs.

Therefore, the system should provide an option of opaqueness. Either the test case should be transparent, so that the student can see both the name of the test case and the internal parameters, or the test case should be opaque, so that the student can only see the name. An opaque test case simply prevents students from accessing the internal parameters; students can still gain information from the name that staff gives to the test case and, if the test results in failure, the error that occurred.

A student's own addition to his personal test suite has no need to be kept opaque, and so will always be fully transparent. If the criterion of opaqueness is present in the implementation of the test case subsystem, students will be kept honest as they work. Also, if some key test cases are kept hidden, students may realize that the standard test suite might not necessarily encompass all the tests required to ensure correctness. They will then attempt to add tests to make the suite complete. Thus, opaqueness encourages students to exercise bounded creativity.

3.3 Implementation

There is one Fusion objective that the design criteria on their own do not address: platform and location independence. In order to maintain platform and location independence, the test case subsystem is entirely coded in Java. In recent versions of Java, Sun Microsystems has introduced a new package of classes for implementing graphical user interfaces, called Swing. The Swing package provides several features that would be extremely useful and timesaving in creating the user interface. However, there is no native support for Swing in major browsers. Using Swing would force Fusion users to download and install a Swing plug-in for their browser, which may be difficult.

3.3.1 Changes to Fusion infrastructure

In implementing the new test case subsystem, it was important to integrate the subsystem into the Fusion infrastructure. This meant taking advantage of the existing classes and methods as much as possible. In order to support the subsystem, however, a few changes were necessary. The dependencies between the new classes and the existing classes are shown in Figure 3-1.

3.3.1.1 EFApplet

Two simple changes were made to the existing EFApplet class. EFApplet is the main class of Fusion, which is loaded whenever a user logs into Fusion. As such, it houses as instance variables all of the major pieces of Fusion, including the controller for file versioning, called VersionController, and the collaboration subsystem, called CollaborationPanel.

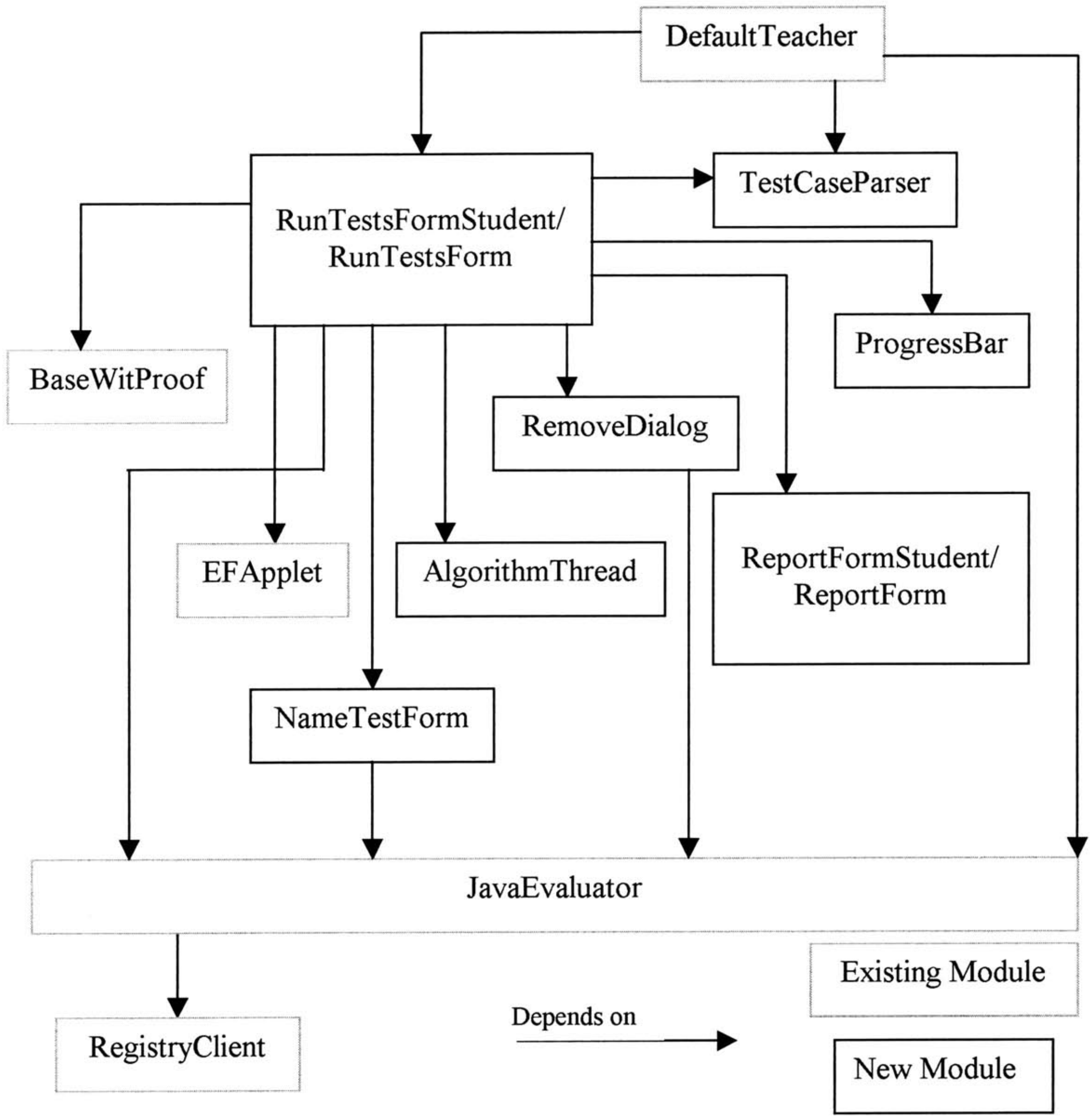


Figure 3-1: Module dependency diagram for test case subsystem

The first change was to allow EFApplet to return the instance variable representing the versioning controller, VersionController. VersionController stores an instance variable that represents the current module. As the visualization panel is divorced from the actual module to create interchangeable modules and panels, the visualization panel does not know what module it is displaying. Hence VersionController was changed to allow it to return the name of the current module. In this way, the panel is still separate from the module it represents, but it can ask VersionController for the module name, to be passed to the test case subsystem.

The second change was to allow EFApplet to return the collaboration subsystem, CollaborationPanel. CollaborationPanel houses the controller for Fusion's messaging systems, called DiscussionController, which it is already able to return. This class has methods for sending and receiving messages. Since the test case subsystem needs to be able to send a test case from a student to a teacher, it must have access to the user's DiscussionController.

3.3.1.2 JavaEvaluator and RegistryServerHandler

Fusion runs on a client/server architecture, between which there are Java streams carrying information [Boyd97]. The two classes that handle almost all of this traffic are RegistryClient and RegistryServer. However, for convenience, when a user logs on to Fusion, the RegistryClient is instantiated in another class, called JavaEvaluator. This class has many static methods that are used to make calls to the RegistryClient, avoiding the need for programmers to pass around the instance of RegistryClient. The

RegistryClient then passes these calls to RegistryServerHandler, which sits on top of the RegistryServer and, as its name implies, handles the incoming and outgoing streams.

Because Fusion runs in a Web browser in Java, it cannot access the file system of the user's machine. Therefore, Fusion's data is stored in files that reside on the server. Currently, JavaEvaluator and RegistryServerHandler have methods for file access, but they are tailored to specific files. Hence it was necessary to modify both these files to accommodate reading and writing test case files from the server. The implementation is generalized, so that future Fusion developers may use these methods to read and write any file on the server.

3.3.1.3 BaseWitProof

Recall that Fusion has a system for detecting errors in the output of algorithms, called witness detection. The BaseWitProof class is the superclass for all classes that implement methods for witness detection. It is required for module creation. BaseWitProof has two abstract methods that must be implemented by the module designer: **FindWitness**, which returns a String, and **FindWitnessList**, which returns a Vector. Both of these methods take as argument two Objects, an input and an output to an algorithm. In most cases, the String returned by FindWitness summarizes the error, so that it can be displayed on the visualization panel, and the Vector returned by FindWitnessList contains some kind of identification for the errors contained in the output, if any.

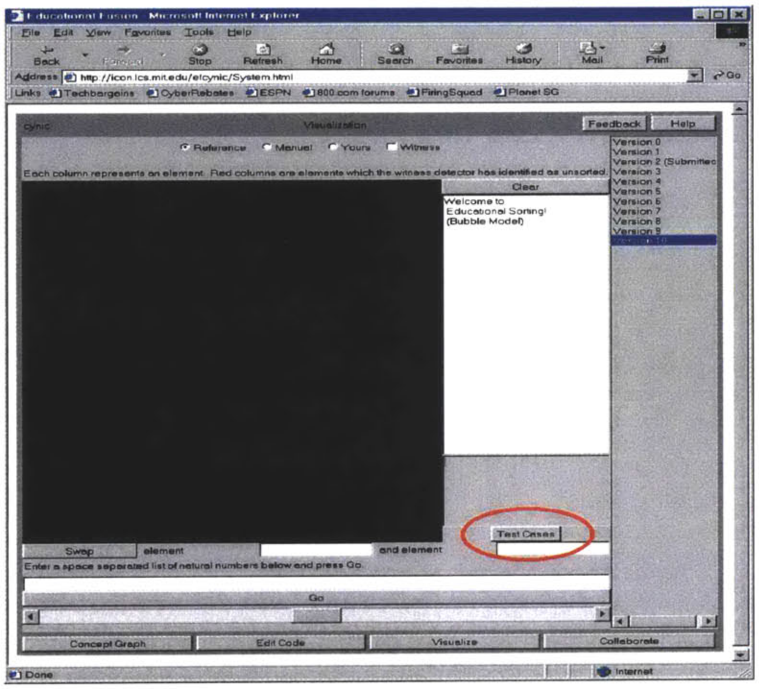
Because of the flexibility of module design, there may be some modules where it is difficult to say what is wrong without a visual display, or where it is obvious what the error is by simply looking at the display. In these modules, FindWitness may not be implemented at all. However, every module is expected to have some display. In order to assess the errors and display them accordingly, every module can be expected to implement FindWitnessList.

Therefore, when the test case subsystem runs a test and needs to know whether that test succeeded or failed, it is natural for it to call the BaseWitProof subclass for that module. BaseWitProof was modified so that it now tracks whether the last test that it was called to evaluate was successful or not, and can return this boolean value. Due to the increased likelihood of the existence of FindWitnessList, it is now also a requirement that FindWitnessList set the value to true or false somewhere within its implementation. See Appendix B, Modifying FindWitnessList, for an example of how simple this change can be to an existing FindWitnessList method.

3.3.2 Changes to DefaultTeacher

DefaultTeacher is the superclass for all algorithm visualizations. It performs many functions, such as creating the user interface, parsing input to that interface, calling the algorithm for the selected mode (reference, manual, user), calling the witness detector, and displaying the results of an algorithm being run. As such, it is necessary to make several changes to the DefaultTeacher class.

First, as the DefaultTeacher interface is included in the interface for every module, it is the logical place to put the entry point for test cases. On the right side of every algorithm visualization panel, there is an area for tracing the calls to the algorithm and an area for displaying the output from the witness detector. Students are used to looking to that side of the panel for the results of running algorithms on the current input. Therefore, it makes sense to put a button to that access the test case features, as shown in Figure 3-2.


Each column represents an element. Red columns are elements which the witness detector has identified as unsorted.

Welcome to Educational Sorting! (Bubble Model)

Enter a space separated list of natural numbers below and press Go.

Educational Fusion - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Address http://icon.lcs.mit.edu/efcynic/System.html
Go
Links Techbargains CyberRebates ESPN 800.com forums FiringSquad Planet SG
cynic
Visualization
Feedback
Help
Reference
Manual
Yours
Witness
Clear
Version 0
Version 1
Version 2 (Submitted
Version 3
Version 4
Version 5
Version 6
Version 7
Version 8
Version 9
Version 10
Test Cases
Swap
element
and element
Go
Concept Graph
Edit Code
Visualize
Collaborate
Done
Internet

Figure 3-2: Placement of entry point into test cases

Second, currently a panel has no way of passing information about its user interface. The layout and elements of the interface are usually either protected instance variables, or they are created when the panel is initialized and cannot be accessed at all. This makes it impossible for the test case subsystem to interact with the user interface, which it must do in order to perform any useful functions through the panel. Thus, DefaultTeacher requires that visualization panels that wish to support test cases properly implement three methods: **populatePanel, readPanel,** and **clearUI**. The first method,

populatePanel, takes a single argument, a Vector of Strings, and applies them to the user interface elements in such a way so that they comprise all the information needed to define input. The second method, **readPanel**, does the inverse: it polls all the necessary user interface elements and creates a Vector of Strings that defines the test data. Finally, **clearUI**, as its name suggests, restores the user interface to the initialization state. These methods are to be implemented by the module designer, who knows exactly what the user interface contains, and so will be able to do so easily.

Third, the flexibility of module authoring implies that not every module's algorithm will return the output all at once. Nor will every module store all the output at the same time. For instance, the Bresenham module, which displays as output a series of Pixels, has an algorithm that returns one Pixel at a time, and stores only the last Pixel as its output. Therefore, to be able to call FindWitnessList on an appropriate object, DefaultTeacher requires that its subclasses implement the method **runWitnessCode**, which makes the call to FindWitnessList on the right object. The four described methods, **runWitnessCode, populatePanel, readPanel,** and **clearUI**, are referred to as test case support methods.

Finally, there are already many modules written using DefaultTeacher. It would be unnecessarily burdensome to create the test case support functions for each of these modules, but wasteful to remove them from Fusion's module library. Therefore, test case support is optional, but is strongly encouraged for all future modules. To make this support optional, DefaultTeacher has a method, **supportTests**, which returns a boolean value that represents whether the module supports test cases or not. By default the method returns false, so that old modules do not need to implement the test case support functions.

3.3.3 Additions

In order to implement test cases and the associated user interface, it was necessary to make several additions to the Fusion codebase. First, several classes were created to house the user interface for the test case subsystem. These classes also contain methods for the subsystem's functionality. Originally, it was believed that only two classes would be needed, one for test case interaction and one for reporting the results of a run. However, it was soon discovered that several user interactions required additional input, which would look awkward and clutter up the main panel. Hence the number of viewing panels has been increased to four. Also, there are two helper functions. One allows tests to be run in a separate thread, while the other houses methods for parsing a properly formatted test case file. Finally, there is a simple class that is called to display errors to the user.

3.3.3.1 RunTestForm and RunTestFormStudent

The RunTestForm and RunTestFormStudent are two classes that represent the bulk of the test case subsystem's functionality. As their names suggest, one is used as an interface for teaching staff, while the other is used for individual students. Figures 3-3 and 3-4 show the windows that are created by these two classes when their respective users click on the "Test Cases" button in the algorithm visualization panel.

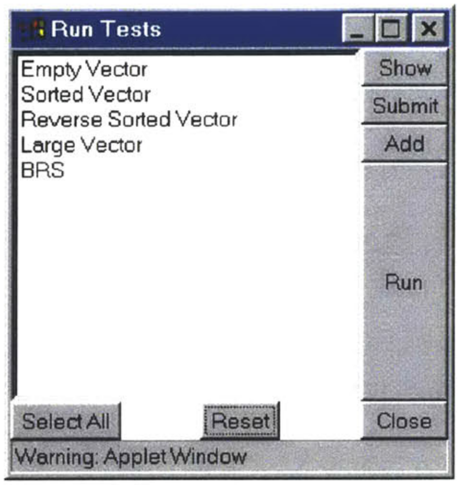


Figure 3-3: GUI created by RunTestFormStudent

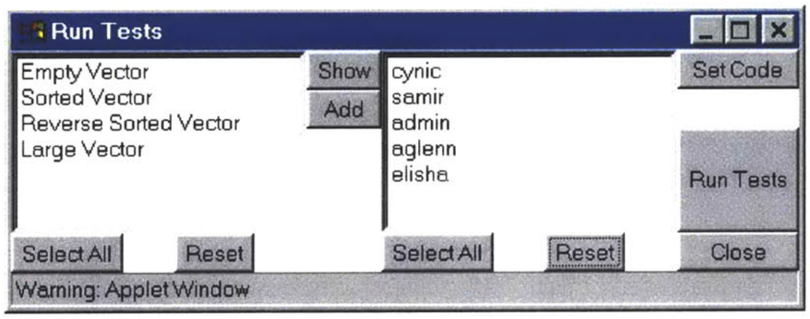


Figure 3-4: GUI created by RunTestForm

In each window, the user is able to see a list of the available test cases. In the student window, these test cases are contained in a personal test case file, stored in the user's personal module directory on the server. The teacher test cases are stored in a standard test case file, located in the directory that houses the other module files. Users can select all of available tests or deselect all of them using the Select All and Reset buttons. Additionally, in the staff window, the user can see a list of all the users who are currently registered, and can select or deselect them using the appropriate buttons. The Show button is used to display the parameters of the test case in the associated visualization panel. In keeping with the opaqueness design criterion, students may only inspect those tests that are not designated opaque, while teachers may inspect any tests.

The Add/Remove button allows users to perform basic test case management. When adding a test case, the system will create a new NameTestForm, which allows the user to name the test case and designate whether it is opaque. The removal procedure is slightly more complicated for students. Teachers can remove any test case that is in the standard suite, but students can only remove tests that they have added themselves. The system will check whether the user is able to remove the selected test cases, then open a RemoveDialog listing the subset of test cases that he is authorized to remove.

The Run Tests button on the teacher window takes the selected tests and runs the selected students' submitted user code on them. The method that is called when the button is pushed puts each test in a separate AlgorithmThread, so that the method will not be disabled by code that contains an infinite loop. It uses the witness detector of the module to detect whether the output is correct or not. As such, it is limited to detecting those errors that the witness detector can detect. Additionally, it can detect whether the submitted code threw a Java exception, if it took too long to return, or if there was no code submitted at all. The user is queried for a timeout period; if no period is given, the default timeout is ten seconds.

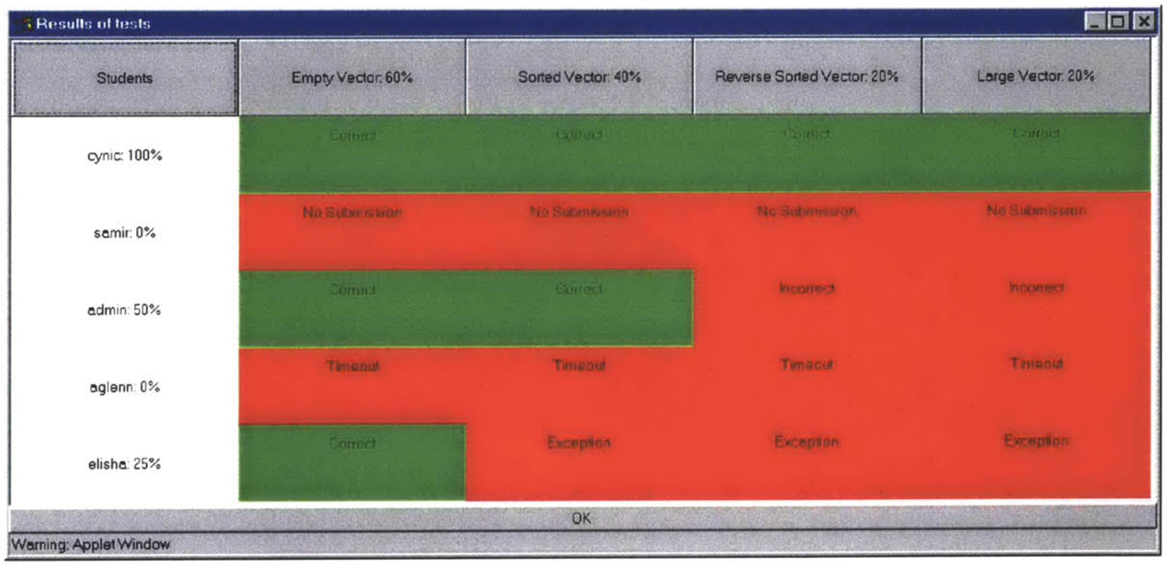
It is important to note that the timeout error is a catchall and can have several causes. First, the code might be inefficient. If the module is not intended to teach students to create efficient algorithm implementations, users can simply set the timeout threshold to be a greater number. Second, the user might have written an implementation of a slower algorithm that performs the same function. For instance, the student might have implemented insertion sort when he was supposed to implement counting sort. This error, which is important to discover, can be separated from the first timeout error if the test suite contains a test case with appropriately large input and the timeout period is long. Finally, the code may contain an infinite loop. No matter what the cause of the timeout error, the system will allow the code to run only as long as the specified timeout period. If the code is still running at the end of the timeout period, it is logged as a timeout error and terminated, regardless of whether the algorithm's output is correct.

The functionality of the Run Tests button is very similar on the student window. The only difference is that instead of running the users' submitted code, the user can select any version that he has compiled so far. In either case, the system will display a progress bar as the tests run, and will open a ReportForm or ReportFormStudent, as appropriate, when the tests are completed.

Finally, each window has a function that is unique to it. The student window has the Submit button, which takes the selected test cases and sends their data to all members of the teaching staff through the help queue of the existing collaboration system. This allows students to submit interesting test cases for inclusion in the generic test file. Staff may choose to accept this test for inclusion in the standard suite of cases by adding it in the normal fashion, reject it by simply doing nothing, or they may respond with an appropriate message. Alternatively, teachers can create a standard test suite that leaves out some interesting case, and assign students to submit one that they believe completes the testing process. On the staff window, there is a Set Code button. This button allows the teaching staff to set the user implementation on his panel to the selected user's submitted code. The staff member may then run tests manually on this implementation.

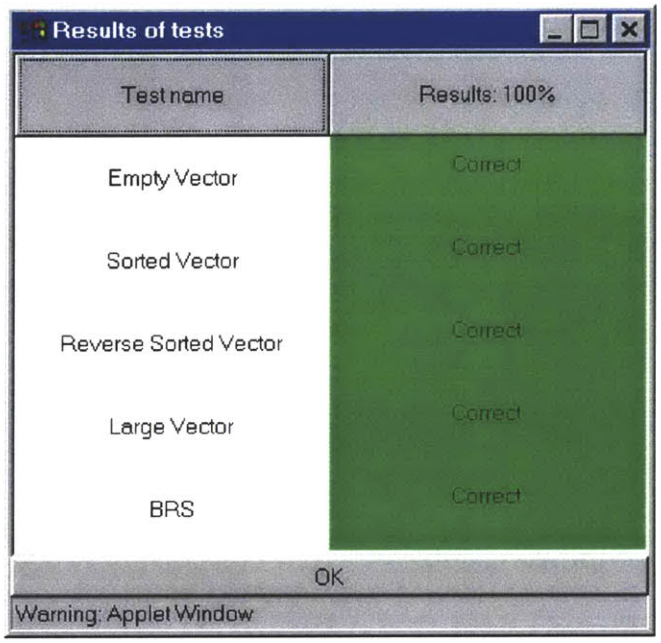
3.3.3.2 ReportForm and ReportFormStudent

As their names suggest, the ReportForm and ReportFormStudent windows deliver the results of the tests that were run; the former is used for tests that were run by teachers, while the latter is used for tests run by students. Figure 3-5 shows a typical window generated by a member of the teaching staff running the standard test suite on multiple students, which includes a sample of every possible error, while Figure 3-6 shows a window generated by a single student running his own personal test suite.



Students	Empty Vector: 60%	Sorted Vector: 40%	Reverse Sorted Vector: 20%	Large Vector: 20%
cynic: 100%	Correct	Correct	Correct	Correct
samir: 0%	No Submission	No Submission	No Submission	No Submission
admin: 50%	Correct	Correct	Incorrect	Incorrect
aglenn: 0%	Timeout	Timeout	Timeout	Timeout
elisha: 25%	Correct	Exception	Exception	Exception

Figure 3-5: Results generated by running four tests on five students


Results of tests

Test name	Results: 100%
Empty Vector	Correct
Sorted Vector	Correct
Reverse Sorted Vector	Correct
Large Vector	Correct
BRS	Correct

OK

Warning: Applet Window

Figure 3-6: Results generated by a student running five tests

The layout of the windows is fairly simple. In both versions, the results of the tests are color-coded: green for correct tests, red for any tests that did not result in correct output. Also, the top of each results column displays the percentage of tests that succeeded. This is so that the user can get a general idea of the results at a glance. Additionally, the exact error is displayed, so that the user knows what specifically went wrong. This may help students fix their code.

The student version contains two columns: a list of the tests that were run (on the left side), and the respective results (on the right). Clicking on either the Test Name button or the Results button will sort the window by that column, either alphabetically for tests, or correct results first for results. Clicking on the same button twice will reverse the sorted order.

The staff version contains the names of the students and their test success rates in the rows and the names of the tests in the columns. Like ReportFormStudent, the Students button sorts the results alphabetically by student and the individual test buttons sort the results by the outcome of the appropriate test.

3.3.3.3 NameTestForm

When a user wants to add a test to the test suite, he clicks on the Add button in the appropriate version of the RunTestForm window. This button reads the data from the visualization panel, but that is not sufficient information for adding a test. The test case subsystem also needs to know the name of the test and, for teachers, whether the test should be visible or opaque.

Therefore, when the user clicks on Add, the system creates a NameTestForm to get this data (see Figures 3-7 and 3-8). Due to the way that the test case file is parsed, the user cannot use any special characters in the test case name, but he is able to use any combination of letters, numbers, and spaces. He then can select whether the test is opaque, if he is a teacher. When the user clicks on Add, the form determines whether the user is a teacher or a student, and makes the addition to the appropriate file.

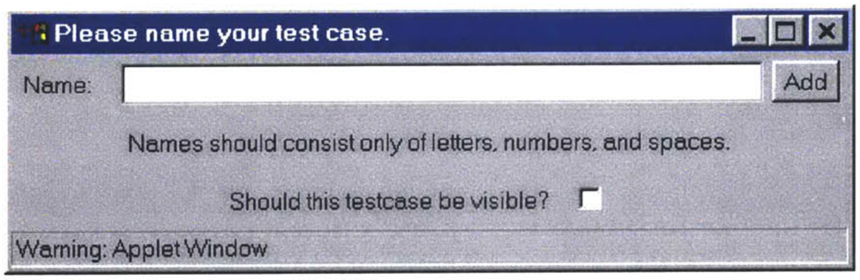
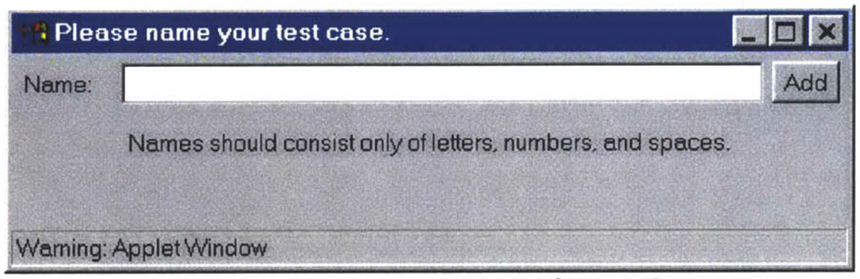

Please name your test case.
Name:
Add
Names should consist only of letters, numbers, and spaces.
Should this testcase be visible?
Warning: Applet Window

Figure 3-7: NameTestForm generated for a staff user



Names should consist only of letters, numbers, and spaces.

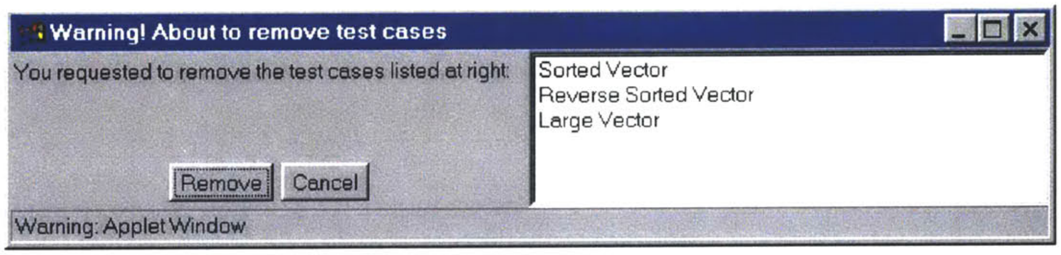
Warning: Applet Window

Figure 3-8: NameTestForm generated for a student user

3.3.3.4 RemoveDialog

When a user clicks on the Remove button, he will be permanently removing the test cases in the appropriate test file. Since Fusion is not in charge of the file system on the server that runs Fusion, there is no undo function. Putting removed test cases in separate storage space would simply add clutter without enhancing usability. Also, removals made by teachers will be propagated out to all students running tests. Therefore it is imperative that tests not be removed by accident.

It is sufficient simply to ask the user to confirm a test case deletion. As such, when the user clicks on Remove, he is presented with a RemoveDialog, which lists all of the test cases that he has chosen to remove, and asks him to confirm this removal. If the user then clicks on Remove again, the RemoveDialog accesses the appropriate file and deletes the chosen test case or test cases. Figure 3-9 shows a RemoveDialog form.



You requested to remove the test cases listed at right:

Sorted Vector
Reverse Sorted Vector
Large Vector

Warning: Applet Window

Figure 3-9: RemoveDialog form

Since students should always be provided with a standard suite of test cases, the test case file has a flag called personal. A test is flagged personal if and only if it was created by a student, in which case it can be deleted by that student. Otherwise, it is part of the standard suite of tests, and can only be deleted by a teacher.

3.3.3.5 AlgorithmThread

Student code may timeout under two conditions: either their code runs so slowly on some input that it takes longer than the specified time period to timeout, or their code contains an infinite loop. It is impossible for code to test for an infinite loop, as this is the well-known Halting Problem. Therefore, it is important that any test code run in a separate Thread, so that if it does timeout, the system can stop its execution and continue running tests. The system uses AlgorithmThreads to start and stop execution of student code.

3.3.3.6 TestcaseParser

Both the student test case file and the teacher test case file are written in specific formats (see Appendix D, Sample Test Case File, for an example of this format). In order for the test case classes to make sense of the format, there must be a way to read in the data and extract the necessary information. These methods are called by instantiating a TestcaseParser and using its methods to return the desired data. A TestcaseParser is based on a StreamTokenizer, which it uses to parse the file. It should be noted that the test case file can be read as a String, and so TestcaseParser could be based on the simpler class StringTokenizer. However, StreamTokenizer is more powerful and flexible.

3.3.3.7 ErrorDialog

Several functions of the test case subsystem have specific requirements. For instance, students cannot remove standard tests, and neither students nor teachers may display more than one test at a time. If a user violates the requirements of the action he is trying to perform, the system alerts him to this violation with an ErrorDialog.

3.4 Impact on Module Design

One of the objectives of Fusion is usability, even by weak programmers. This is especially important in the case of teaching staff trying to create modules. Module creation must be efficient, so as to encourage the creation of a library of reusable modules and visualization panels. Therefore, the additional work required to create a module, regardless of whether it supports test cases, should not be a barrier to module creation.

The implementation described makes no changes to the process of creating a module that does not support test cases. However, implementing test case support does require some additional code. The specifics are covered in Appendix A, Modified Module Creation Checklist, but it is important to analyze the effort required for making these changes so that it can be determined whether module creation is too hard.

First, the module designer must make FindWitnessList set the value that specifies whether the last test succeeded or not. As is shown in Appendix B, this change is usually as simple as one or two lines of code, added after the method is ready to return the Vector of incorrect elements that FindWitnessList normally returns.

Second, the module designer must write the test case support functions. The method **supportTests** simply needs to return true instead of false, which is trivial. Next, the designer must implement **populatePanel, readPanel,** and **clearUI**. Since the module designer knows exactly what user interface elements are being placed in the panel, and he has written a method that includes the creation of a data object from the user interface in **runAlgorithm**, the method **readPanel** should be easy. Once the designer has implemented readPanel, he knows exactly what data is going to be returned from the test case file. Combined with his knowledge of the user interface elements, it should be simple to write populatePanel. Creating the clearUI method simply involves setting the state of each user interface element back to its initial state, which should be easy if the designer knows what elements exist. Finally, the designer must implement the method **runWitnessCode**. This method is simply a call to FindWitnessList, with the proper object. Often, the proper object is just the current output. This is the default implementation, in which case the designer need do nothing. However, if the module does not store the current output, the designer simply needs to call FindWitnessList on the object that represents the output. He must already do this in order to use FindWitnessList elsewhere in his module to support witness detection. Appendix C contains an example of the simplicity of adding test case support functions for the existing module VectorSort.

Finally, the module designer may create the standard test case suite. The designer may choose not to do so, or to use the teaching staff interface for creating new tests, both of which involve negligible effort. The designer, if he so chooses, may also create the file by hand, which simply involves creating a file in the proper directory and writing text in that file according to a specified format. An example of this format can be found in Appendix D, Sample Test Case File.

4 Conclusion

Thus far, a review of Fusion's original goals, and how test cases satisfy those goals, has been discussed. An overview of work in the Web-based teaching and the Java code testing realm has also been shown. Finally, a presentation of Fusion's test case subsystem's design, features, and implementation has been given. This chapter concludes the thesis by determining whether the test case subsystem is in fact detrimental to students' learning. It then suggests future Fusion enhancements and extensions, and closes with some final thoughts.

4.1 Do students have it easy?

It may seem that the test case subsystem, while satisfying all the original objectives of Fusion, makes writing correct code too easy. Students with a deficient understanding of the underlying algorithm may simply take a first stab at writing code, then iterate the process of running all the tests and writing different code, using brute force and time to compensate for their lack of comprehension.

While this is certainly a possibility, the test case subsystem on its own does not trivialize the importance of understanding. A carefully constructed test suite, with some opaque test cases and some important tests missing entirely, can be run any number of times by a student without allowing him to write completely correct code. At best, he can only write code that will satisfy the given tests, and his lack of understanding will be apparent when the missing tests do not run correctly. Alternatively, teaching staff may require all students to submit as an assignment what the student believes to be a missing test. This is not possible without the test case subsystem and clearly aids students in comprehension of the algorithm.

Also, the test suite is limited by the power of the witness detector. As such, the results can only determine whether the output on the given tests is correct, not whether the underlying implementation is correct. A student might implement counting sort, for instance, when he is asked to implement bucket sort, and the test suite would not know the difference. It is still up to the student to understand and implement the correct algorithm.

Finally, it is important to remember that Educational Fusion is intended to teach material. It is not meant to educate students in the intricacies of testing their code, though a module to do so could certainly be created and facilitated using the test case subsystem. Students should already be in the habit of verifying their code for correctness, and when they write professional code, they will have access to test frameworks similar to the one implemented in Fusion. Thus, with proper test suite construction, the benefits of the test case subsystem can be fully utilized without the fear of making code authoring too easy for students.

4.2 Suggested Future Work

Educational Fusion in its current state is a powerful and highly refined tool for teaching and learning algorithms. Taken as is, Fusion has been tested in a typical classroom setting and found to be an effective tool to augment traditional teaching methodology. The test case subsystem continues this trend by staying true to the original objectives of Fusion. It adds many convenient features that will be extensively employed by Fusion users. However, Fusion still has some way to go before it begins to tap its full potential. Some extensions and enhancements to the Fusion infrastructure and interface are enumerated here.

4.2.1 Flexible back-end

As Fusion was originally conceived as a platform for teaching and learning algorithms, its infrastructure is based on providing a useful framework for algorithms. However, as Fusion continues to grow and expand, its users may wish to embed other material and methods in it, in order to get features like collaboration and test cases for free.

Facilitating these extensions is a difficult proposition, since to do so most effectively requires knowledge of what extensions the user wishes to implement. However, there are currently two kinds of extensions that a user is likely to want. First, the module subsystem is geared towards teaching algorithms, but should be flexible enough to allow for authoring other modules. For instance, the recent work on supporting simulations and remote labs is a step in the right direction.

Second, and perhaps more challenging, Fusion's back end should have hooks for external programs. These hooks would allow Fusion to tie into programs that are available anywhere within the educational institution's computing resources. A simple usage of this ability would be to give students a module that simulates a circuit, then asking the students to find the parameters that cause certain behaviors in the circuit. Once those parameters are found, the student would be able to open a math program, say MATLAB, and graph some objective value as it varies with the parameters. A more ambitious use of this feature might be a scenario as complex as asking students to implement a fast line-drawing algorithm, then opening a graphics program and using the algorithm to speed up that program's subroutines.

4.2.2 Database repository

Work on Fusion began in the mid-1990's, before the world really knew about e-commerce and proper Web services design. Now, however, there are powerful tools for creating, maintaining, and serving a Web-hosted service to the targeted users. One such tool is a commercial database for storage purposes.

There are several reasons why installing a database repository in Fusion is a good idea. First, any serious Web service is backed by a powerful database solution. This is perhaps owing to the second reason, which is that a database provides many desirable qualities all at once, including security, user access control, reliability, and scalability. All of these features are currently lacking in Fusion, which stores its data in plain-text files on the Fusion server. Lastly, a database simplifies the storage of otherwise clumsy things, such as Java data objects.

4.2.3 Randomized test case creation

The current implementation of test cases addresses the need for students and teachers to have access to a well-defined suite of tests. However, sometimes it can be useful to simply generate a pseudorandom piece of data and run it through a user's algorithm, just to see if it works properly. This might also be desirable to test the flow of data in a complex Concept Graph, to make sure data is being directed to the correct inputs and outputs, and that the resulting flow arrives at the desired conclusion.

Therefore, Fusion should support a method for adding random test cases. When this option is selected, Fusion will create a module whose reference implementation will be to create instances of these test cases and pass them through its output. In this way, users can quickly run several random tests on their algorithm, or view the workings of an algorithmic pipeline on several different inputs.

4.3 Final notes

As has been noted several times already, Educational Fusion is already in possession of robust infrastructure and a very practical and usable feature set. With the addition of the test case subsystem, Fusion has added another dimension of convenience and possibilities to algorithm learning. There is a great deal of potential, both for different uses of Fusion and for other exciting additions to Fusion's capabilities. Those working on and observing the progress of Fusion have every reason to be optimistic about its future, as the development of Educational Fusion propels it ever closer to being the universal learning environment of the new millennium.

A Modified Module Creation Checklist

This appendix lists additions and changes to the canonical Module Creation Checklist given in Josh Glazer's M.Eng thesis [Gla00]. For further explanation of steps, please reference that document, or Aaron Boyd's M.Eng thesis [Boyd99].

1. Create the reference, base, and student modules.
2. Create the witness detector.
    a. Note that the function **public Vector FindWitnessList(Object input, Object output)** is now required to set the boolean value **lastTestSucceeded**, for those modules that support test cases. Generally speaking, the easiest and most accurate way to do this is to simply add a line in the code that checks for errors; if an error is found, set the boolean to false, but if no error is found, set it to be true. See Appendix B, Modifying FindWitnessList, for a before-and-after example.
3. Create the visualization panel.
4. If you do not wish the module to support test cases, you may stop here. Otherwise, you must write five methods. Examples of such methods are given in Appendix C, Sample Test Case Methods.
    a. Override **public boolean supportsTests()** so that it returns true.
    b. Override **public void populatePanel(Vector data)** so that, given a Vector of Strings, it can place appropriate values into the panel. When this function is complete, the panel should have all the input that is required for running the

algorithm. Also, when this function completes, a user looking at the panel should be able to determine the exact parameters that make up this test case.

c. Override **public Vector readPanel()** so that when run, it returns a Vector of Strings. These Strings represent the data that is the current input on the panel. If one were to invoke **populatePanel(readPanel())**, the panel should have no change in state.

d. Override **public void clearUI()** so that when run, it restores the panel to the state which the user sees when first opening the panel from the ConceptGraph. Alternatively, the module designer may choose to have this method restore to some other state. However, if the designer wishes test cases to be opaque, the method should erase all traces of any test cases that were run previously.

e. Override **public void runWitnessCode()** so that FindWitnessList is called on the appropriate object. In some situations, this may be as simple as the code **FindWitnessList(getCurrentOutput())**.

f. See Appendix C, Sample Test Case Methods, for a listing of code that was required to make the VectorSort module support test cases.

5. Create the default tests file. The following describes the process for manually creating the file; however, it is simpler to log in as a member of the teaching staff, open the desired module panel, enter the required input, open the RunTestForm, and click Add.

    a. Navigate to the appropriate module directory. For a typical Fusion installation, this will be the root Fusion directory (by default, \Fusion\Root), plus the module directory (\projects\graphics\modules\module_name\\).

b. Create a new ASCII text file, called tests.txt.

c. Each line of this test file is a separate test case. In each line there are four pieces of data. Create each line exactly as specified below; white space and case are significant. For each line:

   i. Write "name=" without the quotes, then the name of this test case. Valid names may contain numbers, letters, and spaces, but no other special characters. If a name begins with a number, it may only contain numbers.

   ii. Without adding spacing, write "|opaque=" without the quotes, then either the word "true" or "false", depending on whether students should be able to see the contents of this test case. Teaching staff can always view test cases regardless of this designation.

   iii. Write "|data=" without the quotes. Then, for each piece of actual test case data that is returned by the function **readPanel**, write some data, surrounded by quotes. For instance, if **readPanel** returns four integers, this part would look like "*int1*" "*int2*" "*int3*" "*int4*".

   iv. Write "|personal=false" without the quotes.

   v. Insert a carriage return (new line).

   vi. See Appendix D, Sample Test Case File, for a listing of a possible text file that would be used to generate test cases for the VectorSort module.

B Modifying FindWitnessList

This appendix contains the full listing, sans comments, of two FindWitnessList methods. These methods were taken from the class ProofVectorSort, which is a witness detector for the VectorSort module. On the left is the current implementation, which supports test cases by setting the boolean value lastTestSucceeded to indicate the success or failure of the given Object o to represent the correct output for the Object i. On the right is the original version of FindWitnessList. The changes required to support test cases are highlighted in bold, and are clearly not difficult to implement.

```
public Vector FindWitnessList(Object i, Object o) {
  Vector badVec=new Vector();
  AnimatedVector vo=(AnimatedVector)o;
  int k;
    if(vo!=null) {
      for(k=1;k<vo.size();k++) {
        if(((Integer)vo.elementAt(k-
1)).intValue()>((Integer)vo.elementAt(k)).intValue()){
          badVec.addElement(new Integer(k-1));
          badVec.addElement(new Integer(k));
        }
      }
    }
    if (badVec.isEmpty())
      lastTestSucceeded = true;
    else
      lastTestSucceeded = false;
   return badVec;
}
```

```
public Vector FindWitnessList(Object i, Object o) {
  Vector badVec=new Vector();
  AnimatedVector vo=(AnimatedVector)o;
  int k;
    if(vo!=null) {
      for(k=1;k<vo.size();k++) {
        if(((Integer)vo.elementAt(k-
1)).intValue()>((Integer)vo.elementAt(k)).intValue(
)){
          badVec.addElement(new Integer(k-1));
          badVec.addElement(new Integer(k));
        }
      }
    }
   return badVec;
}
```

C Sample Test Case Methods

This appendix contains a full annotated listing of all the test case support methods for the VectorSort module. This code can be found in the class VecSortPanel. No other changes to the VectorSort module were required to make it support test cases.

```
public boolean supportTests(){
      return true;
}

public void populatePanel(Vector data){
      mListField.setText((String)data.elementAt(0));
}
```

mListField is the name of the text entry field where users can input numbers to create a Vector for sorting. This method takes in the generic Vector of data read from a test case file, in which it expects to have a single element that is a String of numbers. It then places that String in the text field.

```
public Vector readPanel(){
       Vector returnVec = new Vector();
       returnVec.addElement(mListField.getText());
       return returnVec;
}
```

This method creates a data Vector, which it then passes to Fusion for writing to a file. To do so, it first creates a new Vector object. It reads the user interface, which is in this case a single text field, and inserts the data from the interface into the Vector. Finally, it returns the Vector.

```
public void clearUI(){
      int currMode = getMode();
      mListField.setText("0");
      setMode(VizModeController.MANUAL_MODE);
      runAlgorithm();
      mListField.setText("");
      setMode(currMode);
}
```

The clearUI method is the only method that is a little bit tricky. The problem arises because the VecSortPanel retains the output of the last run of the algorithm on its display until the user requests another run. However, it would make no sense to create some dummy input and run the algorithm on it, just to have a different picture in the window. The solution is to create a Vector consisting of a single integer, 0. Then, the panel is run in manual mode, so that the display now shows a Vector with a single element of no size, or a blank screen. Finally, the mode is set back to whatever mode the panel was in before the call to clearUI.

```
public void runWitnessCode(){
      FindWitnessList(getCurrentOutput());
}
```

Finally, this method calls FindWitnessList on the current output. This is the default implementation of runWitnessCode, which is inherited by all visualization panels from DefaultTeacher, and is only reproduced here for completeness.

The above methods were written by a Fusion system designer with no prior knowledge of the VecSortPanel code other than what is discernible from the panel displayed in Fusion. The total time it took to do so was about twenty minutes. It is expected that a module designer creating these methods while in the process of implementing a new module be able to do so in the same time, if not less.

D Sample Test Case File

The following is a sample ASCII text file, tests.txt, which lists four test cases for the VectorSort module. Note that the Large Vector test case is contained wholly on one line and merely appears broken here to fit within the margins.

```
   name=Empty Vector|opaque=true|data = ""|personal=false
   name=Sorted Vector|opaque=true|data = "1 2 3 4 5"|personal=false
   name=Reverse Vector|opaque=true|data = "5 4 3 2 1"|personal=false
   name=Large Vector|opaque=true|data = "13 6 9 12 2 10 4 3 1 8 15 5 7
11 14"|personal=false
```

Also note that this is a typical generic test file, as all the cases are non-personal. In this case, all the test cases have been designated opaque. This might happen in a class where the teaching staff releases the test cases used for grading to the students so that they can check their own work, but a generic test case may contain non-opaque test cases as well. Finally, it is crucially important that there be a carriage return at the end of the last line, as though one were going to start a new line. Otherwise, Fusion will have difficulty parsing the file.

Bibliography

[Boyd97] Boyd, Nathan D. T. "A Platform for Distributed Learning and Teaching of Algorithmic Concepts." MIT Thesis. 1997.

[Boyd99] Boyd, Aaron T. T. "Educational Fusion: A Distributed Visual Environment for Teaching Algorithms." MIT Thesis. 1999.

[Chao00] Chao, Wesley S. "Fusion Extensions". Internal report, MIT Laboratory for Computer Science. 2000.

[Gam+01] Gamma, Erich and Beck, Kent. "Junit, Testing Resources for Extreme Programming." *http://www.junit.org*. JUnit.org, 2001.

[Gla00] Glazer, Joshua E. "Improved Algorithm Visualization and Witness Detection in Educational Fusion". MIT Thesis. 2000.

[Gol+01] Goldberg, Murray, et al. "WebCT Products." *http://www.webct.com/products/*. WebCT, 2001.

[Harr+99] Harrison, Lou, et al. "WolfWare." *http://www.csc.ncsu.edu:80/info/wolfware/www/proposal/proposal.html*. North Carolina State University, 1999.

[Kas98] Kasperowski, Richard. "Automated Testing and Java Class Libraries," published in Conference Proceedings: Eleventh International Software Quality Week, San Francisco, May, 1998. Also available at *http://www.altisimo.com/research/automated-testing-and-java.doc*.

[Kas99] Kasperowski, Richard. "The Design and Implementation of a Java Test Driver," in Proceedings of the 16th International Conference and Exposition on Testing Computer Software, Washington D.C., 1999. Also available at *http://www.altisimo.com/research/design-implement-test-driver.html*.

[Ker01] Kerns, Charles. "CourseWork." *http://www.stanford.edu/group/ats/coursework/*. Stanford University, 2001.

[KST99] Kehoe, Colleen, John Stasko and Ashley Taylor. "Rethinking the Evaluation of Algorithm Animations as Learning Aids: An Observational Study." Technical Report GIT-GVU-99-10. Georgia Institute of Technology, College of Computer Science, 1999. Also available at *ftp://ftp.cc.gatech.edu/pub/gvu/tech-reports/1999/99-10.ps.Z*.

[LBS94] Lawrence, Andrew W., Albert N. Badne, and John T. Stasko. "Empirically Evaluating the Use of Animations to Teach Algorithms." Technical Report GIT-GVU-94-07. Georgia Institute of Technology, College of Computer Science, 1994. Also available at *ftp://ftp.cc.gatech.edu/ pub/gvu/tech-reports/1999/94-07.ps.Z*

[Por98] Porter, Brandon W. "Educational Fusion: An Instructional, Web-based, Software Development Platform." MIT Thesis. 1998.

[Sza+01] Szabo, Victoria E. "PanFora." *http://panfora.stanford.edu/.* Stanford University, 2001.

[Tel94] Teller, Seth. "NSF Career Development Plan." *http://graphics.lcs.mit.edu/ ~seth/proposals/cdp.ps.* MIT Department of Electrical Engineering and Computer Science, 1994.

[Tel+98] Teller, Seth, Nathan Boyd, Brandon Porter, and Nick Tornow: Distributed Development and Teaching of Algorithmic Concepts, in Proc. Siggraph '98 (Educational Track), July 1998, pp. 94-101. Also available at *http://edufuse.lcs.mit.edu/fusion/papers/siggraph98/siggraph98.html.*

[Tor98] Tornow, Nicholas J. "A Distributed Environment for Developing, Teaching, and Learning Algorithmic Concepts." MIT Thesis. 1998.

[Zac+97] Zachary, Joe, et al. "Hamlet Project." *http://www.ct.utah.edu/~hamlet.* University of Utah Department of Computer Science, 1997.