Structural Design of

Efficient Automated GUI Testing/Operations

by

Young K. Kim

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 20, 2000

[= JUNE 2000]

Author_____
Department of Electrical Engineering and Computer Science
May 20, 2000

Certified by_____
William Long
Thesis Supervisor

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

Structural Design of Efficient Automated GUI Testing/Operations
by
Young K. Kim

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2000

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

ABSTRACT

This thesis exhibits a way to implement automated GUI testing and other operations robustly and efficiently. The design is built upon the distributed architecture of Java Swing and Accessibility API, which establishes connectivity between components. We run a demonstration on an existing GUI application. The software architecture of automated GUI testing/operations brings about robustness, fast performance, flexibility in maintenance, simplicity for users, and support for multiple platforms without requiring any extra hardware support. It also enables scripts to be readable and maintainable and provides broad test coverage.

Thesis Supervisor: William Long
Title: Research Associate, MIT Laboratory for Computer Science

Acknowledgement

I would like to extend my warmest thanks to my thesis and academic advisor, Dr. William Long. He has guided me through the thesis work from the beginning to the end. He has offered many thoughtful ideas and much advice on the content and given me a tremendous help in revising and completing the work. He also was a great academic advisor in my 5-year MIT career. His caring and considerate counsel, time after time, has provided me with the confidence and motivation to finish the rigorous demands of MIT. Thank you, Bill.

My thanks also extend to Paul Roush, my 6A supervisor at Teradyne, Inc. He has given me a considerable amount of help in designing the system and oversaw my progress as the term went on. He also advised me through researching the automated testing industry. For my three summers and a term, he has directed the projects I worked on and helped me to feel at home with the group. Thanks a lot, Paul.

I would also like to thank George Patterson and other workers at the software group of Memory Test Division in Agoura Hills, CA. They have made my experience an exceptionally pleasant, fun, educational, and eye-opening one. They have been a great help for me--from the Java Accessibility API to a local steak house, they have helped me and taken me as a close friend. Thank all of you at MTD-SW. You are all great.

As I close up my career at MIT and my 5-year stay in Boston, I would also like to thank all those who have loved me and become an important part of my life. Brothers of Phi Kappa Theta: you all are amazing. Let's continue to pursue life. PB and NCPC: how you have been used to teach, love, and challenge me is indescribable. May the Lord continue to be praised in your midst. NCPC99: thank you for being a family. NCPC Praise: we shall never stop. KCF: thanks for your passion and genuineness. My partner: thank you for your prayers. May He continue to mold you into a godly man. I am excited about many years ahead of us.

I would also like to shoot my deepest thanks to my family: Mom, Dad, Connie, James, and Sunghwan. Thanks for all your support and prayers that have sustained me so far. I thank God for you each day and I look forward to even more satisfying time with you as I head back to California.

Most important of all, I would like to thank my Lord and Savior, who is real and true. I cannot even start to put into words what He has been and done for me and all others. Thank you. Thank you. Like all others, this one is for you.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

The software world keeps on expanding. The number of software users is higher today than yesterday. It has become a critical part of today's life. As more people are entering the software world, the GUI (Graphical User Interface) has come to require its share of attention. Better visibility, animation, and ease of use are a few of the motivations for GUI programmers to involve in more complex development processes. Hence, the GUI testing has become more complex and voluminous and the need for automating this testing process has surfaced as one of the critical issues in fast and robust GUI development.

1.1. What is a GUI?

A GUI is a graphical, rather than all textual, user interface to a computer. Before GUIs came into existence, there were only textual and keyboard-based interactive user interfaces such as the DOS operating system. After the command line interface came the non-graphical menu-based interface, which lets you interact by using a mouse rather than by having to type in keyboard commands.

Today's major operating systems supply graphical user interfaces. Application developers typically use the elements of the GUI that come with the operating system and add their own GUI elements and ideas. A GUI sometimes uses one or more images for objects familiar in real life, such as the desktop or the view through a window. Elements of a GUI usually include windows, pull-down menus, buttons, scroll bars, text fields, iconic images, wizards, tables, and the mouse, with many other things still to be invented. With the increasing use of multimedia as part of the GUI, sound, voice, motion video,

and virtual reality interfaces seem likely to become part of the GUI for many applications. A system's graphical user interface along with its input devices is sometimes referred to as its "look-and-feel."

1.2. GUI testing

As GUIs have become an integral part of practically all applications today, testing GUIs has become a demanding challenge. Manual testing of GUI-based applications is slow, tedious, error-prone, non-repeatable, and expensive. It can no longer keep pace with this rapid development environment where repeating long procedures of operations is a standard part of the software development process.

Every possible access method and scenario needs to be tested in order to ensure the reliability of an application. In addition, in maintenance, regression testing must be performed on modified programs to provide confidence that changes made are correct and have not affected unmodified portions of the software. This dramatically increases testing complexity and the amount of testing that must be done, making automated testing the only viable alternative. By automating the GUI testing, a dramatic gain in speed, robustness, and productivity can be achieved.

1.3. What is regression testing?

Although software may have been completely tested during its development, program changes during maintenance require that parts of the software be tested again. In fact, the maintenance process tends to be the most costly part of software production--current research indicates that software maintenance activities account for as much as

two-thirds of the cost of software production. One major and necessary but expensive maintenance task is regression testing. Regression testing is the process of verifying modified parts of the software and checking that the modifications do not cause any new errors in already tested code. An important difference between regression testing and development testing is that during regression testing an established set of tests may be reused. One approach to reusing tests is to rerun all such tests (retest-all), but this strategy may consume extensive time and resources. An alternative approach, selective test, chooses a subset of the old test set and uses this subset to test the modified program.

There are three main groups of selective retest approaches: minimization, coverage, and safe. Minimization approach identifies and runs a minimal set of tests that must be rerun to meet a minimal test coverage criterion. For example, a minimization approach might select one test in every modified basic block, providing statement coverage of the changed code. The coverage approach is based on coverage criteria, like minimization approach, but it does not require minimization. Instead, it seeks to select all tests that exercise the changed or affected program component. The safe approach places less emphasis on coverage criteria, and it attempts to select every test that will cause the modified program to produce different output than the original program.

Nonetheless, regression testing is one of the most expensive and time-consuming tasks to be done manually. By automating this process, we can achieve a remarkable improvement in efficiency, efficacy, and cost.

1.4. Thesis outline

This thesis presents a robust and efficient way to implement the process of automating GUI testing and any other GUI operations. There is work done and being done in this area of GUI testing, and the thesis discusses current and previous work and its successes and failures. MVC (Model-View-Controller) and Java Swing architectures are incorporated in our design and described. The Java Swing Accessibility feature is applied to create a component-based software architecture design that is sufficient to implement the automated GUI testing process. The thesis then demonstrates the design on an application written in Java and, finally, recommends future work and improvements.

2. Background

The proliferation of GUIs in virtually all applications under development today presents an unprecedented challenge to testing professionals. Manual testing of GUI-based applications is slow, tedious, error-prone, non-repeatable, and expensive. By contrast, automated GUI testing can be dramatically faster, yielding substantial productivity gain.

2.1. Motivation

For an Automated Testing Equipment (ATE) manufacturer, Teradyne manages and operates extensive data generated from the hardware. To work with it, Teradyne has created software tools to interface with the data, including multiple GUI applications that are user-friendly and functional. However, because of the size of the data and the

numerous functions that the software supports, the number of GUI components is substantial.

A recent version of Teradyne's memory tester software includes 13 graphical tools. Each tool contains about 1000 widgets and 25,000 lines of code. This totals 13,000 widgets and 325,000 lines of code in the software package. This is pretty sizable and is only the GUI part of the software. As mentioned earlier, testing requires a major part of software maintenance, and we can see that doing manual regression tests is very tedious.

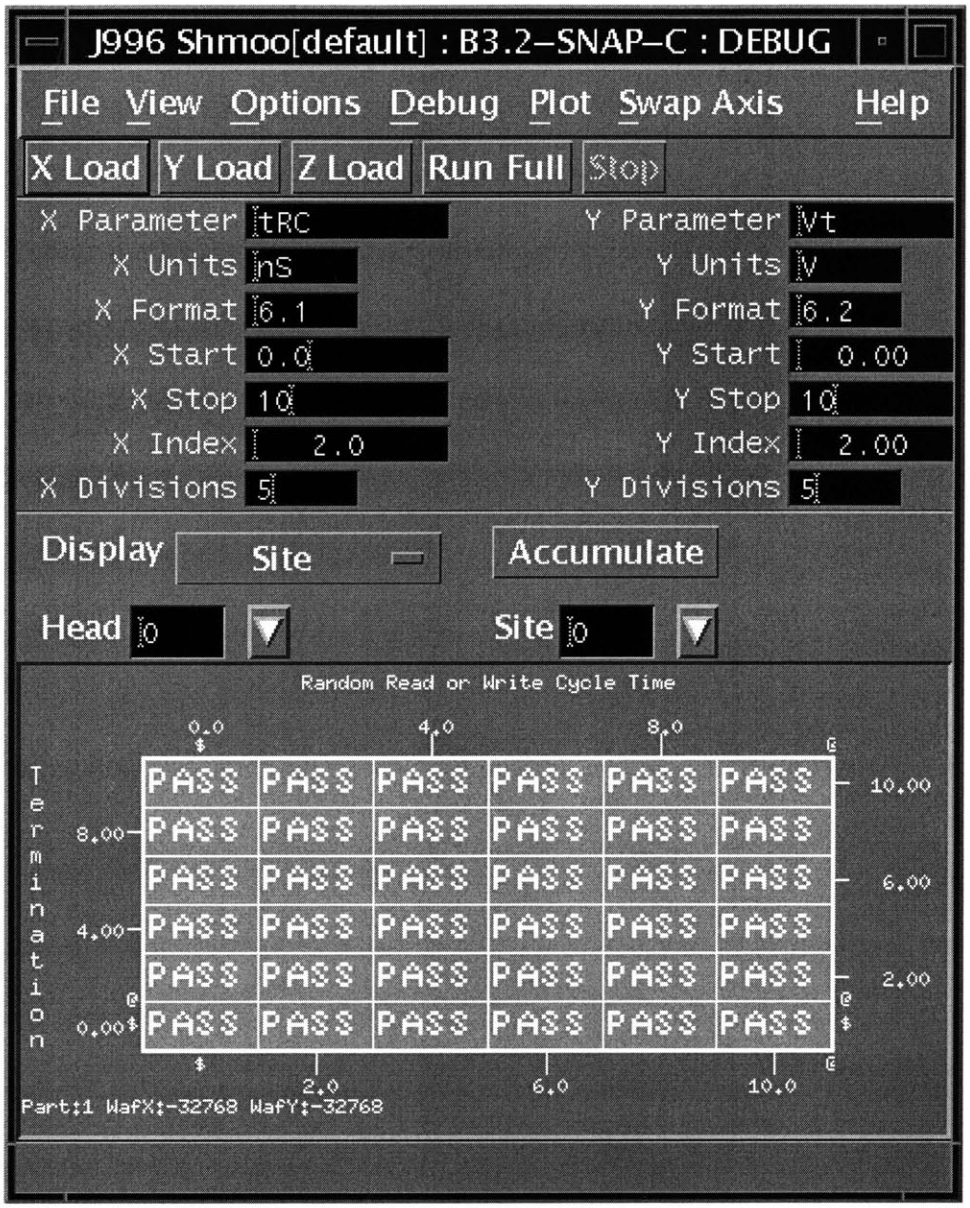

Figure 1. Shmoo GUI tool.

One of the GUI tools that Teradyne's engineers test by regression is Shmoo [15]. In the Memory Test Division where the development of memory chip testers takes place, the Shmoo tool displays the Passes and Fails on tests run on memory chips, given specifications and their ranges of values. In Figure 1, tRC and Vt specs are loaded in X and Y Parameters, and a range of 0 through 10 is defined for both parameters. Although

Figure 1 shows all Passes, sometimes the values of memory tests are Fails for a certain range of spec values—these are all defined by Teradyne's memory chip manufacturing customers. Therefore the Shmoo GUI tool must show correct values, whether they would be Fails or Passes, for all specs and for all ranges.

An example of a regression test on this tool is to go through all the specs for X and Y Parameters and test whether they produce correct values. This regression test would take place if there were a change in one of the control fields such as X Stop, Y Start, etc. A test engineer would begin by clicking on View in the Shmoo GUI tool menu bar and select Spec Symbols. A window (Figure 2) will pop up with a list of all the specs. The engineer would then click on a desired spec and on either X Load or Y Load (ignore Z Load). The RunFull button in the main Shmoo GUI tool then would be pressed and the result would be checked. Because the large number of specs requires numerous regression tests, the process can be extremely repetitive and tedious.

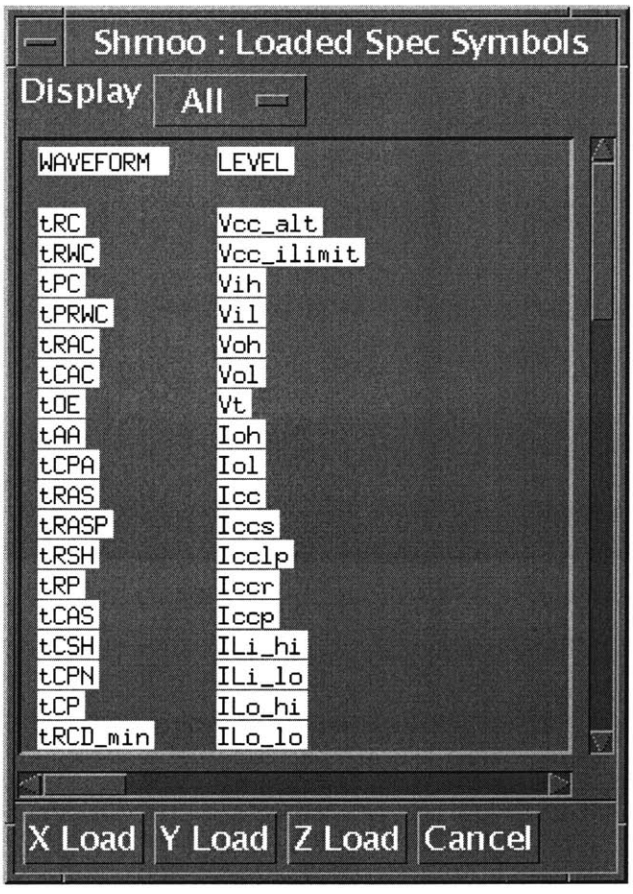

Figure 2. Load Spec Symbols window.

This example is just one of an enormously large number of regression tests required in Teradyne GUI tools and other business processes in the world today. Automating the repetitive and time-consuming—therefore expensive—processes not only means major improvement in performance, but also productivity and overall business profit.

2.2. Previous Work

Considerable research and development has been done in this area. However, the third party vendor products do not completely fulfill the requirements of an automated GUI testing that Teradyne needs. Two major approaches have been taken by most of the today's products. However, neither approach would satisfy our testing criteria (read the criteria in section 4, Problem). Some of the products we looked into are Rational's PreVue [11], Mercury Interactive's XRunner [10], and CenterLine's QC/Replay [12].

These products use one of the two approaches to automate functional testing of X Window-based applications. The first approach is to record and reproduce tests graphically, while the second is a more widget-based approach.

PreVue-X automates both GUI regression testing and load testing for X Window applications and uses the all-graphics approach. Because PreVue-X records testing routines from just the graphical input, such as the pixel-by-pixel mouse movements, and reruns the test suites as such, it does not require any special modifications to the user's application or X libraries. It operates independently from the graphical user interface (Open Look, Motif, CDE, etc.), toolkits, and network [11]. However, there is a disadvantage to this all-graphics approach. Because it runs tests just as if a robot, sitting

in front of the computer, is clicking the mouse at the same locations, over and over, modifications to the GUI looks may be difficult to adjust or simply require the user to record the whole test routine again. For example, as shown in Figure 3, the shrinking of a button or switching two buttons must require user to either modify the original test script—this can be a tedious task, for all events are recorded in pixels—or re-record the test. Also because every graphical input including cursor movements is recorded, this approach slows the execution considerably.

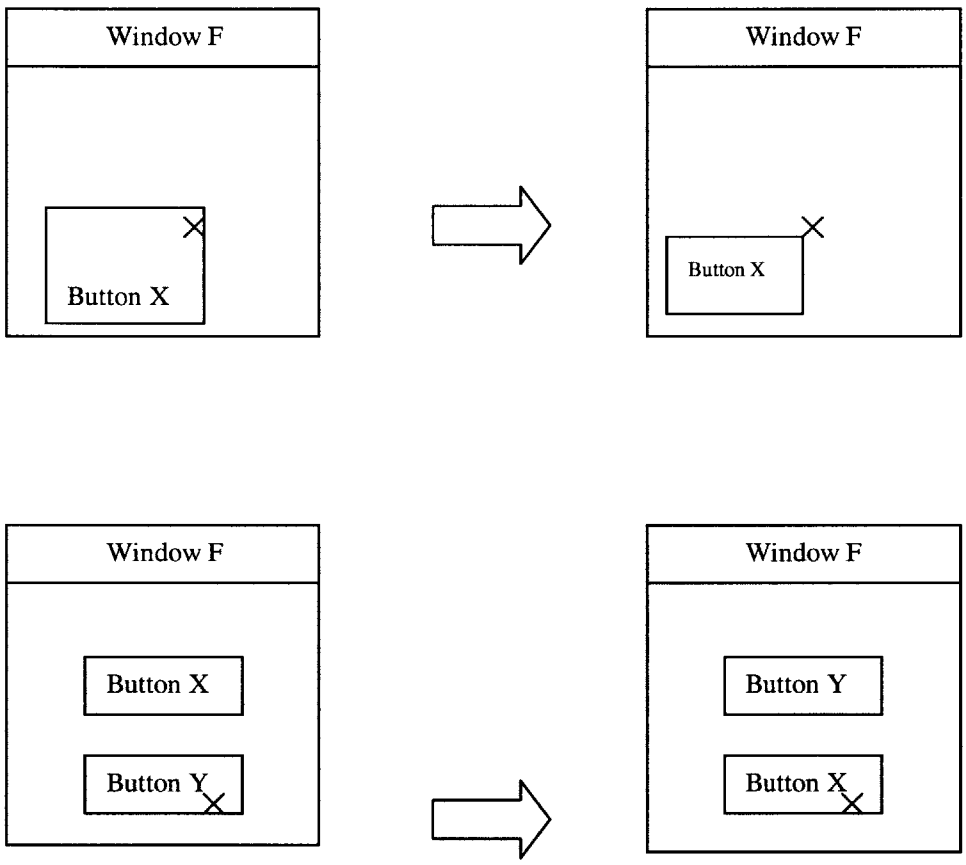

Figure 3. Robustness of the all-graphics approach

The other idea is to modularize GUI components into the specific widgets, which are the basic building blocks of all GUIs. X Window applications are developed using widgets. Buttons, menus, dialog boxes and scroll bars are all widgets. Because all events are recorded and replayed in widgets, the widget-based approach used in XRunner and QC/Replay provides more flexibility to modifications than the all-graphics-based applications [10, 12]. For example, switching Button X and Button Y still works because it records and replays the test by widgets, Buttons X and Button Y in this case. However, the performance issue still lingers. These products establish connection to the application by creating a hook at the X Toolkit library (XtLib) level. Working with XtLib, the application directly captures widget events and monitors the status of every widget in the application, including all the cursor/mouse movements. Recording and replaying every mouse movement on widgets can be very time consuming.

The third party products do not fulfill the role of an efficient automated GUI tester with their current methods. There needs to be a new approach for automating tests that run with speed, reliability, and flexibility.

3. Problem

Currently the GUI systems are only tested manually with high expenditure of manpower and resources. We need to build an automated GUI tester that is robust, maintainable, and high-speed. As mission-critical applications continue to grow in number and scope, change will become more frequent in order to respond to business needs. There is a desperate need for flexibility in maintaining the GUI testing and not requiring manual regression, all the while upholding robustness and performance.

Today's third party products can be categorized into two approaches. First, the all-graphics-based approach tests a GUI by scripting all the graphical changes on the screen, even mouse movements. Second, the widget-based approach is more oriented to the widget level. These approaches are inadequate because they detect and act upon everything, whether important or not (mouse movements, etc.), since they either work on high graphics-level or deep down low in the XtLib level. We need a tester that is smart enough not to record every action taken on the GUI. It must automate only the "important" input, disregarding any meaningless input from the user.

In order to achieve an effective automation, we must modify the internal code of the GUIs in test. This is inevitable because the automated tester must "know" the GUIs—their widget names, supported actions, etc. The new testing software must have access to the GUI structure and always be aware of them even when there are run-time modifications with the GUIs in maintenance. This requires internal code modifications, and the task is to make those modifications absolutely minimal.

The automated testing should result in complete and effective tests, reducing testing costs and testing times due to test standardization and automation. There should be lower guarantee and warranty costs because testing is more thorough and effective, thus leading to lower residual error rates. Many benefits will occur by automating GUI testing and ultimately it will bring lower total development/maintenance costs and shorter time-to-market.

4. Design

In creating a new software structure that would best support automation of GUI testing, we have defined a set of design requirements. The structure of Java Swing is discussed and employed, and the Swing Accessibility feature is studied and applied to establish a modular, proficient and effective design. In this section, benefits of utilizing Swing and Accessibility are explained and the overall design and its advantages as a whole are also described.

4.1. Design Criteria

There are essential criteria for a successful automated GUI tester. Below is the list of requirements of a robust, maintainable, and high-speed automated GUI tester that we desire.

4.1.1. Scripting

In order to support the record and playback feature, it should record business processes into readable scripts, which can later be replayed for verification or reused to emulate thousands of virtual users exercising the system. Using scripts also gives much flexibility in maintenance to modify test routines when necessary. Creating scripts should be easy, such as a simple point-and-click on the GUI, so that even users with limited technical backgrounds can be productive immediately.

4.1.2. Simple Scripting Language

The scripting language must be easily readable for quality assurance programmers to maintain robust test routines. The simplicity of the language also minimizes training,

reducing the cost. A simple script language also allows users to create tests not only graphically but also by allowing users to modify the script.

4.1.3. Repeatability

Building powerful test scripts is only a part of the testing process. We must be able to execute scripts to run tests repeatedly. The replay feature is what distinguishes automated GUI testing from manual testing.

4.1.4. Flexibility in Maintenance

As applications evolve, tests must evolve as well. We must make sure that the scripts created earlier can be reused even when there are modifications to the test procedures—recreating scripts by running the test routines over whenever there is a change can be extremely dreary and exhausting, especially for long tests. Support for script enhancement as the application is developed or updated should be available.

The automated GUI tester should adapt to the following changes:

4.1.4.1. GUI Looks

The all-graphics-based approach tends to be error-prone when there are changes in GUI looks. This approach records and executes events by the pixel-by-pixel approach. The problem with this approach is that when you store events by their locations, pixel-by-pixel, changes such as decreasing size of buttons, shifting graphs, or closing of windows can cause test failures. The tester, therefore, must adjust to changes in GUI looks. This can be accomplished by operating in terms of the widgets, not by the locations of mouse clicks. For example, instead of recording “100 X 130”, the script should record "X button clicked" or "Open file."

4.1.4.2. Texts

Changes in GUI text fields should be processed correctly. Modifying the text field values in scripts is an easy way to accomplish this, however, it should also record when users change the values of text widgets in their GUIs.

4.1.4.3. Graphs

The tester must properly treat changes in graphs such as shifting, resizing, coloring, etc. It must record and rerun with correct positions, sizes, colors, etc.

4.1.4.4. New Widgets

As applications grow and more features appear on GUIs, the automated tester must manage new widgets that appear. This is not a minor modification both to the GUI looks and script rules because it normally would mean new functions. Appropriate changes to parse and unparse script rules should be developed.

4.1.5. Robustness

The automated tester should be robust—it is, after all, an automated GUI *testing* application. Some areas of focus for robustness as we develop an automated tester are test routine modifications, script algorithm, and application synchronization (read below).

4.1.6. Performance

The tester should not record every cursor movement, adjust to every mouse movement, or act upon any other such meaningless events. It should work with just the function calls, making the script much shorter and more readable and therefore making the whole testing process much faster. Also when a user makes changes to a test suite, it

must not require much time to run the new test again, i.e. by recompiling the code. This is also an issue of flexibility and script simplicity.

4.1.7. Broad test coverage

The tester should cover as much of the application as possible. It must test the model the GUI-end component connects to. However, it can skip the highest-level GUI component, such as actual graphics, if it causes significant performance decline. For example, we may skip assimilation of actual mouse clicks and such.

4.1.8. Free of synchronization problems

The root cause of most errors in playbacks is scripts failing due to changes in system response time, such as GUI objects taking longer to appear [12]. Test scripts should continue to run even as systems slow down, speed up, or timing changes.

4.1.9. Hardware efficiency

It should run on the same platform as the application under test. No additional hardware should be required.

4.1.10. Ease of use

Creating tests should be very natural. To create a test, you should simply record a script of your testing process. There should not be a grueling process of describing objects or widgets of your GUI. Replaying and modifying tests should also be easy—this depends on the simplicity of the script language.

4.1.11. Combining test suites

We should be able to combine existing test suites or subtract a test suite from the combination. This makes the regression and integration of smaller modular tests possible.

4.1.12. Thorough test results

It should analyze the results and report what errors were found in the tests and where they were located. A debugging feature is ideal.

4.1.13. Supported platforms

It must run on Windows and Unix environments. As today's applications interoperate on different platforms, the automated tester must also run on and support those platforms.

4.2. Component-based architecture

As the software world keeps on expanding, the need for well-organized software structures has been ever increasing, and for this reason, many software developers of today have turned their eyes to component-based systems.

Component architecture is a notion in object-oriented programming where the "components" of a program are completely generic. Instead of having a specialized set of methods and fields, they have generic methods through which the component can advertise the functionality it supports to the system into which it is loaded. This enables completely dynamic loading of objects. JavaBeans is an example of a component architecture [13].

Some of the main attributes of these systems are the reusability of code, simplicity in structure, and flexibility in maintenance. By using components, we can also develop faster and more efficient applications, including automated graphical user interface (GUI) testing. For this reason, we develop the application in Java, which is a completely component-based (class-based) language, using Swing tools to develop GUIs.

4.3. What is Swing?

Swing is a new GUI component kit that simplifies the development of windowing components such as menus, tool bars, and dialogs [3]. The Swing component set is part of a new class library called the Java Foundation Classes [16], or JFC, and gives Java application programmers outstanding tools for building professional and customizable cross-platform GUIs. Swing contains over 250 classes, representing a mix of components and support classes, and provides more than 40 components—four times the number of components provided by the AWT (Abstract Windowing Toolkit), which used to be the standard package provided with the Java programming language. AWT components are still supported and continue to work but Swing is so much more powerful and elegant than the old AWT counterparts that just about all Java-language developers are now migrating to Swing [3].

What is also interesting about Swing is its architecture, which was evolved from the classic *model-view-controller* (MVC) architecture [14]. MVC architecture was a well-known design for GUI objects. It divides each component into three parts: a model, a view, and a controller, as shown in Figure 4. In the classic MVC architecture, the *model* manages whatever data or values the component uses—such as the minimum,

maximum, and current values used by a scroll bar. The *view* manages the way the component is displayed. Finally, the *controller* determines what happens when the user interacts with the component—for example, what occurs when the user clicks a button [3].

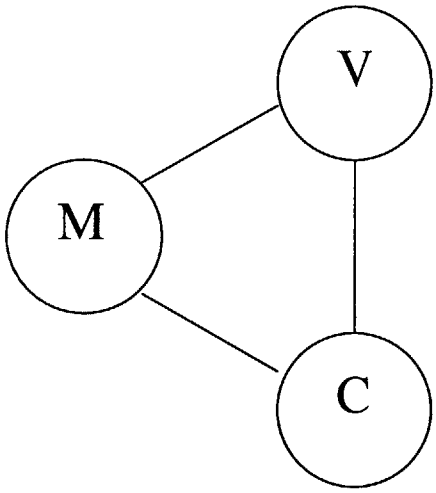

Figure 4. MVC architecture

However, the *view* and the *controller* parts of component require a close relationship that is difficult to separate in real world Swing applications. For example, the traditional MVC architecture makes it very hard to create a generic controller that does not know at design time what kind of view will eventually be used to display it. Therefore Swing has developed a modified MVC architecture, called *separable model architecture* (Figure 5). In this design, the *model* part of a component is the same as the MVC architecture, but the *view* and *controller* parts merge into a single *UI object.*

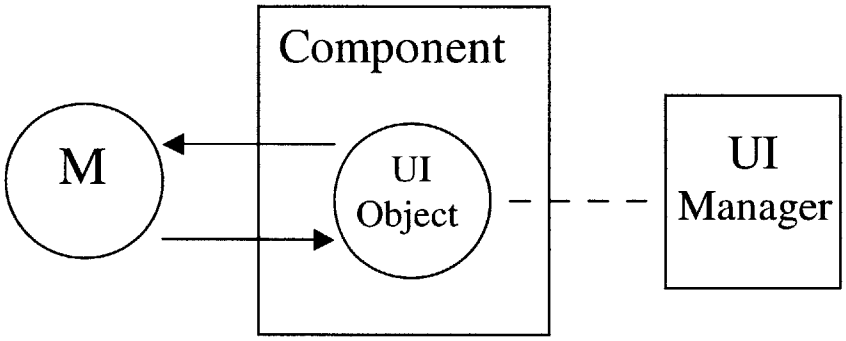

Figure 5. Swing architecture.

Swing has defined a class called *UI Manager* to handle the look-and-feel [16] characteristics of its modified-MVC components. The *UI Manager* always keeps track of the current look and feel and its defaults, and it controls those capabilities by communicating with the *UI Object.*

4.3.1. Benefits of Swing

Swing provides the users a well-organized GUI structure and well-defined development functionalities. The *separable model architecture* modularizes the user interface components and brings simplicity in architecture and reusability of code. This ultimately leads to efficiency and effectiveness in testing, development, and other procedures in software engineering.

By utilizing Swing, we also benefit from its valuable GUI development toolkit and supported classes. We can create a well-defined and structured GUI component without much difficulty, since a lot of GUI components are already defined and

supported. We may also achieve much flexibility and modifiability—i.e. pluggable look and feel design—in maintenance.

4.4. What is Accessibility?

One important feature of the Java Foundation Classes (JFC) is their support for *accessibility* [4]. This allows Assistive Technology to "access" hardware and software and it provides usability to people with disabilities, such as blindness, deafness, or dexterity-related disabilities such as the inability to operate a mouse. Accessible applications can also be useful to people who are not disabled (Figure 6). For instance, the touch-screen programs such as those used in kiosks can be easily developed using the Java Accessibility API.

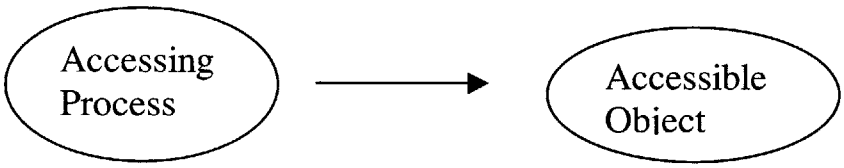

Figure 6. A typical application with Accessibility functionality.

In order to provide even basic access to a Java application, an Assistive Technology needs to be able to "know" or "get" that Java application's characteristics such as the text in a text field and the elements in a list. The Java Accessibility API enables a Java application and an Assistive Technology to communicate this information. Applications that use Swing classes automatically inherit the functionality of the

accessibility API even if they do not specialize the function. Non-Swing components can easily implement Accessibility if they are derived from the JComponent class.

The Java Accessibility API provides support in six basic areas [17]. They include core common information for all UI elements, visual information, text information, selection information, operation information, and information about an object's value. In the first two areas, the API provides a standard mechanism for describing UI components, such as buttons, menus, and trees. Each component may contain information that includes the name and description of the component, location of the component on the screen, the parent and child components, the component's color, its role, its state, etc. In the third area, the Accessibility API provides a mechanism for examining text in detail. This includes parsing a document a letter, word, or sentence at a time, correlating parts of that text with its position on the screen, and getting detailed font and style information about the text [17]. In the fourth area, describing and manipulating selections are supported. Selections involve operations with such things as lists and trees. In the fifth area, querying and performing actions of various UI objects are defined and supported. This mechanism provides an essential functionality for our automating of GUI operations because of the interface to perform actions on a GUI. Some of the supported actions are clicking a button, moving a scroll bar, and opening or closing a tree node. At last, in the last area, a standard mechanism for determining the value of UI objects, such as a scroll bar, is defined by the API.

4.4.1.Benefits of Accessibility

The greatest benefit of Accessibility API for this research is, as mentioned briefly above, having the ability to define and support GUI actions to be "accessible" to other

processes. Accessibility provides a well-defined interface between an "accessible" GUI system of a program and an "accessing" foreign process. Using the idea and support for a third-party process, which are usually voice-recognition-type of support for people with disabilities, to access a server program, we can apply the Accessibility API to our automating GUI testing/operations design. A testing procedure of an *accessible* GUI process can be recorded and an *accessing* process can replay the recorded testing procedure on the *accessible* GUI process.

4.5. New Design

In addition to the Swing's *separable model architecture*, the automating GUI testing system needs a component that records the testing procedures—we call it a Scripting Object. As Swing has done in its architecture, separating the *model* from the UI component makes it possible to capture commands being communicated to and from the model. Using this idea, scripting can be implemented by listening to the communications between the *model* and UI object. There are a couple of ways to accomplish this. First is to have the scripting object in between the UI component and the model (Figure 7). The scripting object receives all the necessary communication between the UI component and the model, however, it may be an unnecessary layer between the UI component and the model. Especially when the operations do not involve scripting, it may only delay the communication and reduce performance.

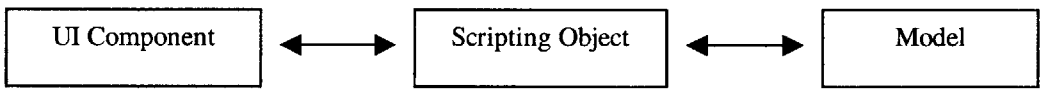

Figure 7. Linear design of having Scripting Object between UI Component and Model.

Second, we can connect the Scripting Object to the model (Figure 8). It will allow scripting of all the changes in the model. This, however, is not preferable, for it is possible for one model to be associated with multiple UI components. This means that when those UI components operate automated GUI testing, the model must manage all the Scripting Objects of the UI components. It is expected that the model may run into database management/network problems and slower performance. In addition, this design is undesirable due to the limited test coverage—Scripting Object only records actions on the model and not the UI component, therefore, ultimately testing just the model.

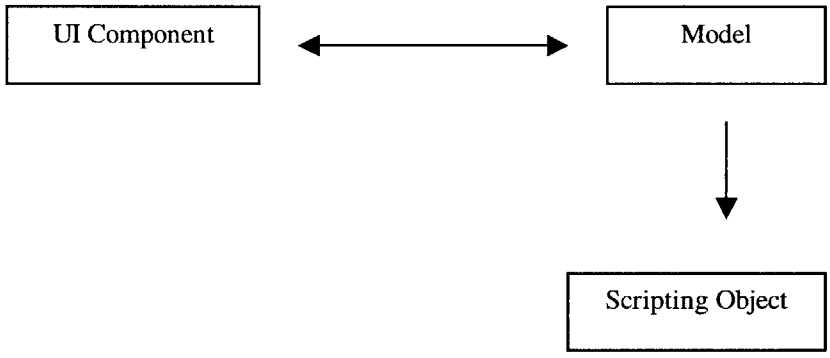

Figure 8. Design of Scripting Object communicating with Model.

In our design, we have the UI component communication with the Scripting Object (Figure 9). This avoids the problem of having multiple Scripting Objects connected to one process—Model. It also increases the test coverage by running the testing from the script *through* the UI component.

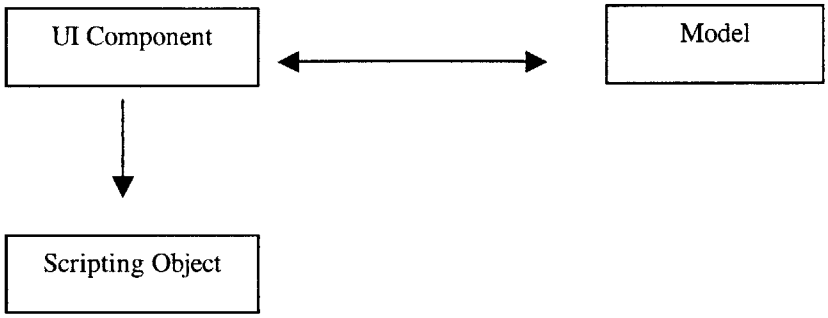

Figure 9. Design of Scripting Object communicating with UI Component.

In our complete software design of automating GUI testing, we modularize the system into the following components: a scripting engine that connects to the UI object, a UI object, a model, and an "accessing" process. This architecture uses the Swing UI Component, however, some customized code is necessary to make our own, *customized UI object*, to add the scripting and accessing components to the architecture. We achieve this by using the Accessibility API, which offers a very useful interface that can be used to efficiently combine any customized code on top of the Swing UI object. Following is our overall structural design of automated GUI testing (Figure 10).

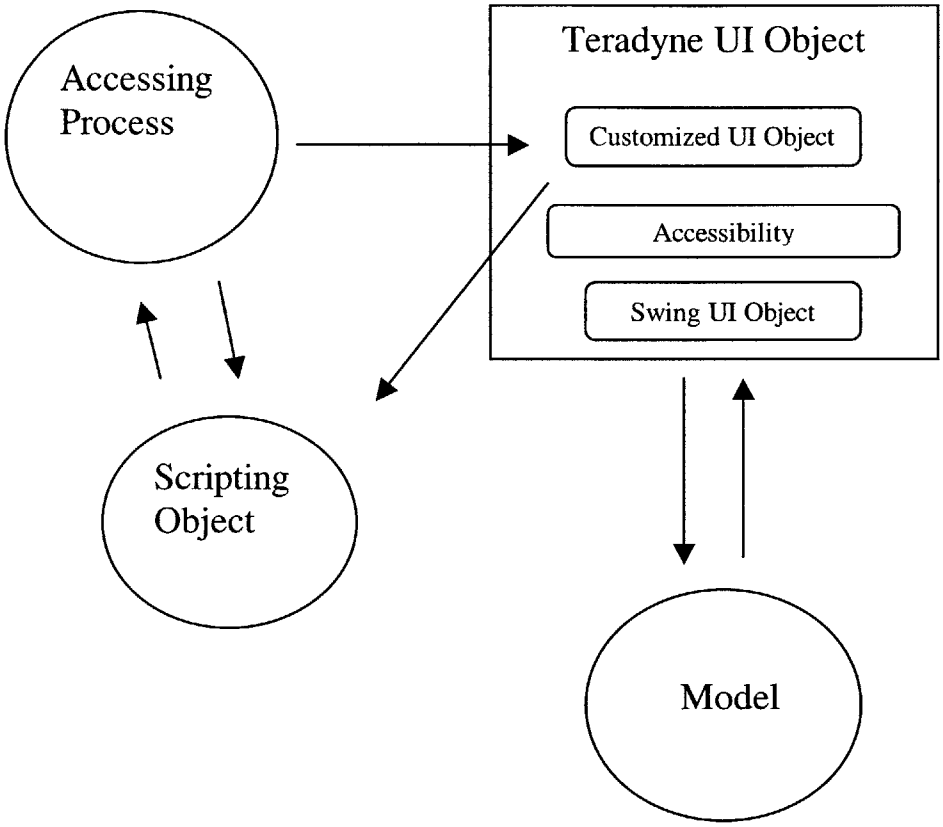

Figure 10. Customized GUI component architecture.

A typical routine of this system usage would start with a user input by a mouse click or keyboard entry. The UI object then fires two actions: triggering the scripting object to record the input and calling the appropriate function in the model. The model returns a result to the UI object. A number of such procedures may occur, incrementing the size of the script that the scripting object maintains. When a user requests a replay of the script, the accessing process contacts the scripting object and receives the script with the previous user inputs on the UI object. The accessing process unparses the script and calls the UI object to rerun the operations as written in the script.

4.5.1.Benefits of the Design as a Whole

In our design, there are numerous benefits. By using Swing GUI tools and components, we achieve well-defined UI objects such buttons and tables, and we benefit from the reusability and flexibility of the architecture. The Accessibility API provides a well-defined connectivity between an "accessible" UI component and a foreign "accessing" process.

Additionally, we create a very concise script and a high-speed testing process. The separation of the UI component and the model allows us to capture and record actual commands to the model, instead of every graphical change in the window. This enables recording one script line per UI input, making the script very readable and easily modifiable. Only the "important" and "necessary" steps are scripted to ensure the high performance of recording and rerunning the testing procedures.

5. Demonstration

A critical part of building and presenting a new design is successfully demonstrating that the proposed architecture correctly implements the desired features. In this section, an application is written and described to demonstrate that a GUI testing process can be correctly rerun automatically. A tentative and simple script language is developed and discussed and a sample regression test is presented. Finally, the demonstration application is evaluated against the criteria described before.

The demonstration application needs to be carefully chosen: it should contain essential features that show sufficient automated GUI testing. One good way to accomplish this is to work with a real application. However, since all the Teradyne GUI

tools are unnecessarily complex for demonstration purposes and are written in C++, and since Teradyne's only Java GUI tool is only in its development stage, working with a Teradyne GUI tool is unfeasible. We are faced with the necessity of creating a whole new application. This is disadvantageous for two reasons: one, it requires a lot of time, and two, we can be biased in our development to create an application that would be more fit for our testing.

However, we have another Java Swing GUI tool that has been written for personal use but contains GUI components similar to those that Teradyne tools use such as text fields, buttons, tables, menu bars, etc. Demonstrating the automated GUI testing with this application is preferable to creating a new demonstration. Some of the benefits of working with the previously developed application are the efficient use of time while accomplishing the same goal and, more importantly, the fact that we can assimilate the whole process of applying automated GUI testing to existing real-time GUI tools. This approach is helpful in detecting some issues that arise, such as how to modify the least amount of original code.

5.1. LoanSave tool

The existing Java GUI tool is called LoanSave (Figure 11). LoanSave is a simple interest calculator. It may be used to calculate parameters of a simple interest loan or a simple interest savings plan [18].

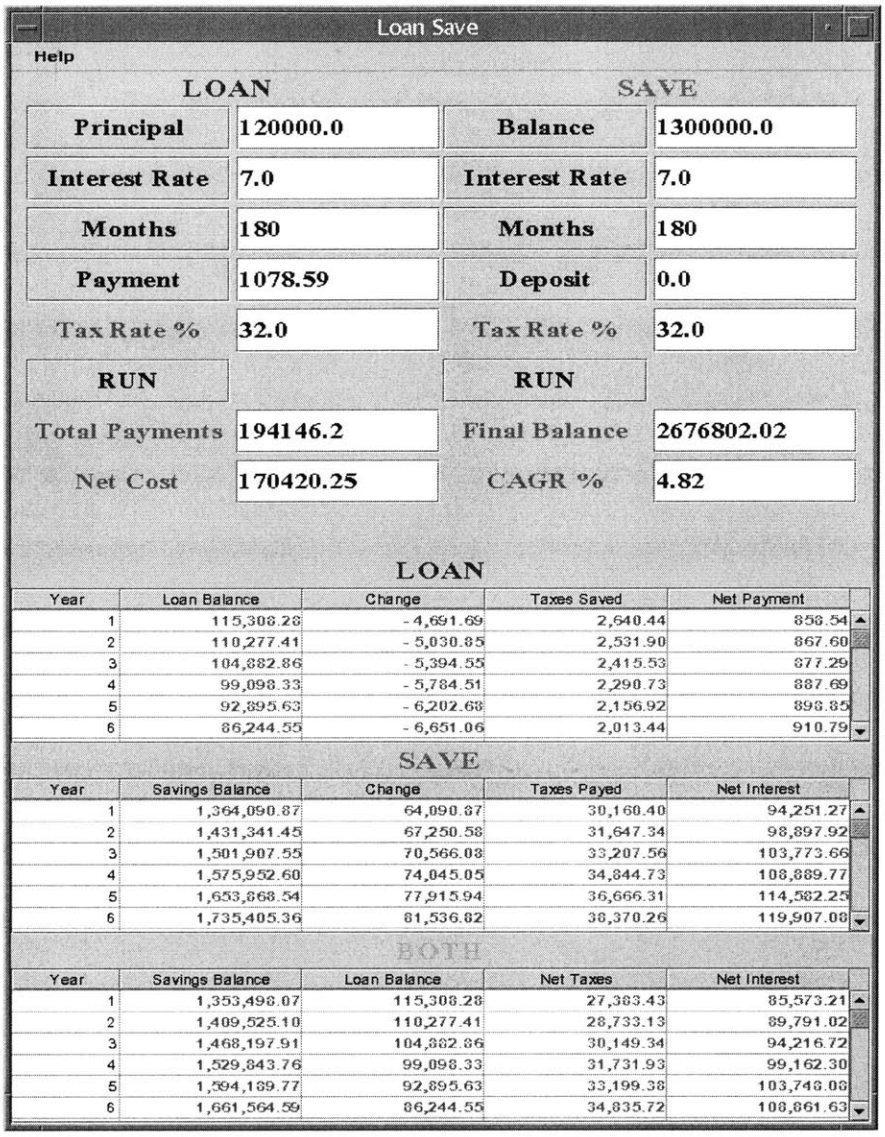

Figure 11. LoanSave application.

There are two types of entries on the LoanSave GUI tool: text fields and buttons. The text fields are used for both data entry and display of calculation. Principal, Balance, Interest Rate, Months Payment, and Deposit text fields are used for both purposes. However, Tax Rate fields are data-entry only, and the Total Payments and Final Balance fields are display only. The Final Balance and CAGR fields are used both to display values resulting from pressing Run and to enter target values for Save text field values,

but they do not possess corresponding buttons. In all cases, only numbers are entered; LoanSave does not operate with other characters. All field values may be floating point and integer values except the Months fields, which must be integers.

LoanSave's buttons are used to cause the calculator to compute something. The Run buttons cause the corresponding Loan or Save panel to execute their actions and enter the resultant data into the tables. All the other buttons are used to update values for their corresponding parameters with new and correct values that match with other parameters. For instance, changing the value of the loan principle by typing it in the Principal text field and by pressing the Run button, the Loan table and the Both table are updated. A user can also update the text field value by pressing its corresponding button. In another words, after changing the Principal value, simply by clicking the Payment button, the Payment value will be updated, corresponding to the values of Principal, Interest Rate, and Months. It is overwritten with new information, and the previous value is lost.

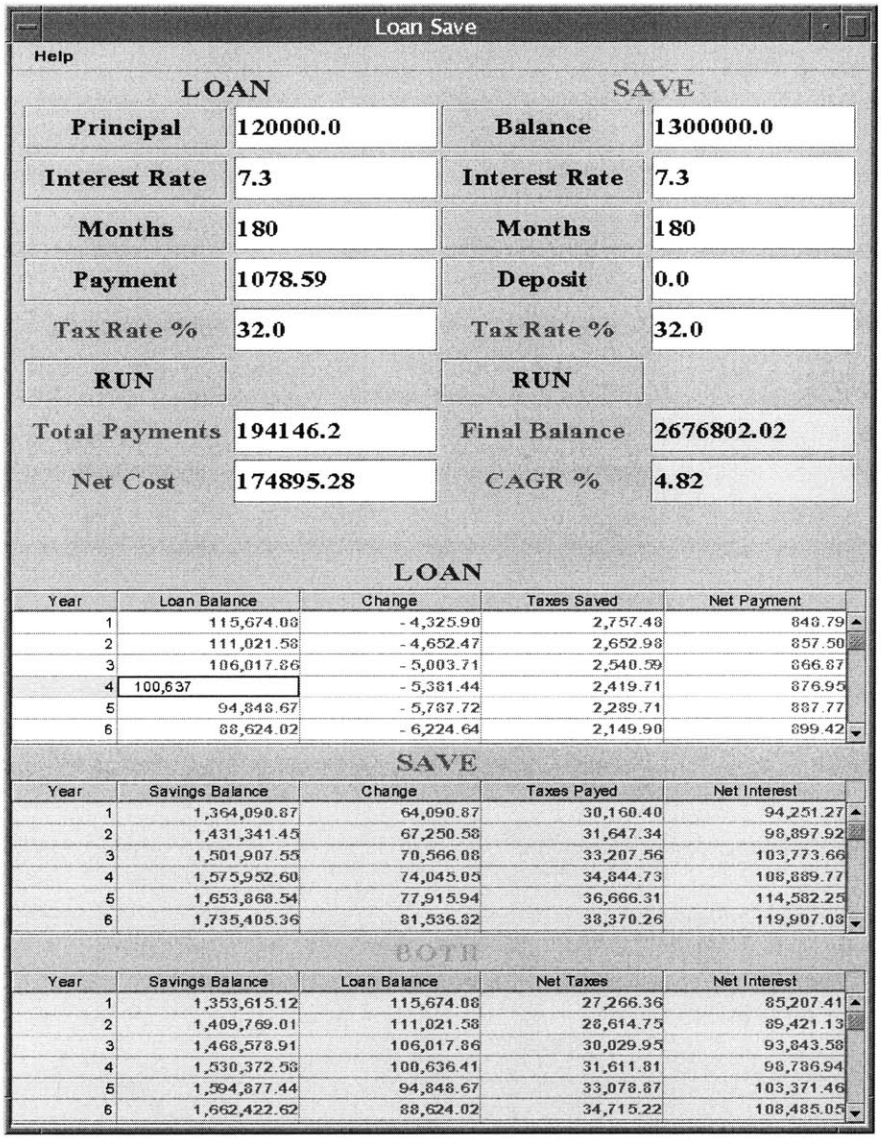

Figure 12. State of changing a Loan table cell value.

LoanSave writes loan and savings annual summary data into three tables: Loan, Save, and Both. Pressing the Run button initiates the operation to update the tables. Whenever there is a change in a text field, or a button other than the Run button is pressed, the background color of the tables darkens from white to gray (Figure 12). This indicates that data in the tables may not correspond to the values of the parameters in the control panel. The tables are the most interesting GUI component from Teradyne's

perspective. Due to management and operations of enormous amount of data, many of Teradyne GUI tools include tables. Scripting and replaying table operations can be very helpful in reducing the number of regression tests to be done manually.

5.1.1. Loan Calculator

There are four important parameters in calculating interest in loan: principal, interest rate, period, and payment. Principal is the amount borrowed, interest rate is just a simple interest rate and not APR, period is the time duration (number of months in LoanSave) over which the loan is to be paid back, and payment is the monthly payment amount. Each parameter value is dependent upon the other three values, therefore, given any three values, there is only one possible value for the last parameter. This allows the user of LoanSave to enter any three parameters of a loan in the control panel and calculate the fourth. For instance, by entering principal, interest rate, and period values in the text fields, a user can obtain the monthly payment value from the GUI.

Total cost is the total amount paid for the loan, including all the interest. Net cost is the total cost minus the amount of money saved from tax deduction.

5.1.2. Savings Calculator

There are four important parameters in calculating interest in savings: balance, interest rate, period, and deposit. Balance is the beginning amount of money in the savings plan, interest rate is just a simple interest rate and not APR, period is the time duration (number of months in LoanSave) over which savings plan is to run, and deposit is the monthly deposit amount. Each parameter value is dependent upon the other three values as was in the loan calculator. A user can enter any three parameters of a savings

plan in the save panel of LoanSave and obtain the fourth parameter value. For example, entering balance, interest rate, and period can calculate the required monthly deposit.

Also when the Run button is pressed, the savings calculator writes the Final Balance and Compound Annual Growth Rate (CAGR) to their respective fields. Final Balance is the result of running the current savings plan. The CAGR is the effective growth rate of the savings balance and includes the effects of monthly deposits and taxes paid on the final balance. These final balance or CAGR values can also be manually entered, and be used as a fixed "target" value to determine Balance, Interest Rate, Months, or Deposit value.

5.1.3. Help Tutorial

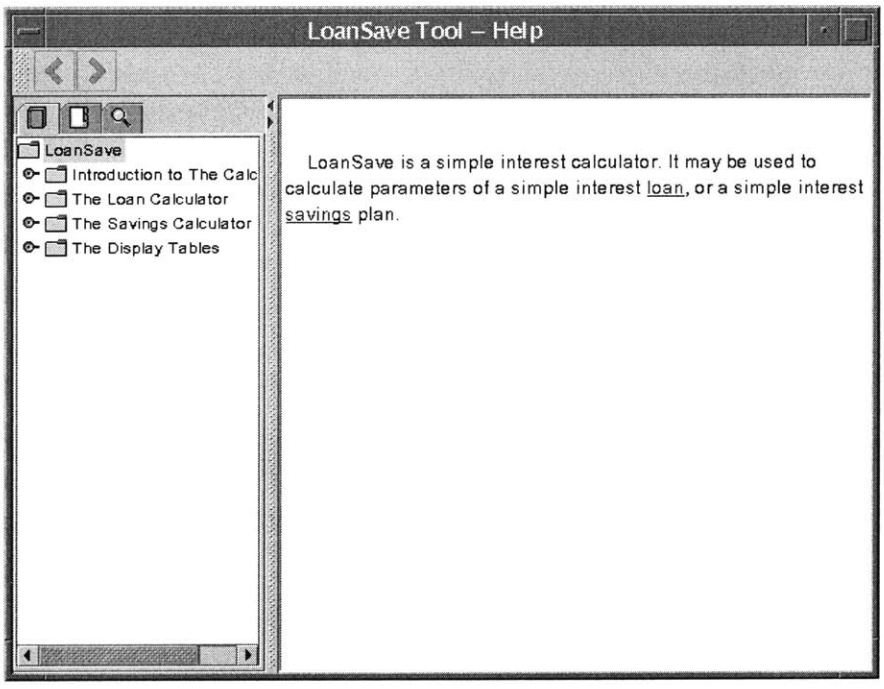

Figure 13. LoanSave Tool Help, Tutorial.

There is a menu bar in LoanSave application. It contains just one menu, Help, and Help menu contains just one menu item, Tutorial. Choosing the Tutorial menu item will pop up a window (Figure 13) with descriptions of different elements of LoanSave application.

5.2. Script design

In automating any sort of operations, there needs to be a means to record and retrieve those procedures that have taken place. Furthermore, because the to-be-replayed procedures must be readable and modifiable to the users, scripting is an inevitable and critical element in automating GUI operations.

There are numerous methods of communication between two different processes, which are an accessible process and an accessing process in this case. We need these processes to communicate the recorded procedures—the accessible process records procedures and the accessing process reads it. This simple one-sided messaging can be accomplished by setting an interprocess communication (IPC) protocol such as sockets or by using shared memory. However, since we desire flexibility in maintenance where users may modify scripts, the most effective method of messaging is file sharing. The accessible process would record its GUI operations onto a file and the accessing process would read the file and replay those operations on the accessible process. This allows replaying scripts with adjustments such as text field value changes, additional button presses, and table column deletions. A user may simply make text changes in the script file.

In LoanSave application, here is a list of GUI components that need their operations to be scripted:

- Tutorial menu item in the Help menu bar,
- Principal, Interest Rate, Months, Payment, and Run buttons in Loan control panel,
- Balance, Interest Rate, Months, Deposit, and Run buttons in Save control panel,
- Principal, Interest Rate, Months, Payment, Tax Rate, Total Payment, and Net Cost text fields in Loan control panel,
- Balance, Interest Rate, Months, Deposit, Tax Rate, Final Balance, and CAGR text fields in Save control panel,
- Loan Balance, Change, Taxes Saved, and Net Payment column cells in Loan table,
- Savings Balance, Change, Taxes Paid, and Net Interest column cells in Save table.

5.2.1. GUI component organization

There is an "address" associated with every Java Swing component in a window. In order to "get" a component and its accessible information and features, this component address is crucial. When an assistive technology first "grabs" an accessible application, it grabs the top-level component, which is the window. In order to access each of the components in the application, the accessing process needs to traverse down to the component level. Sun Microsystems has implemented and provided a very useful application (Figure 14) as one of its assistive technology examples. The application is called Monkey, and it runs assistively with an application, gets the application's accessible information, and displays the hierarchy of the GUI components in a tree format. Like a monkey, Monkey "swings" through the component trees in a particular

JavaTM Virtual Machine and presents the hierarchy of the component organization of the accessible window/application [19].

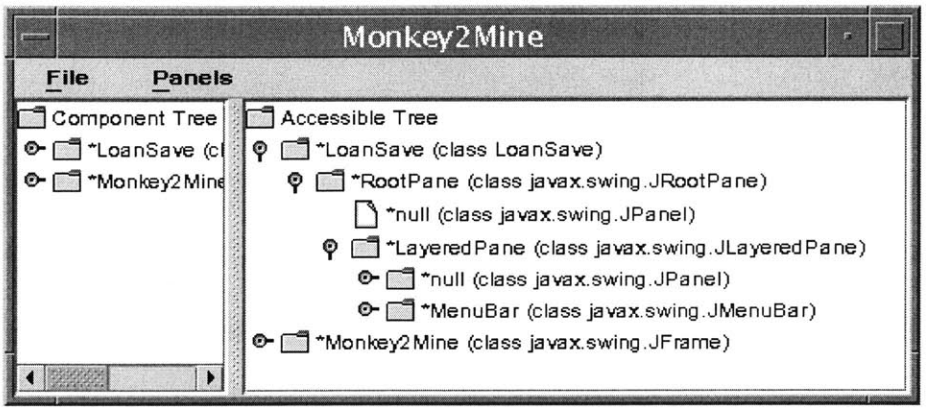

Figure 14. Java's Accessibility Monkey Example Application runs with LoanSave.

As shown above, the LoanSave application has a few layers of components before the first visible component of the GUI application appears in the hierarchy. Nearly every Swing component ultimately resides in an instance of JRootPane because all of Swing's top-level containers contain an instance of JRootPane [16]. Underneath it is an instance of JLayeredPane, which allows multiple layers to hold components. The layered pane contains an optional menu bar—LoanSave contains one—and a content pane1[1]. The content pane is where applets and applications set up its main components, as shown in Figure 15.

[1] I have set accessible names of all the components but could not figure out how to do it with JPanel class components—naming components is discussed more in Implementation section. In Figure 14, the content pane does not have a name—therefore, a null value.

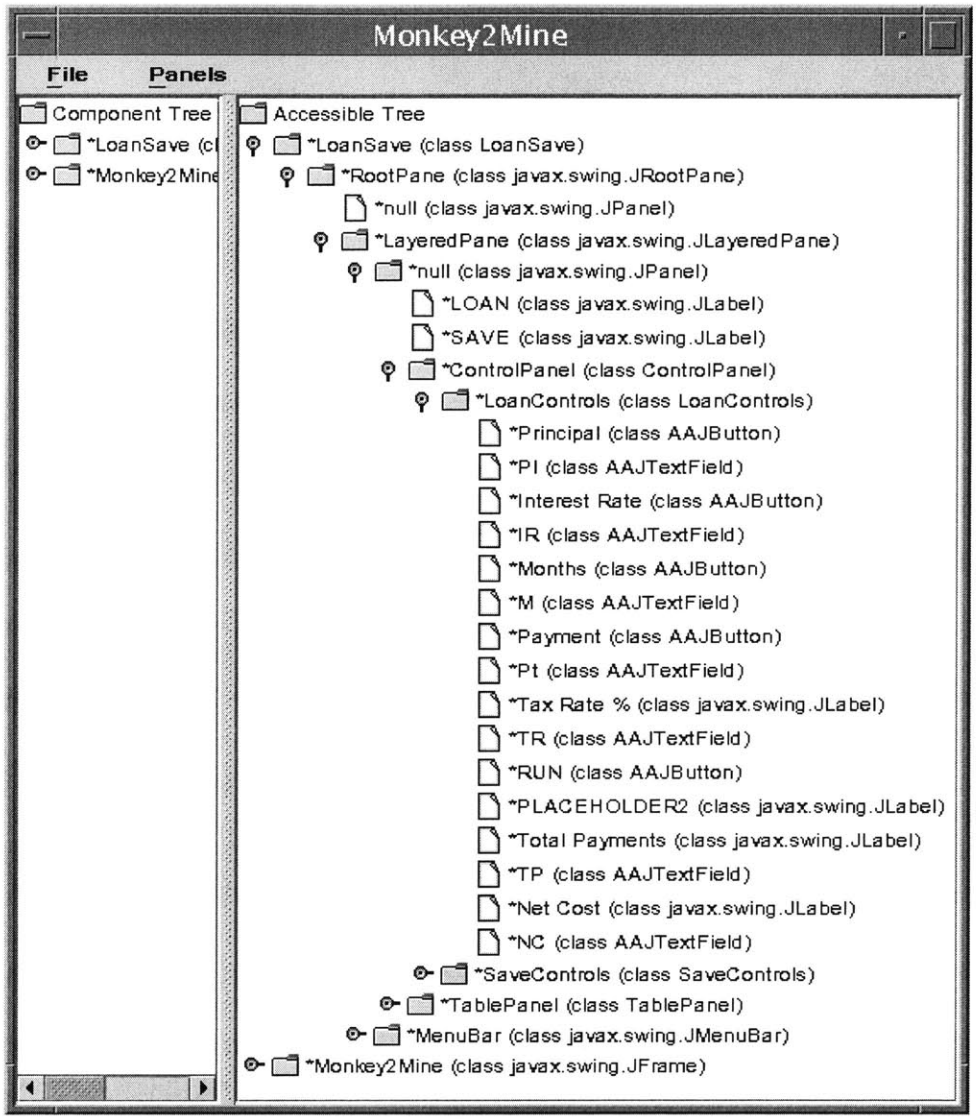

Figure 15. Hierarchy of components under the content pane.

Underneath the content pane, there are four components: "LOAN" and "SAVE" simple JLabel components and ControlPanel and TablePanel. Referring back to Figure 11 of LoanSave application, ControlPanel contains all the elements of the upper half of the window, and TablePanel does the lower. ControlPanel includes LoanControls and SaveControls, which, in turn, contain all the buttons and test fields as shown in Figure 15. To summarize, IR (Interest Rate) text field component resides under LoanSave

application, RootPane, LayeredPane, null (JPanel/content pane), ControlPanel, and LoanControls.

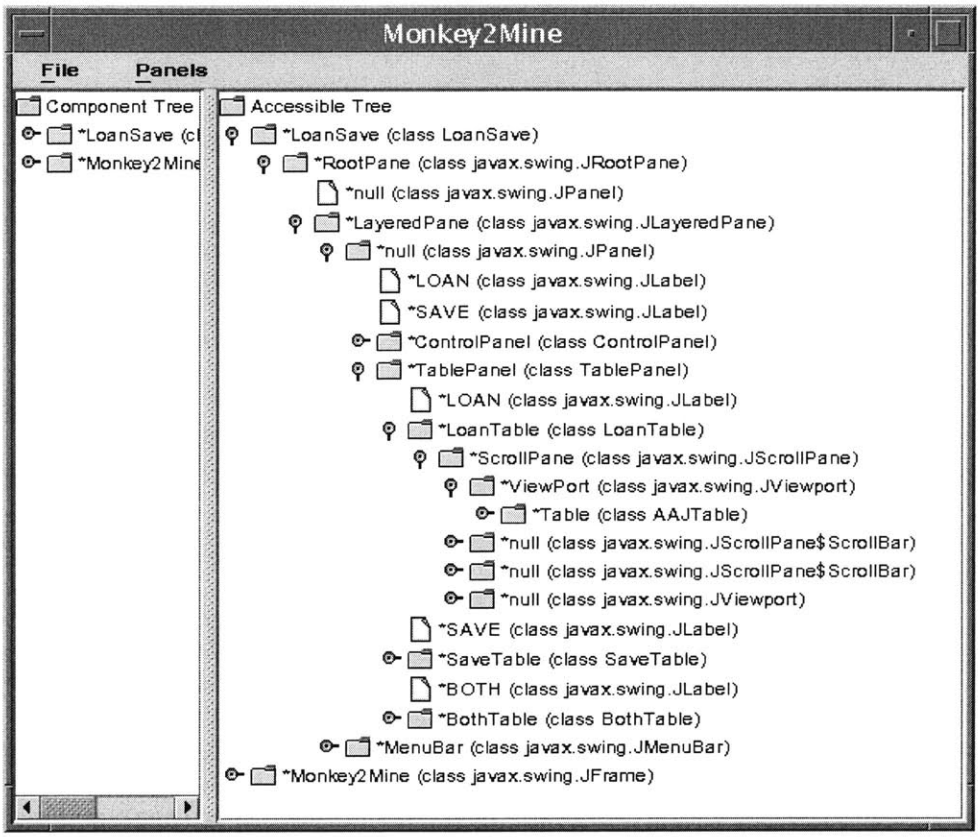

Figure 16. Hierarchy structure of TablePanel.

Let us look into the structure of the table components (Figure 16). Under TablePanel, there are three sets of labels and customized table classes: Loan, Save, and Both. Under each of those table classes is a scroll pane, which contains a view port and scroll bars. The JTable component we desire resides under the first view port. JTable component contains a model that implements all the cells, however, these cells are not accessible components. Nevertheless, because JTable component is capable of controlling/modifying its cells, and because actions on a JTable component, such as

column addition, need to be recorded and replayed as well, accessing the table component is sufficient.

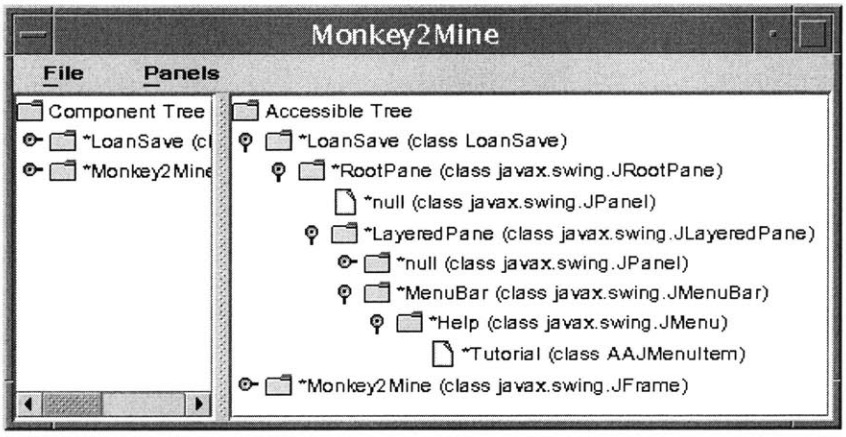

Figure 17. Menu bar component hierarchy.

The Menu bar hierarchy of LoanSave application is fairly simple since there is only one menu item called Tutorial under Help menu, which is underneath MenuBar JMenuBar class.

5.2.2. Script Language

There are three elements that are recorded when an action is recorded: component name (or the full address), action number, and argument. The Component name is necessary because GUI testing/operation involves multiple components. A button may be pressed, a text field value changed, and a table cell value changed. The accessing process must know which component to replay the action on. The component name must also include the full address so that the accessing process may easily and correctly find the component. Without a full address, two components with a same name but underneath different panels may result in an undesired action performed on the other

component. An action number is needed because a component may support more than just one action. Although a lot of components such as buttons support one action, Button_Pressed, components like tables have numerous actions provided. Table action events are fired when cell values are changed, table structure is changed, rows are inserted/deleted/updated, columns are added/removed/moved, column margin is changed, column selection is changed, or row selection is changed. The action number in the script is zero-based. The third element is the argument value. When actions occur, many times they involve values. For instance, when text field is changed, the action must be recorded with the new value.

We will also add a fourth element in the script: comment. This is solely for the users who may read or modify the script. The comment will make it easy to understand each script line and what it means in terms of component actions.

A script format is as follows:

<top component>.<middle components>.<action component>:<action number>:<argument>//<comment>*

Each component is separated with a dot (.) which is a separator of components that is commonly used in the software world. This produces a script that is commonly and easily understood. Colons (:) separate the action number and argument. The comment is following double slashes (//) which are, once again, a general practice of writing comments in programming languages. A sample script of simple procedure of changing the loan interest rate and the savings number of months and pressing both loan and save Run buttons would be:

LoanSave.RootPane.LayeredPane.null.ControlPanel.LoanControls.IR:0:7.5//JTextField Value Changed
LoanSave.RootPane.LayeredPane.null.ControlPanel.SaveControls.M:0:150//JTextField Value Changed
LoanSave.RootPane.LayeredPane.null.ControlPanel.LoanControls.RUN:0://JButton Pressed
LoanSave.RootPane.LayeredPane.null.ControlPanel.SaveControls.RUN:0://JButton Pressed

The above is the actual content of a script file. Of course, the script is recorded automatically when actions occurred in the LoanSave window.

There is a possibility of component names or other elements containing dots, colons, or double slashes. This issue is written more in depth in Future Work section. However, in the mean while, in order to avoid any confusion and error, we set a requirement that all component accessible names to not include any of the separator characters.

5.2.3. JTextField and JTable scripting issues

As mentioned earlier, most of these components provide one action, i.e. button components only need to script when they are pressed and menu item components only need to script when they are selected. However, scripting actions of JTextField and JTable components are done with different approaches.

The original implementation of the text fields in the control panel includes DocumentEventListener. This listener fires an event every time there is any kind of keyboard input. We must not record script listening to this event due to incorrect replays. For instance, a user performs a sequence of actions such as erase, erase, write 1, write 0, and write 0 in a text field with a starting value of 20. These actions will change the value of the text field to 100. However, replaying this same sequence when the initial text field value is different will result in a wrong outcome. For example, if the previous value is 200, the same erase-erase-write1-write0-write0 sequence of actions will change the value

to 2100, which is 2000 more than 100! This scenario is very possible when/if the set of test operations is replayed with different initial values (read Issues section for more on initialization).

Scripting the text field values when the RETURN key is pressed also would result in an incorrect outcome. Because the text fields of LoanSave control panel are implemented with DocumentEventListener, the RETURN key does not need to be hit for the application to run with the new value—remember DocumentEventListener listens for every keyboard input.

Therefore, the text fields record script when it loses focus. This is the correct approach because, in order to run the calculation, user always needs to press a button that takes the focus off of the text field. JTextField focusLost() action is used.

Implementing tables also required a little different approach. Because there are numerous events in JTable class including those of its cells, and because those cells are not accessible, we handle all the events in JTable component level. This involves a minor modification in our script for table actions. When there is an action event in a cell, table component receives the event and records it. However, when it records the script, it must specify which cell value has changed. Remember the component related here is the table.

A simple solution to this problem is to simulate cell component. In other words, we specify the cell in the component accessible name space and let it look as if the cell is what the accessing process is accessing. Below is a sample script involving a cell value change action:

LoanSave.RootPane.LayeredPane.null.TablePanel.SaveTable.ScrollPane.ViewPort.Table.Row=1Col=1:0: 1,376,745.33//Table JTable ValueChanged

Here the cell at row 1 and column 1 has changed to 1,376,745.33. "Row=" and "Col=" is manually inserted and the values are received from the event.

To correctly replay this action, we made a minor adjustment in unparsing to get an accessible handle on the table and information on the modified cell location. This also entails a requirement that component accessible names to not contain substrings of "Row=" or "Col=".

5.3. Implementation

5.3.1. Accessibility interface

The main interface of the accessibility package is the interface Accessible. As mentioned earlier, all Swing components, or any components descended from JComponent, implement Accessible, and we can easily make the class implement Accessible for those that do not. The sole method in the interface Accessible returns an instance of AccessibleContext class [20]. Through the methods of this class we can find out all of the accessible information including that of other subclasses and sub-interfaces. Some of the important information that AccessibleContext class provides are AccessibleName, AccessibleDescription, AccessibleChildrenCount, and AccessibleParent. We can set and get accessible names, descriptions and children count on any accessible component. *getAccessibleParent()* method returns the accessible

component that the current component resides in. This is very useful in figuring out the address of a component.

One of the sub-interfaces includes AccessibleAction. This interface is to be implemented by any object that can perform actions and would like them to be accessible. The interface includes the methods *getAccessibleActionCount()*, *getAccessibleActionDescription()*, and *doAccessibleAction(int index)*. An accessing component can discover the number of accessible actions and their descriptions, and it can perform a desired action by calling *doAccessibleAction(int index)*. A notable fact about the *doAccessibleAction(int index)* method is that the programmer actually needs to assimilate actions. In another words, the programmer must write code for what the function call does, given an integer index. It is very similar to simply calling a function of a remote process.

5.3.2. Coding Design (Requirements)

One of the major goals in implementing automated GUI testing/operation is to minimize the changes in code. If Teradyne has a Java GUI tool, supporting automated GUI testing should not involve much change in its original code.

This is beautifully accomplished by simply using customized classes that support Accessibility instead of plain Swing classes. For example,

JTextField myTextField = new JTextField();

can be simply replaced by

AAJTextField myTextField = new AAJTextField();

where AAJTextField—AA for AccessibleAction—class is a customized class that is basically the same as JTextField class but with an AccessibleAction interface and writing the script implemented in it.

Here is the actual code for AAJTextField class:

```
class AAJTextField extends JTextField{
    AAJTextField thisText;

    AAJTextField(String text, int ColNum){
        super (text, ColNum);
        thisText = this;

        // listen for focusLost event and record script
        this.addFocusListener(new FocusListener() {
            public void focusLost(FocusEvent e) {
                String path;

                // returns a full path from the top-level swing component.
                path = getPath(getAccessibleContext());

                // saveScript()[2] writes to the script file.
                saveScript(path, 0, thisText.getText(), "JTextField Value Changed");
            }
            public void focusGained(FocusEvent e){}
        });
    }

    // getAccessibleContext must return accessibleContext with AccessibleAction implemented.
    public AccessibleContext getAccessibleContext() {
        if (accessibleContext == null) {
            accessibleContext = new AccessibleMyJTextComponent();
        }
        return accessibleContext;
    }


    // This class implements AccessibleAction.
    public class AccessibleMyJTextComponent extends AccessibleJTextComponent
```

[2] In writing and reading script to and from a file, a programmer must not use ObjectReader or ObjectWriter. Although it is convenient in reading a whole file into a string at once, because ObjectWriter writes to a file with a checksum, a user cannot modify the script file and replay it. Also a practice of leaving a script file open for an extended amount of time is undesirable due to a concurrency issues. Both accessing and accessible processes have both read and write permissions to the file.

```
                                        implements AccessibleAction {

            public AccessibleAction getAccessibleAction() {
                    return this;
            }

            public boolean doAccessibleAction(int index){
                    if (index==0){
                            thisText.setText(thisText.getAccessibleContext()
                                            .getAccessibleDescription());
                            return true;
                    } else return false;
            }

            public int getAccessibleActionCount(){return 1;}

            public String getAccessibleActionDescription(int index){
                    return ("Change JTextField Value");}
            }
        }
}
```

The basic format of the new customized classes that implement Accessibility and automated GUI testing is pretty straightforward.

There is one more modification that needs to be made. Every accessible component must define or name its accessible name field. This is tremendously important in finding the right components. AA-classes can automatically set their components' names, however, this can easily lead to a parent component having multiple sub-components that are of a same class and a same name. Hence, we require programmers to name all of their accessible components. This requirement, again, is a very simple and straightforward adjustment from the original program without automated GUI testing support.

One minor compromise in our implementation is the method of communicating an unparsed argument value from a script (read 5.2.2. Script Language). The accessing process replaces whatever is stored in the component accessible description variable with

the argument value. The accessing process reads the script, unparses each element, and, when it calls the appropriate accessible action call on the accessible process, it writes the argument value in string type to the accessible description variable. The accessible process simply runs the action with the appropriate argument from its own accessible description variable. Basically, we are using the accessible description variable, which is predefined to provide description of the accessible action in English, to store and transfer an argument variable for the action. A better defined and more elegant way to communicating argument values should be researched and applied (read more in Future Work section).

A small step to set up for an assistive technology to run is required. In the $JAVAHOME/jre/lib/accessibility.properties file, we need to specify what assistive technology we desire to run with an application. For example, to run AccessLS (Figure 18) we need to write:

assistive_technologies=AccessLS

in accessibility.properties file. This will run AccessLS application whenever a Java application starts. AccessLS application is a simple application with just one button that replays the existing script on the LoanSave application

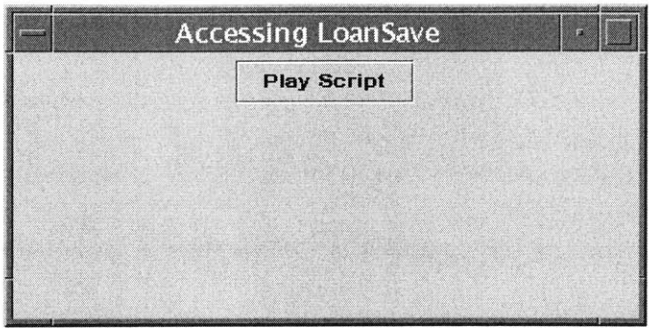

Figure 18. AccessLS application.

5.4. Issues

5.4.1. Initializing

One important aspect of replaying a script of GUI operations is the setting before running the test suite. Because the widget values may be changed during the course of a test procedure, we cannot assure that repeating a same procedure will produce the same result.

For example, let's say that the value of A is 4, and a widget value B is dependent upon A's value of 4 (let's say B calculates A's square root). After B has sourced A's value and gave itself a value of 2, a different operation changes A's value to 9. Now, if the test script is run again, A's initial value is not 4 but 9 and B's value will not be the same 2 but 3. If there are any dependencies among the widgets, rerunning the test procedure will result in different outputs each time. This brings out the question: do we need to save the original setting and initialize the setting before each time we replay the script?

There are pros and cons to the idea of initializing the setting of every replay. Obviously, the advantage of the idea is that we can have consistent tests with same initial

values even if we run it multiple times. No matter how the values of the GUI window change, each replay of the script will run against the same initial values. This will allow users to replay scripts without resetting the initial values.

However, there are a number of disadvantages in resetting the values every time. First of all, it may not even be possible because there are an uncountable number of states we need to record. Not only do we need to record the values of the main window, we need to record all the states of the popup windows (other GUI tools), which is a likely event for the Teradyne tools. Having an enormous number of GUI tools that depend on each other and can pop up during a procedure, recording the setting of ALL the windows that can possibly pop up may be not a worthwhile endeavor. Second, not only can there be unknown number of initial states to save, we may not even want to save the states of these tools, if the tools' values depend on other processes. Let's say we bring up a GUI tool that updates the stock price every minute and does some calculations, we obviously do not want to record the initial states, if we want the updated stock values. Third, if we want to run multiple test suites in a row, where latter suites take the result values of the earlier ones, we do not want to initialize the settings before each test suite. Considering all these drawbacks, it seems like recording the initial values is undesirable.

Another alternative to this issue would be to restart the application whenever we run a test suite. This basically means the settings are always the default/original values of the GUI windows and can eliminate some of the disadvantages and achieve the advantage of recording the settings. This, however, gives the user very little flexibility. Basically the test suite can be only run with the initial values. Also this means the user needs to

restart the GUI tool every time he/she wants to replay a test script. This delay will put this approach at a disadvantage.

We may just need to require the user to make sure the initial values are as desired. For this thesis, because of the uncertainty of how to handle initializations, we do not implement a recording/reset state feature. Also, we record every GUI operation instead of having a button to start and stop recording.

5.4.2. Table implementation

In implementing tables (scripting, more specifically), instead of numbering rows and columns, we may "name" them. For example,

Row=0,Col=2:5'11"

can be

Row=Joe,Col=Height:5'11" .

By doing this, we can make the script easier to modify and read. Also, it yields higher test coverage. Adding, subtracting, or modifying an entry of the table can be done using this scheme. For instance, let's say Joe's entry is moved from row 0 to row 4. Now modifying row 0 will not modify Joe's entry. However, if we name all the rows, "modifying Joe's entry" will always find the correct entry.

There are a couple of issues raised by this approach. What if there is a change in entries, such as Joe's entry being is erased? Or Joe's name is changed? The row- and column-naming scheme will result in an error and will not be robust and flexible.

One possible approach to this problem is to have a version number for each script. Any change in rows or columns can generate a new version. Read Future Work section (7.1) for more. For this thesis, we will just use the numbering scheme.

6. Evaluation

After demonstrating that the new software structure will support automation of GUI testing, we have evaluated the demonstration product, development process and maintenance against the previously defined set of design requirements (read 4.1). Below is a table of the criteria for a successful automated GUI tester and how this new design measures up to the requirements of a robust, maintainable, and high-speed automated GUI tester that we desire.

Scripting	**O**
Simple Scripting Language	**O**
Repeatability	**O**
Flexibility in Maintenance	**O**
GUI Looks	**O**
Texts	**O**
Graphs	**-**
New Widgets	**O**
Robustness	**O**
Performance	**O**
Broad test coverage	**o**
Free of synchronization problems	**-**
Hardware efficiency	**O**
Ease of use	**O**
Combining test suites	**-**
Thorough test results	**-**
Supported platforms	**O**

Table 1. Evaluation of requirements (O = very good, o = good, - = undone)

Some of the obvious successes are the existence of scripting, repeatability (tests and other operations can be rerun), lack of additional hardware requirements, and supported platforms (can run on all the platforms that Java runs on). Simplicity/readability in the scripting language can be measured by comparing it to other

automated GUI testing scripting languages and also by evaluating ease in reading, creating, and modifying the script. In our scripting, because we record the user input by its widget name instead of by pixel location, graphics-based, or lower-level notation (section 2.2), it is definitely easier to understand a test suite. Storing a script into a file in a simple text format with comments also allows users to modify or create any test suite without much effort. When there are changes in the looks of the GUI or texts, it is very flexible to maintain. This is because the system is widget-based. In other words, storing a script and replaying it on a GUI application involves the names of the widgets instead of such things as their locations. Therefore, moving a widget to another location, changing a text in the GUI, or adding a new widget will not require any adjustments. We have not demonstrated how changes in graphs would affect the maintainability, but overall, the application was flexible in its maintenance. As for robustness, because the test does not run on a graphics-based script, it avoids all the problems associated with pixel-by-pixel locations of the buttons, text fields, graphs, windows, icons, etc. Although this widget-based approach requires appropriate script modifications whenever there are changes in widget names, the robustness of replays is raised to the next level. Avoiding all the meaningless recordings and logging just the "important" user inputs, makes replaying a script a much faster process. Another positive aspect of this design is how little of effort it requires for a user to set up and run automated GUI testing/operations. It requires a minor change (using customized classes) in the code of the stand-alone application, and rerunning a script is as simple as pressing a mouse button.

The test coverage is fairly good. Because the Java Accessibility API allows us to access a software window at a very high-level, a test script runs just as if someone was

doing the same series of tasks on the GUI window. However, the widget-based approach faces a minor limitation—the widget names must stay constant. Any change in the widget name requires correcting the name in the existing script.

Some of the things that we leave out in our demonstration are synchronization problems, test suite combinations, and test results. These features/issues can be the subject of further research and development.

7. Conclusion

In this section we will discuss some possible extentions to the system.

7.1. Future Work (Possible Improvements)

While developing the automated GUI testing approach, there were a number of areas left unexplored. To look into these areas more deeply and completely would be the next step in achieving a testing system that is more complete, robust, and fast.

7.1.1. Automated Visual Testing

Because the focus of our research is the automation of the GUI operations, we have not explored testing the correctness of the GUI. Testing whether the operations actually accomplish their actions on replays is investigated and proven. However, a higher level of GUI testing has not really been looked into at all--our project assumes that the changes in the GUI automatically result in a new GUI with a suitable presentation to the user. For instance, the system assimilates a user clicking on the buttons and entering text without problem during test replays. However, what if a change to the number of buttons in a panel causes the buttons to be too small for the labels to display? What if the text box has no room for the appropriate input? These are some serious questions to be

answered. Especially since we are automating Graphical "User" Interface, ignoring visual convenience and compatibility falls short of assimilating user interaction with the computer.

A possible solution to incorporate testing of the looks of the application is to develop a multi-step approach. At the highest level we may have a visual inspection to ensure that the GUI works for people. We can simulate mouse actions to ensure an appropriate mapping of the screen location to the widget. Then we can use our approach that runs using the accessibility functions to simulate events and run a test suite.

Automating the highest level of inspecting the suitability of how a GUI looks for users is a strangely hard problem to explore. The reason is that the program needs to know what looks good to the eye. It also needs to know what the functions of the application and each of its widgets are. We may readily introduce a system that is either intensely intelligent or actually makes the testing process too complicated and difficult for the users to use. This issue introduces an extensive arena of testing to be researched and discussed.

7.1.2. Synchronization

The synchronization problem involves operations within a GUI window as well as those across different pop-up windows. We must ensure that an operation that depends on the result of a previous operation does not start processing until that previous operation finishes its process. This also can be expanded into test suites that run on multiple windows. The concurrency issue must be well examined and its solution carefully implemented.

7.1.3. Test Suites

We could add a feature to manage test suites so that users can work with multiple test suites. Given a list of suites, a user may select one to replay on the application, combine multiple suites into one, subtract one from the combination of suites, and others. This will be useful as GUIs may require multiple sets of tests. In another perspective, a user can divide a long and large test script into more modular, small, and quick sub-suites.

7.1.4. Scripting

First, we will need a checkbox to indicate when to start and stop recording user inputs. Because of the initialization issue (section 5.4.1), the current implementation does not have this feature. The initialization issue should be explored more deeply and a correct start/stop feature should be implemented.

Second, in our script design, we have:

RootName.OtherDescendents'Names.Widget'sParentName.WidgetName:Action#:Arg//Comment

For now, we assume AccessibleName's do not have any .(dot)'s, :(colon)'s, or //(double slashes). Any occurrences of these characters in any AccessibleName will result in an error—parsing incorrectly and not replaying the action.

A possible solution is to put a \(backslash) before all the .(dot)'s, :(colon)'s, and /(slash)'s in Accessible Names (as in some other script languages such as Perl). The parser should parse accordingly.

Another solution is to put an invisible character, i.e. \009 (page break) right in front of the dot separator in the path. The benefit of this approach instead of the previous one is that the script is more readable to the users. However, the flip side is that it cuts down the flexibility in modifying the path in the script.

Lastly, we could make the script more readable. The current script has a long path for each widget and a big portion of each path is shared by many script entries. The script language may be improved by avoiding such repetitive portions. We could specify the application name or a prefix once at the beginning of the script file so that all the entries may share it. Keeping a version number in the script will be a good way of clarifying the contents of a test suite and prevent many incorrect replays.

7.1.5. Test Results

It should analyze the results and report what errors were found in the tests and where they were located. A debugging feature is ideal.

7.2. Accomplishments

As GUI testing has become more complex and voluminous, the need for automating this testing process has surfaced to be one of the critical issues in fast and robust GUI development. Some work has been done in this area, however, this thesis presents a more robust and efficient way to implement the process of automating GUI testing and other operations. The new design is built upon the distributed architecture of Java Swing and the Accessibility API that is applied to establish connectivity between components. This software architecture of automated GUI testing/operations brings about robustness, fast performance, flexibility in maintenance, simplicity for users, and

support for multiple platforms without requiring any extra hardware support. It also enables the script to be readable and maintainable and provides broad test coverage. The new software architecture has accomplished an efficient automated GUI testing/operations. Some future work could be done in the area of synchronization, scripting, test suite management, and test result analysis.

9. Bibliography

1. Scott Dawson, Farnam Jahanian, and Todd Mitton. ORCHESTRA: A Fault Injection Environment for Distributed Systems. University of Michigan, Ann Arbor CSE-TR-318-96, November 1996.
2. Charles B. Owen, Fillia Makedon. Multimedia Data Analysis using ImageTcl (Extended Version), Dartmouth College CS Technical Report PCS-TR97-310, July 1997.
3. Mark Andrews. The Swing Connection: WHAT IS SWING? Sun Microsystem, Inc. java.sun.com/products/jfc/tsc/what_is_swing/what_is_swing.html. March 1999.
4. Mark Andrews. Accessibility and the Swing Set. Sun Microsystem, Inc. java.sun.com/products/jfc/tsc/special_report/accessibility/accessibility.html. March 1999.
5. Trinstan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood and Andy Hopper. Virtual Network Computing. IEEE Internet Computing Volume2, Number 1. January/February 1998.
6. Citrix. MetaFrame. www.citrix.com/products/metaframe.asp.
7. Sun Microsystems, Inc. i-Planet Software Architecture. 1999.
8. Product Specifications. Tarantella Product Information Datasheet Technical Section 11.
9. Mercury Interactive. The Complete Testing Solution for Java-based Applications. 1998.
10. Mercury Interactive. XRunner: The Standard in X Window GUI Testing. www.merc-int.com/products/xrunner4/. 1999.
11. Rational Software Corporation. PreVue-X Product Information. 1999.
12. CenterLine Development Systems, Inc. QC/Replay—GUI Test Automation Designed for Ease-of-Use (Technical Overview). www.centerline.com/productline/qcreplay/qcreplay.html. 1999.
13. Andy Quinn. Tutorial Trail: JavaBeans. java.sun.com/docs/books/tutorial/javabeans/index.html. July 1999.

14. Goldberg. A and Robson. D. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, Mass. 1983.
15. Teradyne, Inc. J990 Series IG900+ Tools Manual: Shmoo. 1998.
16. David M. Geary. Graphic Java 2, Mastering the JFC, 3rd Edition. The Sun Microsystems Press, Java Series. 1999.
17. Sun Microsystems. An Overview of Java, Highlighting Issues of Java Accessibility. March 1998.
18. Collin Stiteler. Java Swing LoanSave, Help Tutorial. 1999.
19. Sun Microsystems. Java Accessibility, Java Accessibility Monkey Example. java.sun.com/products/jfc/jaccess-1.0/doc/Monkey.html. March 1998.
20. Stephen C. Drye, William C. Wake. Java Foundation Classes: Swing Reference. Manning Publications Co. 1999.

Appendix A. Code.

LoanSave/AAJButton.java

```
import javax.swing.*;
import javax.swing.event.*;
import javax.help.*;
import javax.accessibility.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;


// This class is basically JButton class.  What is added is the
// action listener for writing script when the button is clicked.
class AAJButton extends JButton{

   AAJButton(String text){
         super (text);

         // listen for Action (Click) event and record script
         this.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                  String path;
                  path = getPath(getAccessibleContext());
                  saveScript(path, 0, "", "JButton Pressed");
            }});
   }

   // returns a full path from the top level swing component.
   // this is return string value is used to write script.
   String getPath(AccessibleContext thisComp){
         Accessible parent;
         String name;

         //System.out.println(thisComp.getAccessibleName());

         parent = thisComp.getAccessibleParent();
         if (parent == null)  return "LoanSave"; // the top level component
         else {
            name = thisComp.getAccessibleName();
            if (name == null) name = "null";
            return (getPath(parent.getAccessibleContext())+"."+name);}
   }

   // write to the script file.
   // Must not leave file open because the accessing component can open
   // the same file.

    void saveScript(String WidgetPath, int ActionNum,
                        String value, String comment){
```

```
        File f;
        String content = "";
        String entry = "";
    BufferedReader in;
        BufferedWriter out;

        try{
           f = new File("/u/youngkim/1999/LoanSave/script");


           // if the file already exists, it means we just need to
           // concatenate the new script to it.

           if (f.exists()){
                 in = new BufferedReader(new FileReader(f));
                 String line = new String();
                 while ((line = in.readLine()) != null)
                    content = content.concat(line + "\n");
                 in.close();
           }

           out = new BufferedWriter(new FileWriter(f));

           entry = content + WidgetPath + ":" + ActionNum + ":"
                 + value + "//"+ comment + "\n";

           out.write(entry);
           out.close();
        }
        catch (Exception ioe) {
           System.out.println(ioe);
        }
  }
}
```

LoanSave/AAJTable.java

```
import javax.swing.*;
import javax.swing.event.*;
import javax.help.*;
import javax.accessibility.*;
import javax.swing.table.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import java.lang.*;


// This class is basically JTable class.  What is added is the
// action listener for writing script when a cell is updated and
// the AccessibleAction.

class AAJTable extends JTable{

   AAJTable thisTable;
   AbstractTableModel thisModel;

   AAJTable(AbstractTableModel tm){
         super (tm);
         thisModel = tm;
         thisTable = this;


         // listen for the table changed action and save script only when
         // the RUN button is not pressed

         this.getModel().addTableModelListener(new TableModelListener(){
            public void tableChanged(TableModelEvent e){
                  int firstRow = e.getFirstRow(),
                     column = e.getColumn();
                  String path;


                  // when it's not the whole row or all columns that changed...

                  if ((firstRow != TableModelEvent.HEADER_ROW) &&
                     (column != TableModelEvent.ALL_COLUMNS)){

                     path = getPath(getAccessibleContext());

                     saveScript(path+".Row="+firstRow+"Col="+column, 0,
                                 getValueAt(firstRow, column).toString().trim(),
                                 "Table JTable ValueChanged");
                  }

         }});
   }
```

```
// returns a full path from the top level swing component.
// this is return string value is used to write script.

String getPath(AccessibleContext thisComp){

        Accessible parent;
        String name;

        // System.out.println(thisComp.getAccessibleName());

        parent = thisComp.getAccessibleParent();
        if (parent == null)  return "LoanSave";
        else {
           name = thisComp.getAccessibleName();
           if (name == null) name = "null";
           return (getPath(parent.getAccessibleContext())+"."+name);}
}


// write to the script file
// Must not leave file open because the accessing component can open
// the same file.

void saveScript(String WidgetPath, int ActionNum,
                    String value, String comment){

        File f;
        String content = "";
        String entry = "";
    BufferedReader in;
        BufferedWriter out;

        try{
           f = new File("/u/youngkim/1999/LoanSave/script");


           // if the file already exists, it means we just need to
           // concatenate the new script to it.

           if (f.exists()){
                   in = new BufferedReader(new FileReader(f));
                   String line = new String();
                   while ((line = in.readLine()) != null)
                      content = content.concat(line + "\n");
                   in.close();
           }

           out = new BufferedWriter(new FileWriter(f));

           entry = content + WidgetPath + ":" + ActionNum + ":"
                   + value + "//"+ comment + "\n";

           out.write(entry);
           out.close();
        }
        catch (Exception ioe) {
```

```
            System.out.println(ioe);
        }
}


// getAccessibleContext must return accessibleContext with accessibleAction
// implemented.

public AccessibleContext getAccessibleContext() {
        if (accessibleContext == null) {
            accessibleContext = new AccessibleMyJTableComponent();
        }
        return accessibleContext;
}


// This class implements AccessibleAction.

public class AccessibleMyJTableComponent extends AccessibleJComponent
        implements AccessibleAction {

        public AccessibleAction getAccessibleAction() {
            return this;
        }

        public boolean doAccessibleAction(int i){

            if (i==0){
                    //System.out.println("inside doAA setTable");

                    String msg, strArg;
                    int ColIndex, ArgIndex, r, c;

                    // Slice out row, column, and argument
                    msg = thisTable.getAccessibleContext().getAccessibleDescription();
                    System.out.println(msg);
                    ColIndex=msg.indexOf("Col=");
                    ArgIndex=msg.indexOf("Arg=");
                    r = Integer.parseInt(msg.substring(4,ColIndex));
                    c = Integer.parseInt(msg.substring(ColIndex+4, ArgIndex));
                    strArg = msg.substring(ArgIndex+4);
                    strArg = eraseComma(strArg);
                    System.out.println("Row is "+r+". Col is "+c+". Arg is "+strArg);

                    thisTable.setValueAt (new Currency(strArg), r, c);
                    thisModel.fireTableDataChanged();
                    System.out.println("set value and fired event");
                    return true;}
            else return false;}

        public int getAccessibleActionCount(){return 1;}

        public String getAccessibleActionDescription(int i){
            return ("Table Cell Value Changed");}
```

```
    }

    // returns string with commas eradicated from the input string.
    String eraseComma(String str){

            StringBuffer strbuf;
            int index;

            index = str.indexOf(',');
            while (index != -1){
               strbuf = new StringBuffer(str);
               str = new String(strbuf.deleteCharAt(index));
               index = str.indexOf(',');
            }
            return str;
    }
}
```

LoanSave/AAJTextField.java

```
import javax.swing.*;
import javax.swing.event.*;
import javax.help.*;
import javax.accessibility.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;


// This class is basically JTextField class.  What is added is the
// action listener for writing script when a cell is updated and
// the AccessibleAction.

class AAJTextField extends JTextField{
   AAJTextField thisText;

   AAJTextField(String text, int ColNum){

         super (text, ColNum);
         thisText = this;


         // listen for focusLost event and record script

         this.addFocusListener(new FocusListener() {
            public void focusLost(FocusEvent e) {
                  String path;
                  path = getPath(getAccessibleContext());
                  saveScript(path, 0, thisText.getText(),
                              "JTextField Value Changed");
            }
            public void focusGained(FocusEvent e){ }
         });
   }


   // returns a full path from the top level swing component.
   // this is return string value is used to write script.

   String getPath(AccessibleContext thisComp){

         Accessible parent;
         String name;

         //System.out.println(thisComp.getAccessibleName());

         parent = thisComp.getAccessibleParent();
         if (parent == null)  return "LoanSave";
         else {
            name = thisComp.getAccessibleName();
            if (name == null) name = "null";
            return (getPath(parent.getAccessibleContext()) + "."
```

```
                    + name);
        }
}


// write to the script file
// Must not leave file open because the accessing component
// can open the same file.

void saveScript(String WidgetPath, int ActionNum,
                    String value, String comment){

        File f;
        String content = "";
        String entry = "";
    BufferedReader in;
        BufferedWriter out;

        try{
            f = new File("/u/youngkim/1999/LoanSave/script");


            // if the file already exists, it means we just need to
            // concatenate the new script to it.

            if (f.exists()){
                    in = new BufferedReader(new FileReader(f));
                    String line = new String();
                    while ((line = in.readLine()) != null)
                        content = content.concat(line + "\n");
                    in.close();
            }

            out = new BufferedWriter(new FileWriter(f));

            entry = content + WidgetPath + ":" + ActionNum + ":"
                    + value + "//"+ comment + "\n";

            out.write(entry);
            out.close();
        }
        catch (Exception ioe) {
            System.out.println(ioe);
        }
}


// getAccessibleContext must return accessibleContext with
// accessibleAction implemented.

public AccessibleContext getAccessibleContext() {
        if (accessibleContext == null) {
            accessibleContext = new AccessibleMyJTextComponent();
        }
        return accessibleContext;
}
```

```
    // This class implements AccessibleAction.

    public class AccessibleMyJTextComponent
            extends AccessibleJTextComponent
            implements AccessibleAction {

            public AccessibleAction getAccessibleAction() {
               return this;
            }


            // Because the FocusLost event is not triggered, this
            // call will not record a script.

            public boolean doAccessibleAction(int index){
               if (index==0){
                        System.out.println("inside doAA setText");

                        thisText.setText
                           (thisText.getAccessibleContext().getAccessibleDescription());

                        System.out.println("after setting text");
                        return true;
               }
               else return false;
            }

            public int getAccessibleActionCount(){return 1;}

            public String getAccessibleActionDescription(int index){
               return ("Change JTextField Value");}
    }
}
```

LoanSave/Both.java

```
public class Both {

 Loan           L;
 Save           S;
 boolean        trace = false;

 //      Both's Data

 int            loan_months;
 int            save_months;
 int            months;
 int            years;

 double loan_starting_balance;
 double save_starting_balance;

 double loan_intrate;
 double save_intrate;

 double payment;
 double deposit;

 double loan_taxrate;
 double save_taxrate;

 double loan_balance[] = new double[100];
 double save_balance[] = new double[100];
 double net_interest[] = new double[100];
 double net_taxes[] = new double[100];
 double years_deposits[] = new double[100];
 double years_payments[] = new double[100];

 double cumulative_interest;
 double cumulative_net_interest;
 double ending_balance;
 double CAGR;

 Both (LoanSave parent) {
        L = parent.L;
        S = parent.S;

        run_both ();
 }

 void run_both () {

        //      Copy data from Loan, Save objects

// This version makes loan payments out of savings
// Taxes saved on loan payments are deposited into savings at
// 12 month (tax time) intervals

   double daily_rate;
   double daysint;
```

```
double Loan_MonthsInt;
double Save_MonthsInt;
double Loan_YearsInt;
double Save_YearsInt;
double Loan_Tax;
double Save_Tax;
double Loan_Bal;
double Save_Bal;

int monthdays;
int year;
int month;
int day;

loan_starting_balance = L.balance;
save_starting_balance = S.balance;
loan_intrate = L.intrate;
save_intrate = S.intrate;
daily_rate = save_intrate / 36500;
loan_months = L.months;
save_months = S.months;
loan_taxrate = L.taxrate;
save_taxrate = S.taxrate;
payment = L.payment;
deposit = S.deposit;
year = 1;

if (loan_months > save_months) {
   months = loan_months;
}
else {
   months = save_months;
}

if (trace == true) {
       System.out.println ("RUN_BOTH ():");
       System.out.println ("AT START, " +
                           " loan_balance = " + loan_starting_balance +
                           " save_balance = " + save_starting_balance +
                           " months = " + months);
}

Loan_Bal = loan_starting_balance;
Save_Bal = save_starting_balance;
Loan_YearsInt = 0;
Save_YearsInt = 0;

for (month = 1; month <= months; month++) {

//  Error Checks

   if (Save_Bal < 0) {
                           break;
   }

//  Handle one month's Savings Interest and deposit
```

```
// Use "daily Interest" compounding

   switch (month % 12) {
                        case 2:
        if (year % 4 == 0) {
           monthdays = 29;
        }
                                    else {
           monthdays = 28;
        }
                                    break;
      case 4:
      case 6:
      case 9:
      case 11:
         monthdays = 30;
                                    break;
      default:
         monthdays = 31;
                                    break;
               }

   Save_MonthsInt = 0;
   for (day = 1; day <= monthdays; day++) {
      daysint = Save_Bal * daily_rate;
      Save_Bal += daysint;
      Save_MonthsInt += daysint;
   }

   Save_YearsInt += Save_MonthsInt;
   Save_Bal += deposit;

        //      Handle One Month's Payment

   Loan_MonthsInt = Loan_Bal * (loan_intrate / 1200);
   Loan_YearsInt += Loan_MonthsInt;
   if (Loan_Bal >= (payment - Loan_MonthsInt)) {
      Loan_Bal -= (payment - Loan_MonthsInt);
      Save_Bal -= payment;
   }
        else if (Loan_Bal > 0) {
               Save_Bal -= Loan_Bal;
      Loan_Bal = 0;
   }

        //      Handle end-of-year

   if (((month % 12) == 0) || (month == months)) {

               // Pay year's tax on Savings income out of Savings balance

      Save_Tax = Save_YearsInt * (save_taxrate / 100);
      Save_Bal -= Save_Tax;
      years_deposits[year] = deposit * 12;

               // Deposit year's tax savings from Loan Interest into Savings
```

```
        Loan_Tax = Loan_YearsInt * (loan_taxrate / 100);
        Save_Bal += Loan_Tax;

                //      Save Year's Data

                if ((month == months) && (month % 12 != 0))
                        ++year;
                loan_balance[year] = nearest_cent(Loan_Bal);
                save_balance[year] = nearest_cent(Save_Bal);
                net_interest[year] = nearest_cent(Save_YearsInt - Loan_YearsInt);
                net_taxes[year] = nearest_cent(Save_Tax - Loan_Tax);

                Loan_YearsInt = 0;
                Save_YearsInt = 0;
                years = year;

                if (trace == true) {
                   System.out.println (
                        "year = " + year +
                        " loan_balance = " + loan_balance[year] +
                        " save_balance = " + save_balance[year] +
                        " net_taxes = " + net_taxes[year] +
                        " net_interest = " + net_interest[year]);
                }

                ++year;
      }
    }
  }

  double nearest_cent (double d) {
        long    L;

        L = (long)((d + 0.005) * 100);
        d = (double)(L);
        d /= 100;
        return (d);
  }
}
```

LoanSave/ControlPanel.java

```
//REQUIRES ALL THE ACCESSIBLE NAMES TO NOT CONTAIN DOTS.


import javax.swing.*;
import javax.swing.event.*;
import javax.help.*;
import javax.accessibility.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class ControlPanel extends JPanel {

   boolean        trace = false;

   // An event stack, NOT a boolean !!
   int            program_changing_Loan_text = 0;
   int            program_changing_Save_text = 0; // ditto

   LoanSave       parent;
   Loan           L;
   Save           S;
   Both           B;

   LoanControls   LoanPanel;
   SaveControls   SavePanel;


   //     Constructor

   public ControlPanel(LoanSave target) {

         // set accessible name
         getAccessibleContext().setAccessibleName("ControlPanel");

         parent = target;

         L = parent.L;
         S = parent.S;
         B = parent.B;

         setLayout(new GridLayout(1,2,2,2));

         //        Create The Control Panels

         LoanPanel = new LoanControls (this);
         SavePanel = new SaveControls (this);

         //        Put Control Panels On The Screen

         add (LoanPanel);
         add (SavePanel);
   }
```

```
public void update_loan (LoanControls LoanPanel) {
        String S_balance = LoanPanel.principal_field.getText();
        String S_intrate = LoanPanel.intrate_field.getText();
        String S_months  = LoanPanel.months_field.getText();
        String S_payment = LoanPanel.payment_field.getText();
        String S_taxrate = LoanPanel.taxrate_field.getText();
        Double  D_balance, D_intrate, D_payment, D_taxrate;
        Integer  I_months;

        if (trace == true)
           System.out.println ("In ControlPanel.update_loan () (sent panel input to Loan object)");

        //        Parse the strings (error check)

        D_balance = Double.valueOf(S_balance);
        D_intrate = Double.valueOf(S_intrate);
        I_months = Integer.valueOf(S_months);
        D_payment = Double.valueOf(S_payment);
        D_taxrate = Double.valueOf(S_taxrate);

        //        pass the string values to Loan object

        L.balance = D_balance.doubleValue();
        L.intrate = D_intrate.doubleValue();
        L.months = I_months.intValue();
        L.payment = D_payment.doubleValue();
        L.taxrate = D_taxrate.doubleValue();
}

public void update_save (SaveControls SavePanel) {
        String S_balance =        SavePanel.balance_field.getText();
        String S_intrate =SavePanel.intrate_field.getText();
        String S_months =         SavePanel.months_field.getText();
        String S_deposit =        SavePanel.deposit_field.getText();
        String S_taxrate =        SavePanel.taxrate_field.getText();
        String S_final_balance = SavePanel.final_balance_field.getText();
        String S_CAGR = SavePanel.CAGR_field.getText ();

        Double  D_balance, D_intrate, D_deposit,
           D_taxrate, D_final_balance, D_CAGR;
        Integer  I_months;

        if (trace == true)
           System.out.println
                ("In ControlPanel.update_save () (send panel text to Save object)");

        //        Parse the strings (error check)

        D_balance = Double.valueOf(S_balance);
        D_intrate = Double.valueOf(S_intrate);
        I_months = Integer.valueOf(S_months);
        D_deposit = Double.valueOf(S_deposit);
        D_taxrate = Double.valueOf(S_taxrate);
        D_final_balance = Double.valueOf(S_final_balance);
        D_CAGR = Double.valueOf(S_CAGR);
```

```
        //      pass the string values to Save object

        S.balance = D_balance.doubleValue();
        S.intrate = D_intrate.doubleValue();
        S.months = I_months.intValue();
        S.deposit = D_deposit.doubleValue();
        S.taxrate = D_taxrate.doubleValue();
        S.ending_balance = D_final_balance.doubleValue();
        S.CAGR = D_CAGR.doubleValue();
    }

    double nearest_cent (double d) {
        long    L;

        L = (long)((d + 0.005) * 100);
        d = (double)(L);
        d /= 100;
        return (d);
    }

    public void hang () {

        //      THIS WILL HANG THE COMPUTER !!

        int     x,y,z;

        x = 0;
        y = 0;
        z = 0;

        while (++x != 0) {
          while (++y != 0) {
                while (++z != 0)
                  ;
          }
        }
    }
}

class LoanControls extends  JPanel
    implements DocumentListener, ActionListener {

    AAJTextField  principal_field, intrate_field, months_field;
    AAJTextField  payment_field, taxrate_field, run_field;
    AAJTextField  total_payments_field, net_cost_field;
    AAJButton     principal_button, intrate_button, months_button;
    AAJButton     payment_button, run_button;
    JLabel        taxratelabel, totalpaymentslabel, netcostlabel;
    ControlPanel  parent;
    boolean       text_changed;
    Loan          L;
    boolean       trace;

    //    LoanControls Panel constructor
```

```
LoanControls (ControlPanel target) {

        // set the accessible name
        getAccessibleContext().setAccessibleName("LoanControls");

        //          save ("pointer" to) parent object

        parent = target;
        L = parent.L;
        trace = parent.trace;

        //          create panel

        setLayout(new GridLayout(9,2,5,5));          // 9 rows  2 columns

        //          build panel

        principal_button = new AAJButton ("Principal");
        principal_button.addActionListener(this);
        principal_button.setFont(parent.parent.ButtonFont);
        add (principal_button);
        CSH.setHelpIDString(principal_button, "Loan_Principal.field");

        principal_field = new AAJTextField(String.valueOf(parent.L.balance), 10);
        // set accessible name
        principal_field.getAccessibleContext().setAccessibleName("Pl");
    principal_field.getDocument().addDocumentListener(this);

        principal_field.setFont(parent.parent.TextFont);
        add (principal_field);
        CSH.setHelpIDString(principal_field, "Loan_Principal.button");

        intrate_button = new AAJButton ("Interest Rate");
        intrate_button.addActionListener(this);
        intrate_button.setFont(parent.parent.ButtonFont);
        add (intrate_button);
        CSH.setHelpIDString(intrate_button, "Loan_Interest_Rate.field");

        intrate_field = new AAJTextField(String.valueOf(parent.L.intrate), 10);
        // set accessible name
        intrate_field.getAccessibleContext().setAccessibleName("IR");
    intrate_field.getDocument().addDocumentListener(this);

        intrate_field.setFont(parent.parent.TextFont);
        add (intrate_field);
        CSH.setHelpIDString(intrate_field, "Loan_Interest_Rate.field");


        months_button = new AAJButton ("Months");
        months_button.addActionListener(this);
        months_button.setFont(parent.parent.ButtonFont);
        add (months_button);
        CSH.setHelpIDString(months_button, "Loan_Months.field");

        months_field = new AAJTextField(String.valueOf(parent.L.months), 10);
        // set accessible name
```

```
    months_field.getAccessibleContext().setAccessibleName("M");
months_field.getDocument().addDocumentListener(this);

    months_field.setFont(parent.parent.TextFont);
    add (months_field);
    CSH.setHelpIDString(months_field, "Loan_Months.field");


    payment_button = new AAJButton ("Payment");
    payment_button.addActionListener(this);
    payment_button.setFont(parent.parent.ButtonFont);
    add (payment_button);
    CSH.setHelpIDString(payment_button, "Loan_Payment.field");

    payment_field = new AAJTextField(String.valueOf(parent.L.payment), 10);
    // set accessible name
    payment_field.getAccessibleContext().setAccessibleName("Pt");
payment_field.getDocument().addDocumentListener(this);

    payment_field.setFont(parent.parent.TextFont);
    add (payment_field);
    CSH.setHelpIDString(payment_field, "Loan_Payment.field");


    taxratelabel = new JLabel ("Tax Rate %", JLabel.CENTER);
    taxratelabel.setFont(parent.parent.ButtonFont);
    add (taxratelabel);

    taxrate_field = new AAJTextField(String.valueOf(parent.L.taxrate), 10);
    // set accessible name
    taxrate_field.getAccessibleContext().setAccessibleName("TR");
taxrate_field.getDocument().addDocumentListener(this);

    taxrate_field.setFont(parent.parent.TextFont);
    add (taxrate_field);
    CSH.setHelpIDString(taxrate_field, "Tax_Rate.field");


    run_button = new AAJButton ("RUN");
    run_button.addActionListener(this);
    run_button.setFont(parent.parent.ButtonFont);
    add (run_button);
    CSH.setHelpIDString(run_button, "Loan_Run.button");

    JLabel L2;
    add (L2 = new JLabel ("PLACEHOLDER2", JLabel.CENTER));
    L2.setVisible(false);

    /*
     * NOTE: although "Total Payments" and "Net Cost" do not accept input,
     * a user can edit these text fields, causing them to disagree with other
     * information on the screen. Hence we must grey the tables, and hence
     * these text fields need a document listener.
     */

    totalpaymentslabel = new JLabel ("Total Payments", JLabel.CENTER);
```

```
        totalpaymentslabel.setFont(parent.parent.ButtonFont);
        add (totalpaymentslabel);

        total_payments_field = new AAJTextField (String.valueOf
                                                (parent.L.total_payments), 10);

        // set accessible name
        total_payments_field.getAccessibleContext().setAccessibleName("TP");
    total_payments_field.getDocument().addDocumentListener(this);

        total_payments_field.setFont(parent.parent.TextFont);
        add (total_payments_field);
        CSH.setHelpIDString(total_payments_field, "Total_Payments.field");


        netcostlabel = new JLabel ("Net Cost", JLabel.CENTER);
        netcostlabel.setFont(parent.parent.TextFont);
        add (netcostlabel);

        net_cost_field = new AAJTextField(String.valueOf(parent.L.net_cost), 10);
        // set accessible name
        net_cost_field.getAccessibleContext().setAccessibleName("NC");
    net_cost_field.getDocument().addDocumentListener(this);

        net_cost_field.setFont(parent.parent.TextFont);
        add(net_cost_field);
        CSH.setHelpIDString(net_cost_field, "Net_Cost.field");

        text_changed = false;
}

//      Document Listener Interface Implementation

//      These functions get called when a TextField is modified

public void changedUpdate(DocumentEvent DE) {
        text_changed = true;
        parent.parent.Set_Loan_Grey();
        System.out.println("changedUpdate");
}
public void insertUpdate(DocumentEvent DE) {
        text_changed = true;
        parent.parent.Set_Loan_Grey();
        System.out.println("insertUpdate");
}
public void removeUpdate(DocumentEvent DE) {
        text_changed = true;
        parent.parent.Set_Loan_Grey();
        System.out.println("removeUpdate");
}

//      Action Event Interface Implementation

//      This function gets called when a Button is pressed

public void actionPerformed(ActionEvent e) {
```

```
String S = e.getActionCommand ();
String S2 = "";

if (trace == true) {

   System.out.println
         ("In LoanControls.actionPerformed (), getActionCommand == \""
          + e.getActionCommand() + "\"");

   System.out.println ("text_changed = " + text_changed);
}

//         'update_loan' READS the control panel; this is the
//         mechanism for passing changes into the Loan object...

if (text_changed == true) {
   parent.update_loan (this);

}

//         Which button was pressed ?

if (S.compareTo ("Principal") == 0) {
   L.Find_balance ();
   S2 = S2.valueOf(L.balance);
   parent.LoanPanel.principal_field.setText(S2);
   parent.parent.Set_Loan_Grey();
}
else if (S.compareTo ("Interest Rate") == 0) {
   L.Find_intrate ();
   S2 = S2.valueOf(L.intrate);
   parent.LoanPanel.intrate_field.setText(S2);
   parent.parent.Set_Loan_Grey();
}
else if (S.compareTo ("Months") == 0) {
   L.Find_months();
   S2 = S2.valueOf(L.months);
   parent.LoanPanel.months_field.setText(S2);
   parent.parent.Set_Loan_Grey();
}
else if (S.compareTo ("Payment") == 0) {
   L.Find_payment ();
   S2 = S2.valueOf(L.payment);
   parent.LoanPanel.payment_field.setText(S2);
   parent.parent.Set_Loan_Grey();
}
else if (S.compareTo ("RUN") == 0) {
   L.run_loan ();
   update ();
   parent.parent.tablepanel.LT.update ();
   parent.B.run_both ();
   parent.parent.tablepanel.BT.update ();
   text_changed = false;
   parent.parent.Set_Loan_White();
}
else      {
```

```
            System.out.println("ActionPerformed() called with \"" + S + "\"");
        }
    }

    void update () {
        if (trace == true)

            System.out.println
                ("In LoanControls.update ()  (write to total payments and net cost fields)");

        ++parent.program_changing_Loan_text;
        total_payments_field.setText(String.valueOf(parent.L.total_payments));
        ++parent.program_changing_Loan_text;
        net_cost_field.setText(String.valueOf(parent.L.net_cost));
    }

}

class GoalListener implements ActionListener {

    int             id;
    static int      last;
    AAJTextField textfield;

    //      GoalListener Constructor

    GoalListener (int x, AAJTextField TF) {
        id = x;                         //      ERROR CHECKS ?
        textfield = TF;
        last = 0;
    }

    boolean last_changed () {
        if (last == id)
            return (true);
        else
            return (false);
    }

    void reset_last_changed () {
        last = 0;
    }

    public void actionPerformed(ActionEvent e) {
        /*
         if (true) {
         System.out.println ("In GoalListener.textValueChanged(), ID = " + id);
         }
        */
        last = id;
    }
}

class SaveControls extends   JPanel
    implements DocumentListener, ActionListener {
```

```
AAJTextField balance_field, intrate_field, months_field, deposit_field;
AAJTextField taxrate_field, run_field, final_balance_field, CAGR_field;
AAJButton    balance_button, intrate_button, months_button;
AAJButton    deposit_button, run_button;
JLabel       taxratelabel, finalbalancelabel, cagrlabel;
ControlPanel parent;
boolean      text_changed;
Save         S;
boolean      trace;
GoalListener ML1, ML2;

//      SaveControls Panel constructor

SaveControls (ControlPanel target) {

        // set accessible name
        getAccessibleContext().setAccessibleName("SaveControls");

        //          save ("pointer" to) parent object

        parent = target;
        S = parent.S;
        trace = parent.trace;

        if (trace == true)
           System.out.println ("In SaveControls () (constructor)");

        //          create panel

        setLayout(new GridLayout(9,2,5,5));

        //          build panel

        balance_button = new AAJButton ("Balance");
        balance_button.addActionListener(this);
        balance_button.setFont(parent.parent.ButtonFont);
        add (balance_button);
        CSH.setHelpIDString(balance_button, "Save_Balance.field");

        balance_field = new AAJTextField(String.valueOf(parent.S.balance), 10);
        // set accessible name
        balance_field.getAccessibleContext().setAccessibleName("B");
    balance_field.getDocument().addDocumentListener(this);

        balance_field.setFont(parent.parent.TextFont);
        add (balance_field);
        CSH.setHelpIDString(balance_field, "Save_Balance.field");


        intrate_button = new AAJButton ("Interest Rate");
        intrate_button.addActionListener(this);
        intrate_button.setFont(parent.parent.ButtonFont);
        add (intrate_button);
        CSH.setHelpIDString(intrate_button, "Save_Interest_Rate.field");

        intrate_field = new AAJTextField(String.valueOf(parent.S.intrate), 10);
```

```
    // set accessible name
    intrate_field.getAccessibleContext().setAccessibleName("IR");
intrate_field.getDocument().addDocumentListener(this);

    intrate_field.setFont(parent.parent.TextFont);
    add (intrate_field);
    CSH.setHelpIDString(intrate_field, "Save_Interest_Rate.field");


    months_button = new AAJButton ("Months");
    months_button.addActionListener(this);
    months_button.setFont(parent.parent.ButtonFont);
    add (months_button);
    CSH.setHelpIDString(months_button, "Save_Months.field");

    months_field = new AAJTextField(String.valueOf(parent.S.months), 10);
    // set accessible name
    months_field.getAccessibleContext().setAccessibleName("M");
months_field.getDocument().addDocumentListener(this);

    months_field.setFont(parent.parent.TextFont);
    add (months_field);
    CSH.setHelpIDString(months_field, "Save_Months.field");


    deposit_button = new AAJButton ("Deposit");
    deposit_button.addActionListener(this);
    deposit_button.setFont(parent.parent.ButtonFont);
    add (deposit_button);
    CSH.setHelpIDString(deposit_button, "Save_Deposit.field");

    deposit_field = new AAJTextField(String.valueOf(parent.S.deposit), 10);
    // set accessible name
    deposit_field.getAccessibleContext().setAccessibleName("D");
deposit_field.getDocument().addDocumentListener(this);

    deposit_field.setFont(parent.parent.TextFont);
    add (deposit_field);
    CSH.setHelpIDString(deposit_field, "Save_Deposit.field");


    taxratelabel = new JLabel ("Tax Rate %", JLabel.CENTER);
    taxratelabel.setFont(parent.parent.ButtonFont);
    add (taxratelabel);

    taxrate_field = new AAJTextField(String.valueOf(parent.S.taxrate), 10);
    // set accessible name
    taxrate_field.getAccessibleContext().setAccessibleName("TR");
taxrate_field.getDocument().addDocumentListener(this);

    taxrate_field.setFont(parent.parent.TextFont);
    add (taxrate_field);
    CSH.setHelpIDString(taxrate_field, "Tax_Rate.field");


    run_button = new AAJButton ("RUN");
```

```
        run_button.addActionListener(this);
        run_button.setFont(parent.parent.ButtonFont);
        add (run_button);
        CSH.setHelpIDString(run_button, "Save_Run.button");

        JLabel L2;
        add (L2 = new JLabel ("PLACEHOLDER2", JLabel.CENTER));
        L2.setVisible(false);


        finalbalancelabel = new JLabel ("Final Balance", JLabel.CENTER);
        finalbalancelabel.setFont(parent.parent.ButtonFont);
        add (finalbalancelabel);
        CSH.setHelpIDString(finalbalancelabel, "Final_Balance.field");

        final_balance_field = new AAJTextField(String.valueOf
                                                (parent.S.ending_balance), 10);

        // set accessible name
        final_balance_field.getAccessibleContext().setAccessibleName("FB");
        ML1 = new GoalListener (1, final_balance_field);
        final_balance_field.addActionListener(ML1);
    final_balance_field.getDocument().addDocumentListener(this);

        final_balance_field.setFont(parent.parent.TextFont);
        add (final_balance_field);
        CSH.setHelpIDString(final_balance_field, "Final_Balance.field");


        cagrlabel = new JLabel ("CAGR %", JLabel.CENTER);
        cagrlabel.setFont(parent.parent.TextFont);
        add (cagrlabel);
        CSH.setHelpIDString(cagrlabel, "CAGR.field");

        CAGR_field = new AAJTextField(String.valueOf(parent.S.CAGR), 10);
        // set accessible name
        CAGR_field.getAccessibleContext().setAccessibleName("C");
        ML2 = new GoalListener (2, CAGR_field);
        CAGR_field.addActionListener(ML2);
    CAGR_field.getDocument().addDocumentListener(this);

        CAGR_field.setFont(parent.parent.TextFont);
        add (CAGR_field);
        CSH.setHelpIDString(CAGR_field, "CAGR.field");

        text_changed = false;
}

//      Document Listener Interface Implementation

//      These functions get called when a TextField is modified

public void changedUpdate(DocumentEvent DE) {
        text_changed = true;
        parent.parent.Set_Save_Grey();
}
```

```
public void insertUpdate(DocumentEvent DE) {
        text_changed = true;
        parent.parent.Set_Save_Grey();
}
public void removeUpdate(DocumentEvent DE) {
        text_changed = true;
        parent.parent.Set_Save_Grey();
}

//      Action Event Interface Implementation

public void actionPerformed(ActionEvent e) {
        String S1 = e.getActionCommand ();
        String S2 = "";

        if (trace == true) {
           System.out.println
                ("In SaveControl.actionPerformed (), getActionCommand == "
                 + e.getActionCommand());
        }

        //      If ANY field's text changed,
        //      push the new text into the Save Object

        if (text_changed == true) {
           parent.update_save (parent.SavePanel);
        }

        //      Which button was pressed ?

        if (S1.compareTo ("Balance") == 0) {
           if (ML1.last_changed () == true)
                S.Find_Balance (1);
           else if (ML2.last_changed () == true)
                S.Find_Balance (2);
           else
                S.Find_Balance (1);
           S2 = S2.valueOf(S.balance);
           parent.SavePanel.balance_field.setText(S2);
           parent.parent.Set_Save_Grey();
        }
        else if (S1.compareTo ("Interest Rate") == 0) {
           if (ML1.last_changed () == true)
                S.Find_Intrate (1);
           else if (ML2.last_changed () == true)
                S.Find_Intrate (2);
           else
                S.Find_Intrate (1);
           S2 = S2.valueOf(S.intrate);
           parent.SavePanel.intrate_field.setText(S2);
           parent.parent.Set_Save_Grey();
        }
        else if (S1.compareTo ("Months") == 0) {
           if (ML1.last_changed () == true)
                S.Find_Months (1);
           else if (ML2.last_changed () == true)
```

```
                S.Find_Months (2);
            else
                S.Find_Months (1);
            S2 = S2.valueOf(S.months);
            parent.SavePanel.months_field.setText(S2);
            parent.parent.Set_Save_Grey();
        }
        else if (S1.compareTo ("Deposit") == 0) {
            if (ML1.last_changed () == true)
                S.Find_Deposit (1);
            else if (ML2.last_changed () == true)
                S.Find_Deposit (2);
            else
                S.Find_Deposit (1);
            S2 = S2.valueOf(S.deposit);
            parent.SavePanel.deposit_field.setText(S2);
            parent.parent.Set_Save_Grey();
        }
        else if (S1.compareTo ("RUN") == 0) {
            if (trace == true)
                System.out.println ("   running updates...");

            S.run_save ();
            parent.parent.tablepanel.ST.update ();
            parent.B.run_both ();
            parent.parent.tablepanel.BT.update ();
            update ();
            text_changed = false;
            ML1.reset_last_changed ();
            parent.parent.Set_Save_White();
        }
        else
            return;
    }

    void update () {
        if (trace == true)

            System.out.println
                ("In SaveControls.update () (write to final balance and CAGR fields)");

        parent.program_changing_Save_text += 1;
        final_balance_field.setText(String.valueOf(parent.S.ending_balance));
        parent.program_changing_Save_text += 1;
        CAGR_field.setText(String.valueOf(parent.S.CAGR));
        ML1.reset_last_changed();
    }
}
```

LoanSave/Currency.java

```
/*
 *      Currency class
 *
 *      This class extends Double, to provide nicely-formatted Strings
 *      for printing Doubles which represent currency amounts, by overriding
 *      toString() in Double.
 *
 *      The class effectively fixes a Java bug when printing currency, which
 *      is the inability to specify a precision (as you could with printf()).
 *      The result is that values ending in '0', such as $10.50, are printed
 *      as "$10.5".
 *
 */

import java.awt.*;

public class Currency  {

  Double D;
  int           field_columns;
  int           fraction_digits;
  boolean       print_dollarsign;

  public Currency (double d) {
        d = Nearest_Cent(d);
        D = new Double(d);
        init();
  }
  public Currency (String S) throws NumberFormatException {
        D = new Double(S);
        init();
  }
  public Currency (double d, int cols, int nfracs, boolean b) {
        d = Nearest_Cent(d);
        D = new Double(d);
        if (cols >= 0)
              field_columns = cols;
        else
              field_columns = 2;
        if (nfracs >= 0)
              fraction_digits = nfracs;
        else
              fraction_digits = 2;
        print_dollarsign = b;
  }
  public Currency (double d, boolean b) {
        d = Nearest_Cent(d);
        D = new Double(d);
        field_columns = 2;
        fraction_digits = 2;
        print_dollarsign = b;
  }
  private void init() {
        fraction_digits = 2;
```

```
        field_columns = 2;
        print_dollarsign = false;
    }
    public double Value () {
        return (D.doubleValue());
    }
    public Double DoubleValue () {
        return (D);
    }
    public String toString() {
        return (double_comma(Value()));
    }
    public void set_minColumns(int mincol) {
        if (mincol > 0)
            field_columns = mincol;
    }
    public int set_minColumns(double maxvalue) {
        field_columns = 2;
        while ((maxvalue /= 10) > 0.0)
            ++field_columns;
        return (field_columns);
    }
    public static int get_minColumns(double maxvalue) {
        int     cols = 0;

        if (maxvalue < 0.0)
            maxvalue *= -1.0;
        while ((maxvalue /= 10) > 1.0)
            ++cols;
        return (cols);
    }
    public void set_fraction_digits(int ndigits) {
        if (ndigits > 0)
            fraction_digits = ndigits;
    }
    public void print_$ (boolean b) {
        print_dollarsign = b;
    }

/*
 *      This function takes a double and formats it with commas, for
 *      "natural" presentation:
 *
 *          1436672.87      -->     1,436,672.87
 *
 *      The caller specifies the minimum field size
 *          (columns to the left of the '.')
 *      and the desired precision (
 *          (columns to the right of the '.')
 *
 *      The return value is a new, ready-to-print String.
 *
 *      PORTED from a very old C function...
 *
 */
```

```
public String double_comma (double d) {

        StringBuffer    Commabuf;
        String          S;

        double  saved;
        double  remd;
        int     value_columns;
        int     value_commas = 0;
        int     v;
        int     field_commas = 0;
        int     f;
        int     index;
        double  mult = 1.0;
        char    c;
        int     x;
        int     savem = 0;
        boolean negative;

        //      Negative ?

        if (d < 0) {
                negative = true;
                d *= -1.0;
        }
        else
                negative = false;

        saved = d;
        remd = d;

        //      How Many Int Digit Columns In This Value ?

        value_columns = 1;
        while ((d /= 10) >= 1.0) {
                ++value_columns;
                mult *= 10;
        }

        //      How Many Commas In This Value?

        value_commas = value_columns / 3;
        if (value_commas != 0 && (value_columns % 3 == 0))
                value_commas -= 1;

        //      How Many Commas in Field Size ?

        field_commas = field_columns / 3;
        if (field_commas != 0 && (field_columns % 3 == 0))
                field_commas -= 1;

        //      Pad Up to Specified Field Size

        v = value_commas + value_columns;
        f = field_commas + field_columns;
```

```
        x = v > f ? v + 5 : f + 5;
        if (print_dollarsign == true)
                x += 2;

        //      Create StringBuffer

        Commabuf = new StringBuffer(x);

        //      Build String in StringBuffer

//      Commabuf.append('\"');

        //      $

        if (print_dollarsign == true) {
                Commabuf.append('$');
                Commabuf.append(' ');
                index = 2;
        }
        else
                index = 0;

        //      PAD actual column requirements up to specified number of columns

        while (v < f) {
                ++index;
                Commabuf.append(' ');
                f--;
                ++savem;
        }

        if (negative == true) {
                if (index >= 2)
                        Commabuf.setCharAt(index - 2, '-');
                else if (index == 1)
                        Commabuf.setCharAt(0, '-');
                else
                        Commabuf.append('-');
        }

        //      Print Whole Digits and Commas

        while (value_columns != 0)          {
                if (print_dollarsign == true) {
                   if (value_columns % 3 == 0 && ((index - 2)  > savem)) {
                        ++index;
                        Commabuf.append(',');
                   }
                }
                else      {
                   if (value_columns % 3 == 0 && index  > savem) {
                        ++index;
                        Commabuf.append(',');
                   }
                }
                ++index;
```

```
                Commabuf.append((char)(((int) (remd / mult)) + '0'));
                while (remd >= mult)
                        remd -= mult;
                mult /= 10;
                --value_columns;
        }

        if (fraction_digits > 0)
                Commabuf.append('.');

        //      Print Fractional Digits

        f = fraction_digits;

        while (f-- > 0) {
                x = (int) (remd / mult);
                x += '0';
                c = (char) x;
                Commabuf.append(c);
                while (remd >= mult)
                        remd -= mult;
                mult /= 10;
        }
//      Commabuf.append('\"');
        S = new String(Commabuf);
//System.out.println("FINAL STRING = \"" + S + "\",  Length = " + S.length());
        return (S);
  }

  //    Round Off Value To Nearest Whole Cent

  public double Nearest_Cent (double d) {
        long    L;

        L = (long)((d + 0.005) * 100);

        d = (double)(L);
        d /= 100;

        return (d);
  }

}
```

LoanSave/Loan.java

```
public class Loan {
 boolean        trace = false;

 //      Loan's Data

 //      NOTE:  Loan's Data is distinguished from the corresponding
 //              entries in the Control Panel

 int            months = 180;
 int            years = 15;
 boolean        months_changed = true;

 double balance = 120000;
 boolean        balance_changed = true;

 double intrate = 7.0;
 boolean        intrate_changed = true;

 double payment;
 double total_payments;
 boolean        payment_changed = false;

 double minimum_payment = 0.0;
 double taxrate = 32.0;
 double resolution = 0.01;

 double years_balance[] = new double[101];
 double years_interest[] = new double[101];
 double years_taxes_saved[] = new double[101];
 double years_net_payment[] = new double[101];
 double cumulative_interest;
 double cumulative_net_interest;
 double net_cost;
 double ending_balance;

 Loan () {

        //      find initial payment value (from starting data)

        payment = nearest_cent (Find_payment ());
        minimum_payment = (nearest_cent (balance * (intrate / 1200)));
        run_loan ();
 }

 //      This function runs the loan and puts the summary into variables

 void run_loan () {
        double  bal = balance;
        double  monthly_rate = intrate / 1200;
        double  months_int;
        double  years_int = 0.0;
        int             month;
        int             year = 1;
```

```
        if (trace == true)
                System.out.println ("In RUN_LOAN ()");

        minimum_payment = nearest_cent (balance * monthly_rate);
        cumulative_interest = 0.0;
        cumulative_net_interest = 0.0;
        years_balance[0] = balance;

        for (month = 1; month <= months; month++) {
                months_int = bal * monthly_rate;
                years_int += months_int;
                bal -= (payment - months_int);
                if ((month % 12 == 0) || (month == months)) {
                        year = month / 12;
                        if ((month % 12) != 0)
                                ++year;
                        years_balance[year] = nearest_cent (bal);
                        years_interest[year] = nearest_cent(years_int);
                        years_taxes_saved[year] = nearest_cent (years_int * (taxrate/100));
                        years_net_payment[year] = nearest_cent(((payment * 12) -
years_taxes_saved[year])/12);
                        cumulative_interest += nearest_cent (years_int);
                        cumulative_net_interest += (years_int - years_taxes_saved[year]);

                        years_int = 0.0;
                        if (trace == true) {
System.out.println ("Year " + year + "   Balance = " + bal);
                        }
                }
        }
        ending_balance = nearest_cent(bal);
        net_cost = nearest_cent(balance + cumulative_net_interest);
        total_payments = nearest_cent(payment * months);
        years = year;
 }

 double Find_payment () {

        //      Adjust / Until (you find the payment...)

        int     month;
        int     iterations = 0;
        double  bal = balance;
        double  monthly_rate = intrate / 1200;
        double  increment = balance;
        double  Payment = increment;

        if (trace == true)
                System.out.println ("In Find_payment");

        while (bal > resolution || bal < -resolution) {
                ++iterations;
                bal = balance;

                if (trace == true) {
                        System.out.println (
```

```
"Iteration " + iterations + "  Payment = " + Payment);
                }

                for (month = 1; month <= months + 1; month++) {
                        bal -= Payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }

                increment /= 2;

                if (month < months)
                        Payment -= increment;
                else if (month > months)
                        Payment += increment;
                else    {
                        if (bal > resolution)
                                Payment += increment;
                        else if (bal < 0.0)
                                Payment -= increment;
                        else
                                break;
                }
        }
        if (trace == true)
                System.out.println ("Find_payment = " + Payment);
        payment = nearest_cent(Payment);
        return (payment);
 }

 double Find_balance () {

        //      Adjust / Until (you find the payment...)

        int     month;
        int     iterations = 0;
        double  bal;
        double  monthly_rate = intrate / 1200;
        double  increment;
        double  Balance;

        if (trace == true)
                System.out.println ("In find_balance");

        //      Get into the "ballpark"

        Balance = months * payment;

        while (true) {
                ++iterations;
                bal = Balance;

                if (trace == true) {
                        System.out.println (
"Iteration " + iterations + "  Balance = " + Balance);
                }
```

```
                for (month = 1; month <= (months + 1); month++) {
                        bal -= payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }
                if (month > months)
                        break;

                Balance *= 2;
        }

        //      Adjust/Until balance

        increment = Balance;
        bal = Balance;

        while (bal > resolution || bal < -resolution) {
                ++iterations;
                bal = Balance;

                if (trace == true) {
                        System.out.println (
"Iteration " + iterations + "  Balance = " + Balance);
                }

                for (month = 1; month <= months + 1; month++) {
                        bal -= payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }

                increment /= 2;

                if (month < months)
                        Balance += increment;
                else if (month > months)
                        Balance -= increment;
                else    {
                        if (bal > resolution)
                                Balance -= increment;
                        else if (bal < 0.0)
                                Balance += increment;
                        else
                                break;
                }
        }
        if (trace == true)
                System.out.println ("Find_balance = " + Balance);
        balance = nearest_cent(Balance);
        return (balance);
 }

 double Find_intrate () {

        //      Adjust / Until (you find the interest rate...)
```

```
        int     month;
        int     iterations = 0;
        double  bal;
        double  monthly_rate;
        double  increment;
        double  Intrate;
        double  direction;

        if (trace == true)
                System.out.println ("In Find_intrate");

        //      Get into the "ballpark"
        //      First, determine direction of preliminary search:
        //              if loan doesn't pay off at zero interest, we're
        //              looking for negative interest !!

        Intrate = 0.0;
        monthly_rate = Intrate / 1200;
        bal = balance;

        for (month = 1; month <= (months + 1); month++) {
                bal -= payment - (bal * monthly_rate);
                if (bal <= 0.0)
                        break;
        }

        //      Can't pay off at 0% ?

        if (month > months)
                direction = -1.0;
        else
                direction = 1.0;

        Intrate = 0.5 * direction;

        while (true) {
                ++iterations;
                bal = balance;
                Intrate *= 2;
                monthly_rate = Intrate / 1200;

                if (trace == true) {
                        System.out.println (
"Iteration " + iterations + "  Intrate = " + Intrate);
                }

                for (month = 1; month <= months + 1; month++) {
                        bal -= payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }

                if (month > months)
                        break;
        }
```

```
        //      Adjust/Until intrate

        increment = Intrate;
        bal = balance;

        while (bal > resolution || bal < -resolution) {
                ++iterations;
                bal = balance;
                monthly_rate = Intrate / 1200;

                if (trace == true) {
                        System.out.println (
"Iteration " + iterations + "  Intrate = " + Intrate);
                }

                for (month = 1; month <= months + 1; month++) {
                        bal -= payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }

                increment /= 2;

                if (month < months)
                        Intrate += increment;
                else if (month > months)
                        Intrate -= increment;
                else    {
                        if (bal > resolution)
                                Intrate -= increment;
                        else if (bal < 0.0)
                                Intrate += increment;
                        else
                                break;
                }
        }
        if (trace == true)
                System.out.println ("Find_intrate = " + Intrate);
        intrate = nearest_cent(Intrate);
        return (intrate);
 }

 int Find_months () {

        //      Adjust / Until (you find the period...)

        int     month;
        int     iterations = 0;
        double  bal;
        double  monthly_rate = intrate / 1200;
        int     increment;
        int     Months;

        if (trace == true)
                System.out.println ("In Find_months");
```

```
        //      Get into the "ballpark"

        if (balance > payment)
                Months = 1;
        else
                return (1);

        while (true) {
                ++iterations;
                bal = balance;

                if (trace == true) {
                        System.out.println (
"Iteration " + iterations + "  Months = " + Months);
                }

                for (month = 1; month <= (Months + 1); month++) {
                        bal -= payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }
                if (month < Months)
                        break;

                Months *= 2;
        }

        increment = Months;
        bal = balance;

        while (true) {
                ++iterations;
                bal = balance;

                if (trace == true) {
                        System.out.println (
"Iteration " + iterations + "  Months = " + Months + "  increment = " + increment);
                }

                for (month = 1; month <= (Months + 1); month++) {
                        bal -= payment - (bal * monthly_rate);
                        if (bal <= 0.0)
                                break;
                }

                if (increment >= 2)
                        increment /= 2;

                if (month < Months)
                        Months -= increment;
                else if (month > Months)
                        Months += increment;
                else    {
                        if (bal < 0.0)
                                Months -= 1;
```

```
                        break;
                }
        }
        if (trace == true)
                System.out.println ("Find_months = " + Months);
        months = Months;
        return (Months);
}

double nearest_cent (double d) {
        long    L;

        L = (long)((d + 0.005) * 100);
        d = (double)(L);
        d /= 100;
        return (d);
}

//      format a float to specified precision
//      includes addition of ',' and '.'
// 123456.7890, n = 8, m = 2, yields:
//              "  123,456.78"
// returns a string longer than 'n' characters when d will not fit
//      in 'n' characters
/*
String double_align (double d, int n, int m) {
        String  S;
        int             columns = 3;
        int             x, y;
        int             commas, l;

        if (n < 1)
                n = 1;

        char newstring[] = new char[n];

        //      find number of significant characters to LEFT of '.'

        x = 1; y = 0;
        while ( d/x > 1.0) {
                x *= 10;
                y++;
        }

        if (y < n)
                y = n;

        //      find number of commas

        commas = y / 3;

        //      find total length of string

        l = y + commas + m + 1;  //  1 for '.'

        for (i = 0; i < (n - y); i++) {
```

```
                newstring[i] = ' ';
        }

        x /= 10;
  }
*/
}
```

LoanSave/LoanSave.java

```
import javax.swing.*;
import javax.help.*;
import javax.accessibility.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

import ControlPanel;
import TablePanel;

public class LoanSave extends JFrame {

   static boolean  trace = false;

   HelpSet                    helpset;
   URL                helpsetURL;
   HelpBroker         helpbroker;
   static final String    helpsetName = "LoanSaveHelpSet.hs";

   ControlPanel       controlpanel;
   TablePanel         tablepanel;

   Loan               L;
   Save               S;
   Both               B;

   Font               TitleFont, TextFont, ButtonFont;
   Color              Cred, Cgreen, Corange, Cwhite, Cgrey;

   private boolean    loan_grey;
   private boolean    save_grey;


   public static void main(String[] args) {
        try     {
          LoanSave frame = new LoanSave ();

          frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                   System.exit(0);
                }
          });
          frame.setTitle(new String("Loan Save"));
          frame.setSize (700, 1000);
          frame.setVisible(true);
          frame.setResizable(true);

          if (trace == true)
                System.out.println("AT END OF 'TRY' IN 'main()'");
        }
        catch (Exception e) {
          if (trace == true)
                System.out.println("IN 'CATCH' IN 'main()'");
```

```
            System.err.println(e);
            e.printStackTrace ();
        }
}

public LoanSave () {
        this.getAccessibleContext().setAccessibleName("LoanSave");
        this.getRootPane().getAccessibleContext().setAccessibleName("RootPane");

        this.getLayeredPane().getAccessibleContext()
            .setAccessibleName("LayeredPane");

        Container contentpane = getContentPane();

        //      Layout Manager

    GridBagLayout Layout = new GridBagLayout();
        GridBagConstraints C = new GridBagConstraints();

    contentpane.setLayout(Layout);

        //      Create Fonts

        TitleFont = new Font ("Serif", Font.BOLD, 20);
        TextFont = new Font ("Serif", Font.BOLD, 18);
        ButtonFont = new Font ("Serif", Font.BOLD, 18);

        //      Create Colors

        Cred = new Color(200,0,0);
        Cgreen = new Color(0,200,0);
        Corange = new Color(200,200,0);
        Cwhite = new Color (255,255,255);
        Cgrey = new Color (225,225,225);

        //      Help

        try     {
            helpsetURL = HelpSet.findHelpSet(null, helpsetName);
            helpset = new HelpSet(null, helpsetURL);
        }
        catch (Exception e) {
            System.out.println("ERROR, LoanHist: HelpSet \""
                                + helpsetName + "\" NOT FOUND");
            return;
        }

        helpbroker = helpset.createHelpBroker();

        //      Create Menu Bar and "HELP" Menu Item

        JMenuBar menubar = new JMenuBar();
        menubar.getAccessibleContext().setAccessibleName("MenuBar");
        setJMenuBar(menubar);

        JMenu helpmenu = new JMenu("Help");
```

```
menubar.add(helpmenu);

AAJMenuItem helpitem = new AAJMenuItem("Tutorial");
helpitem.addActionListener(new CSH.DisplayHelpFromSource(helpbroker));

helpmenu.add(helpitem);

//        Context-Sensitive Help

JRootPane rootpane = getRootPane();

helpbroker.enableHelp(helpmenu, "help menu", helpset);

//        Window-level help (on frame)

helpbroker.enableHelpKey(rootpane, "help", helpset);

//
//        Create Data Objects (Loan, Save, Both)

L = new Loan ();
S = new Save ();
B = new Both (this);

//        Create Control Panel Labels

JLabel loanlabel = new JLabel ("LOAN", JLabel.LEFT);
loanlabel.setForeground (Cred);
loanlabel.setFont (TitleFont);
C.weightx = 0.50;
C.weighty = 0.0;
C.gridwidth = 1;
C.gridx = 0;
C.gridy = 0;
Layout.setConstraints(loanlabel, C);
contentpane.add (loanlabel);
CSH.setHelpIDString(loanlabel, "Loan.intro");

JLabel savelabel = new JLabel ("SAVE", JLabel.LEFT);
savelabel.setForeground(Cgreen);
savelabel.setFont (TitleFont);
//        C.weightx = 0.75;
C.gridx = 1;
Layout.setConstraints(savelabel, C);
contentpane.add (savelabel);
CSH.setHelpIDString(savelabel, "Save.intro");

//        Create Control Panel

C.weightx = 1.0;
C.gridx = 0;
C.gridy = 1;
C.gridwidth = 2;
controlpanel = new ControlPanel (this);
Layout.setConstraints(controlpanel, C);
contentpane.add(controlpanel);
```

```
        //      Create Table Panel

        tablepanel = new TablePanel (this);
        C.weightx = 1.0;
        C.weighty = 1.0;
        C.gridy = 2;
        C.fill = GridBagConstraints.BOTH;
        Layout.setConstraints(tablepanel, C);
        contentpane.add(tablepanel);

        //      Initialize Table Backgrounds

        loan_grey = false;
        save_grey = false;
        Set_Loan_White();
        Set_Save_White();
    }

    //      Table backgrounds are greyed any time something happens which
    //      might make the display inconsistent. The table backgrounds are
    //      restored after action has occurred which re-rationalizes the
    //      display.

    public void Set_Loan_Grey () {
        tablepanel.LT.table.setBackground(Cgrey);
        tablepanel.BT.table.setBackground(Cgrey);
        controlpanel.LoanPanel.total_payments_field.setBackground(Cgrey);
        controlpanel.LoanPanel.net_cost_field.setBackground(Cgrey);
        loan_grey = true;
    }
    public void Set_Loan_White () {
        tablepanel.LT.table.setBackground(Cwhite);
        if (save_grey == false)
           tablepanel.BT.table.setBackground(Cwhite);
        controlpanel.LoanPanel.total_payments_field.setBackground(Cwhite);
        controlpanel.LoanPanel.net_cost_field.setBackground(Cwhite);
        loan_grey = false;
    }
    public void Set_Save_Grey () {
        tablepanel.ST.table.setBackground(Cgrey);
        tablepanel.BT.table.setBackground(Cgrey);
        controlpanel.SavePanel.final_balance_field.setBackground(Cgrey);
        controlpanel.SavePanel.CAGR_field.setBackground(Cgrey);
        save_grey = true;
    }
    public void Set_Save_White () {
        tablepanel.ST.table.setBackground(Cwhite);
        if (loan_grey == false)
           tablepanel.BT.table.setBackground(Cwhite);
        controlpanel.SavePanel.final_balance_field.setBackground(Cwhite);
        controlpanel.SavePanel.CAGR_field.setBackground(Cwhite);
        save_grey = false;
    }

}
```

```
// This class is basically JMenuItem class.  What is added is the
// action listener for writing script when the menuitem is selected.
class AAJMenuItem extends JMenuItem {

   AAJMenuItem (String name){
         super (name);

         //  write script when action triggered
         this.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent e){
                  String path;
                  path = getPath(getAccessibleContext());
                  saveScript(path, 0, "", "Tutorial HelpItemSelected");
            }});
   }

   // returns a full path from the top level swing component.
   // this is return string value is used to write script.
   String getPath(AccessibleContext thisComp){
         Accessible parent;
         String name;

         System.out.println(thisComp.getAccessibleName());

         parent = thisComp.getAccessibleParent();
         if (parent == null)  return "LoanSave";
         else {
            name = thisComp.getAccessibleName();
            if (name == null) name = "null";
            return (getPath(parent.getAccessibleContext())+"."+name);}
   }

   // write to the script file
   // Must not leave file open because the accessing component can open
   // the same file.

    void saveScript(String WidgetPath, int ActionNum,
                        String value, String comment){

         File f;
         String content = "";
         String entry = "";
      BufferedReader in;
         BufferedWriter out;

         try{
            f = new File("/u/youngkim/1999/LoanSave/script");


            // if the file already exists, it means we just need to
            // concatenate the new script to it.

            if (f.exists()){
                  in = new BufferedReader(new FileReader(f));
                  String line = new String();
```

```
                while ((line = in.readLine()) != null)
                    content = content.concat(line + "\n");
                in.close();
            }

            out = new BufferedWriter(new FileWriter(f));

            entry = content + WidgetPath + ":" + ActionNum + ":"
                + value + "//"+ comment + "\n";

            out.write(entry);
            out.close();
        }
        catch (Exception ioe) {
            System.out.println(ioe);
        }
    }
}
```

LoanSave/Save.java

```
public class Save {
 boolean        trace = false;

 //      Save's Data

 //      NOTE:  Save's Data is distinguished from the corresponding
 //              entries in the Control Panel

 //      USER'S INPUT DATA

 double balance = 1300000;
 double intrate = 7.0;
 int            months = 180;
 double deposit = 0.0;
 double taxrate = 32.0;
 double ending_balance;
 double CAGR;

 //      Derived Values and Constants

 int            years;
 double minimum_deposit;
 double resolution = 0.001;

 //      WORKING COPIES

 double w_balance;
 double w_intrate;
 int            w_months;
 double w_deposit;
 double w_taxrate;
 double w_ending_balance;
 double w_CAGR;
 int            w_years;

 //      "Run" Data

 double years_balance[] = new double[101];
 double years_interest[] = new double[101];
 double years_taxes_payed[] = new double[101];
 double years_net_deposit[] = new double[101];
 double cumulative_interest;
 double cumulative_net_interest;

 Save () {
        run_save ();
 }

 void run_save () {

        if (trace == true)
                System.out.println ("In Save.run_save ()");

        //      move object's data into working variables
```

```
  w_balance = balance;
  w_intrate = intrate;
  w_months = months;
  w_deposit = deposit;
  w_taxrate = taxrate;

        do_save (true);

        //  move do's results into object's variables

        ending_balance = w_ending_balance;
        CAGR = w_CAGR;
        years = w_years;
}

void test_save () {

        // "find" has set do's variables; just run it and return...

        do_save (false);

}

//      This function runs the save and (maybe) puts the summary into variables

private void do_save (boolean do_summary) {
        double  bal = w_balance;
        double  monthly_rate = w_intrate / 1200;
        double  daily_rate = w_intrate / 36500;
        double  years_int = 0.0;
        double  years_tax;
        double  months_int;
        double  days_int;
        int             day;
        int             monthdays;
        int             month;
        int             year = 1;

        if (do_summary == true) {
                if (trace == true)
                        System.out.println ("In Save.do_save (true)");

                cumulative_interest = 0.0;
                cumulative_net_interest = 0.0;
                years_balance[0] = w_balance;
        }

        for (month = 1; month <= w_months; month++) {

                //      Find days / month

                switch (month % 12) {
                 case 2:
                        //      leap year ?
```

```
                if (year % 4 == 0)
                        monthdays = 29;
                else
                        monthdays = 28;
                break;
        case 4:
        case 6:
        case 9:
        case 11:
                monthdays = 30;
                break;
        default:
                monthdays = 31;
                break;
        }

        //      Do a month's saving using daily interest compounding

        for (day = 1; day <= monthdays; day++) {
                days_int = bal * daily_rate;
                years_int += days_int;
                bal += days_int;
        }

        //      Do month's deposit

        bal += w_deposit;

        //      Do End-Of-Year ?

        if ((month % 12 == 0) || (month == w_months)) {

                //      Pay year's tax out of balance

                years_tax = years_int * (w_taxrate / 100);
                bal -= years_tax;

                //      Record year's summary

                year = month / 12;
                if ((month % 12) != 0)
                        ++year;

                if (do_summary == true) {
                        years_balance[year] = nearest_cent (bal);
                        years_interest[year] = nearest_cent(years_int);
                        years_taxes_payed[year] = nearest_cent (years_tax);
                        years_net_deposit[year] = nearest_cent((w_deposit * 12) - years_tax);
                        cumulative_interest += years_int;
                        cumulative_net_interest += years_int - years_tax;
                }
                years_int = 0.0;
                if (trace == true) {
System.out.println ("Year " + year + "   Balance = " + bal);
                }
        }
```

```
        }
        w_ending_balance = nearest_cent(bal);
        w_CAGR = nearest_cent(CAGR_Func (w_balance, bal, w_months));
        w_years = year;
 }

/*
 *              FIND (Goal Seeking) DISCUSSION
 *
 *      When working with a loan, as implemented in "Loan.java", the
 *      target value when finding balance, interest rate, months, or
 *      payment is understood to be "Balance == 0.0"
 *
 *      When working with a save, the target value must be supplied by
 *      the user.  In this implementation, the target value can be specified
 *      as either "final balance" or "CAGR", with "final balance" the default.
 */

 //             FIND DEPOSIT

 double Find_Deposit (int which_target) {
  double        d;

        switch (which_target) {
                case 2:
                        d = Find_Deposit_Target_CAGR ();
                        break;
                case 1:
                default:
                        d = Find_Deposit_Target_BALANCE ();
                        break;
        }

        //      Wrap-Up

  w_deposit = deposit = nearest_cent(d);
        if (trace == true)
                System.out.println ("Find_Deposit = " + w_deposit);
        return (w_deposit);
 }

 //     Find DEPOSIT targeting ENDING BALANCE

 private double Find_Deposit_Target_BALANCE () {

  //    Set Working Variables (except deposit)

        w_intrate = intrate;
        w_taxrate = taxrate;
        w_months = months;

        //      Variables and Constants

        int             iterations = 0;
        double  increment;
        int             max_iterations;
```

```
        if (trace == true)
                System.out.println ("In Find_Deposit_Target_BALANCE ()");

        //      Find starting value for deposit

        w_deposit = 0.0;
        w_balance = balance;
        test_save ();
        max_iterations = 1;

        if (w_ending_balance < ending_balance) {
                w_deposit = resolution;
                while (w_ending_balance < ending_balance) {
                        w_balance = balance;
                        test_save ();
                        w_deposit *= 2;
                        ++max_iterations;
                }
        }
        else if (w_ending_balance > ending_balance) {
                w_deposit = -resolution;
                while (w_ending_balance > ending_balance) {
                        w_balance = balance;
                        test_save ();
                        w_deposit *= 2;
                        ++max_iterations;
                }
        }

        increment = w_deposit;

        //      Vary deposit, sense ending balance

        while (iterations == 0 ||
            (w_ending_balance > (ending_balance + resolution)) ||
            (w_ending_balance < (ending_balance - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  deposit = " + nearest_cent(w_deposit) + " balance = " +
nearest_cent(w_ending_balance));

                if (++iterations > max_iterations)
                        break;

                //      Set "starting" balance

                w_balance = balance;      //      "balance" is conceptually the "starting balance"

                //      Run the Save

                test_save ();

                //      Calculate next deposit increment
```

```
                increment /= 2;

                if (increment >= 0.0) {
                        if (increment < resolution / 2) {
                                if (trace == true) {
                                        System.out.println (
"increment < resolution / 2, increment = " + increment);
                                }
                                break;
                        }
                }
                else if (increment > -(resolution / 2)) {
                        if (trace == true) {
                                System.out.println (
"increment > -(resolution / 2), increment = " + increment);
                        }
                        break;
                }

                //      Calculate new deposit value

                if (increment >= 0.0) {
                        if (w_ending_balance > (ending_balance + resolution))
                                w_deposit -= increment;
                        else if (w_ending_balance < (ending_balance - resolution))
                                w_deposit += increment;
                        else
                                break;
                }
                else    {
                        if (w_ending_balance > (ending_balance + resolution))
                                w_deposit += increment;
                        else if (w_ending_balance < (ending_balance - resolution))
                                w_deposit -= increment;
                        else
                                break;
                }
  }
  return (w_deposit);
 }

 //     Find DEPOSIT targeting CAGR (Compound Annual Growth Rate)

private double Find_Deposit_Target_CAGR () {

  //    Set Working Variables (except deposit)

        w_intrate = intrate;
        w_taxrate = taxrate;
        w_months = months;

        //      Variables and Constants

        int             iterations = 0;
        double  increment;
        int             max_iterations;
```

```
        if (trace == true)
                System.out.println ("In Find_Deposit_Target_CAGR ()");

        //      Find a "starting" deposit value.

        w_deposit = 0.0;
        w_balance = balance;
        test_save ();
        max_iterations = 1;

        if (w_CAGR < CAGR) {
                w_deposit = resolution;
                while (w_CAGR < CAGR) {
                        w_balance = balance;
                        test_save ();
                        w_deposit *= 2;
                        ++max_iterations;
                }
                increment = w_deposit;
        }
        else if (w_CAGR > CAGR) {
                w_deposit = -resolution;
                while (w_CAGR > CAGR) {
                        w_balance = balance;
                        test_save ();
                        w_deposit *= 2;
                        ++max_iterations;
                }
                increment = w_deposit;
        }
        //      else:  w_CAGR == CAGR and w_deposit == 0.0  !!
        else    {
                increment = 0.0; //          shut up javac...
                iterations = 1;
        }

        if (trace == true)
                System.out.println ("w_deposit = " + nearest_cent (w_deposit));

        //      Vary deposit, Sense CAGR

        while (iterations == 0 ||
            (w_CAGR > (CAGR + resolution)) ||
                (w_CAGR < (CAGR - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  deposit = " + nearest_cent(w_deposit) + " CAGR = " +
nearest_cent(w_CAGR));

                if (++iterations > max_iterations)
                        break;

                //      Set "starting" balance
```

```
                w_balance = balance;      //      "balance" is conceptually the "starting balance"

                //      Run the Save

                test_save ();

                //      Calculate next deposit increment

                increment /= 2;

                if (increment >= 0.0) {
                        if (increment < resolution/2) {
                                if (trace == true) {
                                        System.out.println (
"increment < resolution/2, increment = " + increment);
                                }
                                break;
                        }
                }
                else if (increment > -(resolution / 2)) {
                        if (trace == true) {
                                System.out.println (
"increment > -(resolution/2), increment = " + increment);
                        }
                        break;
                }

                //      Calculate new deposit value

                if (increment >= 0.0) {
                        if (w_CAGR > (CAGR + resolution))
                                w_deposit -= increment;
                        else if (w_CAGR < (CAGR - resolution))
                                w_deposit += increment;
                        else
                                break;
                }
                else    {
                        if (w_CAGR > (CAGR + resolution))
                                w_deposit += increment;
                        else if (w_CAGR < (CAGR - resolution))
                                w_deposit -= increment;
                        else
                                break;
                }
 }
 return (w_deposit);
}

//      FIND BALANCE

double Find_Balance (int which_target) {
  double        d;

        switch (which_target) {
                case 2:
```

```
                        d = Find_Balance_Target_CAGR ();
                        break;
                case 1:
                default:
                        d = Find_Balance_Target_BALANCE ();
                        break;
        }

        //      Wrap-Up

 w_balance = balance = nearest_cent(d);
        if (trace == true)
                System.out.println ("Find_Balance = " + w_balance);
        return (w_balance);
}

//      Find STARTING BALANCE targeting ENDING BALANCE

private double Find_Balance_Target_BALANCE () {

 //     Set Working Variables (except balance)

        w_deposit = deposit;
        w_intrate = intrate;
        w_taxrate = taxrate;
        w_months = months;

        //      Variables and Constants

        int             month;
        int             iterations = 0;
        double  monthly_rate = w_intrate / 1200;
        double  increment;
        double  max_val;
        int             max_iterations;

        if (trace == true)
                System.out.println ("In Find_Balance_Target_BALANCE ()");

        //      Find initial value for starting balance

        w_balance = 0.0;
        test_save ();
        max_iterations = 1;

        if (w_ending_balance < ending_balance) {
                w_balance = resolution;
                while (w_ending_balance < ending_balance) {
                        test_save ();
                        w_balance *= 2;
                        ++max_iterations;
                }
        }
        else if (w_ending_balance > ending_balance) {
                w_balance = -resolution;
                while (w_ending_balance > ending_balance) {
```

```
                    test_save ();
                    w_balance *= 2;
                    ++max_iterations;
                }
        }

        increment = w_balance;

        //      Vary starting balance, sense ending balance

        while (iterations == 0 ||
            (w_ending_balance > (ending_balance + resolution)) ||
            (w_ending_balance < (ending_balance - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  starting balance = " + nearest_cent(w_balance) + " final balance = " +
nearest_cent(w_ending_balance));

                if (++iterations > max_iterations)
                        break;

                //      Run the Save

                test_save ();

                //      Calculate next balance increment

                increment /= 2;

                if (increment >= 0.0) {
                        if (increment < resolution / 2) {
                                if (trace == true) {
                                        System.out.println (
"increment < resolution / 2, increment = " + increment);
                                }
                                break;
                        }
                }
                else if (increment > -(resolution / 2)) {
                        if (trace == true) {
                                System.out.println (
"increment > -(resolution / 2), increment = " + increment);
                        }
                        break;
                }

                //      Calculate new balance value

                if (increment >= 0.0) {
                        if (w_ending_balance > (ending_balance + resolution))
                                w_balance -= increment;
                        else if (w_ending_balance < (ending_balance - resolution))
                                w_balance += increment;
                        else
                                break;
```

```
                }
                else    {
                        if (w_ending_balance > (ending_balance + resolution))
                                w_balance += increment;
                        else if (w_ending_balance < (ending_balance - resolution))
                                w_balance -= increment;
                        else
                                break;
                }
 }
 return (w_balance);
}

 //     Find STARTING BALANCE targeting CAGR (Compound Annual Growth Rate)

private double Find_Balance_Target_CAGR () {

 //     Set Working Variables (except balance)

        w_deposit = deposit;
        w_intrate = intrate;
        w_taxrate = taxrate;
        w_months = months;

        //      Variables and Constants

        int             iterations = 0;
        double  increment;
        int             max_iterations;

        if (trace == true)
                System.out.println ("In Find_Balance_Target_CAGR ()");

        //      Find a "starting" balance value.

        w_balance = 0.0;
        test_save ();
        max_iterations = 1;

        if (w_CAGR < CAGR) {
                w_balance = resolution;
                while (w_CAGR < CAGR) {
                        test_save ();
                        w_balance *= 2;
                        ++max_iterations;
                }
                increment = w_balance;
        }
        else if (w_CAGR > CAGR) {
                w_balance = -resolution;
                while (w_CAGR > CAGR) {
                        test_save ();
                        w_balance *= 2;
                        ++max_iterations;
                }
                increment = w_balance;
```

```
        }

        //      else:  w_CAGR == CAGR and w_balance == 0.0  !!

        else
                return (0.0);

        if (trace == true) {
                System.out.println (
"at start, w_balance = " + nearest_cent (w_balance));
        }

        //      Vary starting balance, Sense CAGR

        while (iterations == 0 ||
            (w_CAGR > (CAGR + resolution)) ||
                (w_CAGR < (CAGR - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  balance = " + nearest_cent(w_balance) + " CAGR = " +
nearest_cent(w_CAGR));

                if (++iterations > max_iterations)
                        break;

                //      Run the Save

                test_save ();

                //      Calculate next balance increment

                increment /= 2;

                if (increment >= 0.0) {
                        if (increment < resolution/2) {
                                if (trace == true) {
                                        System.out.println (
"increment < resolution/2, increment = " + increment);
                                }
                                break;
                        }
                }
                else if (increment > -(resolution / 2)) {
                        if (trace == true) {
                                System.out.println (
"increment > -(resolution/2), increment = " + increment);
                        }
                        break;
                }

                //      Calculate new deposit value

                if (increment >= 0.0) {
                        if (w_CAGR > (CAGR + resolution))
                                w_balance -= increment;
```

```
                        else if (w_CAGR < (CAGR - resolution))
                                w_balance += increment;
                        else
                                break;
                }
                else    {
                        if (w_CAGR > (CAGR + resolution))
                                w_balance += increment;
                        else if (w_CAGR < (CAGR - resolution))
                                w_balance -= increment;
                        else
                                break;
                }
 }
 return (w_balance);
}

//      FIND INTEREST RATE

double Find_Intrate (int which_target) {
 double         d;

        switch (which_target) {
                case 2:
                        d = Find_Intrate_Target_CAGR ();
                        break;
                case 1:
                default:
                        d = Find_Intrate_Target_BALANCE ();
                        break;
        }

        //      Wrap-Up

 w_intrate = intrate = nearest_cent(d);
        if (trace == true)
                System.out.println ("Find_Intrate = " + w_intrate);
        return (w_intrate);
}

//      Find INTEREST RATE targeting ENDING BALANCE

private double Find_Intrate_Target_BALANCE () {

 //     Set Working Variables (except intrate)

        w_deposit = deposit;
        w_taxrate = taxrate;
        w_months = months;

        //      Variables and Constants

        int             iterations = 0;
        double  increment;
        int             max_iterations;
```

```
        if (trace == true)
                System.out.println ("In Find_Deposit_Target_BALANCE ()");

        //      Find starting value for intrate

        w_intrate = 0.0;
        w_balance = balance;
        test_save ();
        max_iterations = 1;

        if (w_ending_balance < ending_balance) {
                w_intrate = resolution;
                while (w_ending_balance < ending_balance) {
                        w_balance = balance;
                        test_save ();
                        w_intrate *= 2;
                        ++max_iterations;
                }
        }
        else if (w_ending_balance > ending_balance) {
                w_intrate = -resolution;
                while (w_ending_balance > ending_balance) {
                        w_balance = balance;
                        test_save ();
                        w_intrate *= 2;
                        ++max_iterations;
                }
        }

        increment = w_intrate;

        //      Vary intrate, sense ending balance

        while (iterations == 0 ||
            (w_ending_balance > (ending_balance + resolution)) ||
            (w_ending_balance < (ending_balance - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  Interest Rate = " + nearest_cent(w_intrate) + " balance = " +
nearest_cent(w_ending_balance));

                if (++iterations > max_iterations)
                        break;

                //      Set "starting" balance

                w_balance = balance;    //      "balance" is conceptually the "starting balance"

                //      Run the Save

                test_save ();

                //      Calculate next deposit increment

                increment /= 2;
```

```
                if (increment >= 0.0) {
                        if (increment < resolution / 2) {
                                if (trace == true) {
                                        System.out.println (
"increment < resolution / 2, increment = " + increment);
                                }
                                break;
                        }
                }
                else if (increment > -(resolution / 2)) {
                        if (trace == true) {
                                System.out.println (
"increment > -(resolution / 2), increment = " + increment);
                        }
                        break;
                }

                //      Calculate new intrate value

                if (increment >= 0.0) {
                        if (w_ending_balance > (ending_balance + resolution))
                                w_intrate -= increment;
                        else if (w_ending_balance < (ending_balance - resolution))
                                w_intrate += increment;
                        else
                                break;
                }
                else    {
                        if (w_ending_balance > (ending_balance + resolution))
                                w_intrate += increment;
                        else if (w_ending_balance < (ending_balance - resolution))
                                w_intrate -= increment;
                        else
                                break;
                }
 }
 return (w_intrate);
}

 //     Find INTEREST RATE targeting CAGR (Compound Annual Growth Rate)

private double Find_Intrate_Target_CAGR () {

  //    Set Working Variables (except intrate)

        w_deposit = deposit;
        w_taxrate = taxrate;
        w_months = months;

        //      Variables and Constants

        int             iterations = 0;
        double  increment;
        int             max_iterations;
```

```
        if (trace == true)
                System.out.println ("In Find_Intrate_Target_CAGR ()");

        //      Find a "starting" interest rate value.

        w_balance = balance;
        test_save ();
        max_iterations = 1;

        if (w_CAGR < CAGR) {
                w_intrate = resolution;
                while (w_CAGR < CAGR) {
                        w_balance = balance;
                        test_save ();
                        w_intrate *= 2;
                        ++max_iterations;
                }
                increment = w_intrate;
        }
        else if (w_CAGR > CAGR) {
                w_intrate = -resolution;
                while (w_CAGR > CAGR) {
                        w_balance = balance;
                        test_save ();
                        w_intrate *= 2;
                        ++max_iterations;
                }
                increment = w_intrate;
        }
        //      else:  w_CAGR == CAGR and w_deposit == 0.0  !!
        else    {
                increment = 0.0; //          shut up javac...
                iterations = 1;
        }

        if (trace == true) {
                System.out.println (
"at start, w_intrate = " + nearest_cent (w_intrate));
        }

        //      Vary intrate, Sense CAGR

        while (iterations == 0 ||
            (w_CAGR > (CAGR + resolution)) ||
                (w_CAGR < (CAGR - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  Interest Rate = " + nearest_cent(w_intrate) + " CAGR = " +
nearest_cent(w_CAGR));

                if (++iterations > max_iterations)
                        break;

                //      Set "starting" balance
```

```
                w_balance = balance;      //      "balance" is conceptually the "starting balance"

                //      Run the Save

                test_save ();

                //      Calculate next deposit increment

                increment /= 2;

                if (increment >= 0.0) {
                        if (increment < resolution/2) {
                                if (trace == true) {
                                        System.out.println (
"increment < resolution/2, increment = " + increment);
                                }
                                break;
                        }
                }
                else if (increment > -(resolution / 2)) {
                        if (trace == true) {
                                System.out.println (
"increment > -(resolution/2), increment = " + increment);
                        }
                        break;
                }

                //      Calculate new intrate value

                if (increment >= 0.0) {
                        if (w_CAGR > (CAGR + resolution))
                                w_intrate -= increment;
                        else if (w_CAGR < (CAGR - resolution))
                                w_intrate += increment;
                        else
                                break;
                }
                else    {
                        if (w_CAGR > (CAGR + resolution))
                                w_intrate += increment;
                        else if (w_CAGR < (CAGR - resolution))
                                w_intrate -= increment;
                        else
                                break;
                }
        }
        return (w_intrate);
        }

        //      FIND MONTHS

        int Find_Months (int which_target) {
                int     i;

                switch (which_target) {
                   case 2:
```

```
                i = Find_Months_Target_CAGR ();
                break;
            case 1:
            default:
                i = Find_Months_Target_BALANCE ();
                break;
        }

        //      Wrap-Up

        months = w_months;
        if (trace == true)
                System.out.println ("Find_Months = " + w_months);
        return (w_months);
}

//      Find MONTHS targeting ENDING BALANCE

private int Find_Months_Target_BALANCE () {

 //     Set Working Variables (except months)

        w_deposit = deposit;
        w_intrate = intrate;
        w_taxrate = taxrate;

        //      Variables and Constants

        int             iterations = 0;
        int             increment;
        int             max_iterations;

        if (trace == true)
                System.out.println ("In Find_Months_Target_BALANCE ()");

        //      Find starting value for months

        w_months = 1;
        w_balance = balance;
        test_save ();
        max_iterations = 1;

        if (w_ending_balance < ending_balance) {
                while (w_ending_balance < ending_balance) {
                        w_balance = balance;
                        test_save ();
                        w_months *= 2;
                        ++max_iterations;
                }
        }
        else
                return (1);

        increment = w_months;

        //      Vary months, sense ending balance
```

```
        while (iterations == 0 ||
            (w_ending_balance > (ending_balance + resolution)) ||
            (w_ending_balance < (ending_balance - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  months = " + w_months + " balance = " + nearest_cent(w_ending_balance));

                if (++iterations > max_iterations)
                        break;

                //      Set "starting" balance

                w_balance = balance;    //      "balance" is conceptually the "starting balance"

                //      Run the Save

                test_save ();

                //      Calculate next deposit increment

                increment /= 2;

                if (increment == 0)
                        break;

                //      Calculate new months value

                if (w_ending_balance > (ending_balance + resolution))
                        w_months -= increment;
                else if (w_ending_balance < (ending_balance - resolution))
                        w_months += increment;
                else
                        break;
 }
 return (w_months);
}

 //     Find MONTHS targeting CAGR (Compound Annual Growth Rate)

private int Find_Months_Target_CAGR () {

 //     Set Working Variables (except months)

        w_deposit = deposit;
        w_intrate = intrate;
        w_taxrate = taxrate;

        //      Variables and Constants

        int             iterations = 0;
        int             increment;
        int             max_iterations;

        if (trace == true)
```

```
            System.out.println ("In Find_Months_Target_CAGR ()");

        //      Find a "starting" months value.

        w_balance = balance;
        test_save ();
        max_iterations = 1;

        if (w_CAGR < CAGR) {
                w_months = 1;
                while (w_CAGR < CAGR) {
                        w_balance = balance;
                        test_save ();
                        w_months *= 2;
                        ++max_iterations;
                }
                increment = w_months;
        }

        //      else:  w_CAGR == CAGR and w_months == 1  !!

        else
                return (1);

        if (trace == true)
                System.out.println ("w_months = " + w_months);

        //      Vary w_months, Sense CAGR

        while (iterations == 0 ||
             (w_CAGR > (CAGR + resolution)) ||
                  (w_CAGR < (CAGR - resolution))) {

                if (trace == true)
                        System.out.println (
"Iteration " + iterations + "  months = " + w_months + " CAGR = " + nearest_cent(w_CAGR));

                if (++iterations > max_iterations)
                        break;

                //      Set "starting" balance

                w_balance = balance;      //      "balance" is conceptually the "starting balance"

                //      Run the Save

                test_save ();

                //      Calculate next deposit increment

                increment /= 2;

                if (increment == 0)
                        break;

                //      Calculate new deposit value
```

```
                if (w_CAGR > (CAGR + resolution))
                        w_months -= increment;
                else if (w_CAGR < (CAGR - resolution))
                        w_months += increment;
                else
                        break;
 }
 return (w_months);
}

private double CAGR_Func (
 double         start_bal,
 double         end_bal,
 int    months) {

 double         CAGR_Func;
 double         percentage;
 double         lastpercentage;
 double         increment;
 double         balance;
 double         res;
 int    month;
 int    isneg;
 int    iterations;

 iterations = 0;

 // Error Checks

 if (months < 2) {
//      MsgBox "ERROR, CAGR_Func:  months < 2 ILLEGAL
 }

 if (start_bal == end_bal) {
 }

 // Always run positive

 if (end_bal > start_bal) {
    isneg = 0;
 }
 else {
    isneg = 1;
    double      d1;
    d1 = start_bal;
    start_bal = end_bal;
    end_bal = d1;
 }

 if (start_bal == 0) {
    start_bal = start_bal + 1;
    end_bal = end_bal + 1;
 }

 // start at 100% per month
```

```
percentage = 1200;
lastpercentage = 2400;
increment = 1200;
res = 0.01;

do {
   iterations = iterations + 1;
   balance = start_bal;
   for (month = 1; month <= months; month++) {
      balance = balance + ((percentage / 1200) * balance);
      if (balance > end_bal) {
         break;
      }
   }

   //  done ?

   if (balance == end_bal) {
      break;
   }

   //  done due to resolution ?

   if (lastpercentage > percentage) {
      if (lastpercentage - percentage < res) {
         break;
      }
         }
   else {
      if (percentage - lastpercentage < res) {
         break;
      }
   }

   //  calculate next value

   lastpercentage = percentage;
   increment = increment * 0.5;

   if (balance > end_bal)
      percentage = percentage - increment;
   else
      percentage = percentage + increment;

} while (true);

//  set return value

if (isneg != 0)
   CAGR_Func = percentage * -1;
else
   CAGR_Func = percentage;

return (CAGR_Func);
}
```

```
 double nearest_cent (double d) {
        long    L;

        L = (long)((d + 0.005) * 100);
        d = (double)(L);
        d /= 100;
        return (d);
 }

 //     format a float to specified precision
 //     includes addition of ',' and '.'
 // 123456.7890, n = 8, m = 2, yields:
 //             "  123,456.78"
 // returns a string longer than 'n' characters when d will not fit
 //     in 'n' characters
/*
 String double_align (double d, int n, int m) {
        String  S;
        int             columns = 3;
        int             x, y;
        int             commas, l;

        if (n < 1)
                n = 1;

        char newstring[] = new char[n];

        //      find number of significant characters to LEFT of '.'

        x = 1; y = 0;
        while ( d/x > 1.0) {
                x *= 10;
                y++;
        }

        if (y < n)
                y = n;

        //      find number of commas

        commas = y / 3;

        //      find total length of string

        l = y + commas + m + 1;  //  1 for '.'

        for (i = 0; i < (n - y); i++) {
                newstring[i] = ' ';
        }

        x /= 10;
 }
*/
}
```

LoanSave/TablePanel.java

```
import javax.swing.*;
import javax.swing.table.*;
import javax.swing.event.*;
import javax.help.*;
import javax.accessibility.*;

import java.awt.*;
import java.awt.Font.*;
import java.awt.event.*;
import java.io.*;
import java.lang.*;
import java.net.*;
import java.util.*;

public class TablePanel extends JPanel
   implements TableCellRenderer {

   LoanSave       parent;
   boolean        trace = false;

   // An event stack, NOT a boolean !!
   int            program_changing_Loan_text = 0;
   int            program_changing_Save_text = 0; // ditto

   LoanTable      LT;
   SaveTable      ST;
   BothTable      BT;

   Loan           L;
   Save           S;
   Both           B;

   GridLayout     Layout;
   JLabel         renderer = new JLabel("");

   //     Constructor

   public TablePanel(LoanSave target) {

         // Always set the accessible name.
         this.getAccessibleContext().setAccessibleName("TablePanel");

         //        Pointers

         parent = target;
         L = parent.L;
         S = parent.S;
         B = parent.B;

         //        Layout

         GridBagLayout Layout = new GridBagLayout ();
         setLayout(Layout);
         GridBagConstraints C = new GridBagConstraints();
```

```
renderer = new JLabel("");
renderer.setHorizontalAlignment(SwingConstants.RIGHT);

//        Create The Display Tables

LT = new LoanTable (this);
ST = new SaveTable (this);
BT = new BothTable (this);

//        Update the tables

LT.update ();
ST.update ();
BT.update ();

//        Create The Labels

JLabel   LoanL, SaveL, BothL;

LoanL = new JLabel ("LOAN", JLabel.CENTER);
LoanL.setForeground (parent.Cred);
LoanL.setBackground (parent.Cwhite);
LoanL.setFont (parent.TitleFont);

SaveL = new JLabel ("SAVE", JLabel.CENTER);
SaveL.setForeground (parent.Cgreen);
SaveL.setBackground (parent.Cwhite);
SaveL.setFont (parent.TitleFont);

BothL = new JLabel ("BOTH", JLabel.CENTER);
BothL.setForeground (parent.Corange);
BothL.setBackground (parent.Cwhite);
BothL.setFont (parent.TitleFont);

//        Put It All On The Screen

C.weightx = 1.0;
C.weighty = 0.0;
C.gridx = 0;
C.gridy = 0;
C.fill = GridBagConstraints.NONE;
Layout.setConstraints(LoanL, C);
add (LoanL);

++C.gridy;
C.fill = GridBagConstraints.BOTH;
C.weighty = 0.33;
Layout.setConstraints(LT, C);
add (LT);

++C.gridy;
C.fill = GridBagConstraints.NONE;
C.weighty = 0.0;
Layout.setConstraints(SaveL, C);
add (SaveL);
```

```
        ++C.gridy;
        C.fill = GridBagConstraints.BOTH;
        C.weighty = 0.33;
        Layout.setConstraints(ST, C);
        add (ST);

        ++C.gridy;
        C.fill = GridBagConstraints.NONE;
        C.weighty = 0.0;
        Layout.setConstraints(BothL, C);
        add (BothL);

        ++C.gridy;
        C.weighty = 0.33;
        C.fill = GridBagConstraints.BOTH;
        Layout.setConstraints(BT, C);
        add (BT);
    }
    double nearest_cent (double d) {
        long    L;

        L = (long)((d + 0.005) * 100);
        d = (double)(L);
        d /= 100;
        return (d);
    }
    public Component getTableCellRendererComponent (
                                                    JTable          table,
                                                    Object          value,
                                                    boolean         isSelected,
                                                    boolean hasFocus,
                                                    int     row,
                                                    int     column) {

        if (value != null)
            renderer.setText((String)value.toString());

        if (isSelected == true)
            renderer.setBackground(parent.Cgrey);
        else
            renderer.setBackground(parent.Cgrey);
        if (hasFocus == true)
            renderer.setBackground(parent.Cgrey);
        else
            renderer.setBackground(parent.Cgrey);

        return renderer;
    }
}

class LoanTable extends JPanel {
    AAJTable            table;
    JScrollPane     scrollPane;
    LoanTableModel      Model = new LoanTableModel (this);
    TablePanel          parent;
```

```
Loan            L;
boolean         trace;
int             highest_row_used = 0;

// this boolean is for the model to know when to write script.
// Must not write script when RUN button is pressed.
static boolean RunPressed = false;

//      constructor

LoanTable (TablePanel target) {

    // Always set the accessible name.
    getAccessibleContext().setAccessibleName("LoanTable");

    //      save ("pointer" to) parent object

    parent = target;
    trace = parent.trace;

    L = parent.L;

    //      Create and Init Data Model

    Model = new LoanTableModel (this);
    Model.data = new Object[100][5];

    Initialize_Model ();

    //      create panel

    setLayout(new GridLayout(1,1,5,5));

    //      build table

    table = new AAJTable(Model);

    //    set the table name
    table.getAccessibleContext().setAccessibleName("Table");

    table.setPreferredScrollableViewportSize(new Dimension(450, 50));
    table.getColumnModel().getColumn(0).setPreferredWidth(1);
    table.setDefaultRenderer(Currency.class, parent);
    CSH.setHelpIDString(table, "Loan.table");

    //      Create the scroll pane; insert table

    scrollPane = new JScrollPane(table);
    scrollPane.getAccessibleContext().setAccessibleName("ScrollPane");

    scrollPane.getViewport().getAccessibleContext()
       .setAccessibleName("ViewPort");

    //      Put the scroll pane on screen

    add(scrollPane);
```

```
}

private void Initialize_Model () {
        int     i, j;

        for (i = 0; i < 100; i++) {
           for (j = 0; j < 5; j++) {
                Model.data[i][j] = new Double (0.0);
           }
        }
}

//      Move new table data FROM Loan object TO Loantable object

void update () {
        int     year;
        int     row;
        int     col;
        int     mincols;
        double  d;
        boolean pd = false;     //      print '$'
        int     fracs = 2;      //      fraction digits to print


        // do not write script for cell changes now
        RunPressed = true;

        mincols = Currency.get_minColumns(L.years_balance[1]);
        mincols += 3;    // for '.12', the CENTS

        Initialize_Model ();

        for (year = 1; year <= L.years; year++) {
           row = year - 1;
           col = 0;

           //      Year

           table.setValueAt (new Double (year), row, col);

           //      Balance

           table.setValueAt (new Currency (L.years_balance[year],
                                           mincols, fracs, pd), row, ++col);

           //      Change

           d = L.years_balance[year] - L.years_balance[year - 1];
           table.setValueAt (new Currency (d, mincols, fracs, pd), row, ++col);

           //      Taxes Saved

           table.setValueAt (new Currency (L.years_taxes_saved[year],
                                           mincols, fracs, pd), row, ++col);

           //      Net Payment
```

```
            table.setValueAt (new Currency (L.years_net_payment[year],
                                            mincols, fracs, pd), row, ++col);

        }
        for ( ; year < highest_row_used; year++) {
            row = year - 1;
            col = 0;

            //      Year

            table.setValueAt (new Double (0.0), row,   col);

            //      Balance

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);

            //      Change

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);

            //      Taxes Saved

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);

            //      Net Payment

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);
        }
        highest_row_used = L.years;
        Model.fireTableDataChanged();

        // write script for cell changes now
        RunPressed = false;
    }
    public void restore_background () {
        table.setBackground (parent.parent.Cwhite);
    }

}

class LoanTableModel extends AbstractTableModel {

    final String[] columnNames = {
        "Year",
        "Loan Balance",
        "Change",
        "Taxes Saved",
        "Net Payment",
    } ;

    Object data[][];

    LoanTable parent;

    LoanTableModel (LoanTable target){
```

```
        parent = target;
    }

    public int getColumnCount() {
        return columnNames.length;
    }

    public int getRowCount() {
        return (100);
    }

    public String getColumnName(int col) {
        return columnNames[col];
    }

    public Object getValueAt(int row, int col) {
        return data[row][col];
    }

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }

    //      implement this method if table is editable

    public boolean isCellEditable(int row, int col) {
        // Note that the data/cell address is constant,
        // no matter where the cell appears onscreen.

        if (row < 0 || row > 100)
           return (false);
        if (col <= 0 || col > 4)
           return (false);
        else
           return (true);
    }

    //      implement this method if table data can change

    public void setValueAt(Object value, int row, int col) {
        data[row][col] = value;

        // CHANGE IN A CELL GETS THIS FUNCTION TO BE CALLED
        // Write script only when the RUN button is NOT pressed.

        //System.out.println("cell value changed");

        if (parent.RunPressed == false)
           this.fireTableCellUpdated(row, col);
    }

}

class SaveTable extends JPanel{
    AAJTable            table;
    JScrollPane         scrollPane;
```

```
SaveTableModel      Model = new SaveTableModel (this);
TablePanel          parent;
Save                S;
boolean             trace;
int                 highest_row_used = 0;

// this boolean is for the model to know when to write script
// Must not write script when RUN button is pressed.
static boolean RunPressed = false;


//      constructor

SaveTable (TablePanel target) {

        // Always set the accessible name.
        getAccessibleContext().setAccessibleName("SaveTable");

        //          save ("pointer" to) parent object

        parent = target;
        trace = parent.trace;
        if (trace == true)
           System.out.println ("In SaveTable () (constructor)");

        S = parent.S;
        Model.data = new Object[100][5];

        Initialize_Model ();

        //          create panel

        setLayout(new GridLayout(1,1,5,5));

        //          build table

        table = new AAJTable(Model);
        table.getAccessibleContext().setAccessibleName("Table");
        table.setPreferredScrollableViewportSize(new Dimension(450, 50));
        table.getColumnModel().getColumn(0).setPreferredWidth(1);
        table.setDefaultRenderer(Currency.class, parent);
        CSH.setHelpIDString(table, "Save.table");

        //          Create the scroll pane; insert table

        scrollPane = new JScrollPane(table);
        scrollPane.getAccessibleContext().setAccessibleName("ScrollPane");
        scrollPane.getViewport().getAccessibleContext()
           .setAccessibleName("ViewPort");

        //          Put the scroll pane on screen

        add(scrollPane);
        Model.fireTableDataChanged();
        //          setVisible(true);
}
```

```
private void Initialize_Model () {
        int         i, j;

        if (trace == true)
           System.out.println ("In SaveTable.Initialize_Model ()  (write 0's to Save Table)");

        for (i = 0; i < 100; i++) {
           for (j = 0; j < 5; j++) {
                Model.data[i][j] = new Double (0.0);
           }
        }
}

//      Move new table data FROM Save object TO Savetable object

void update () {
        int         year;
        int         row;
        int         col;
        double      d;
        boolean     pd = false;         //          print '$'
        int         fracs = 2;          //          fraction digits to print
        int         mincols;

        // do not write script for cell changes now
        RunPressed = true;

        mincols = Currency.get_minColumns(S.years_balance[S.years - 1]);
        mincols += 3;     // for '.12', the CENTS

        if (trace == true)

           System.out.println
                ("In Savetable.update ()  (write Save object's data to Save Table)");

        for (year = 1; year <= S.years; year++) {
           row = year - 1;
           col = 0;

           //       Year

           table.setValueAt (new Double (year), row, col);

           //       Balance

           table.setValueAt (new Currency (S.years_balance[year],
                                           mincols, fracs, pd), row, ++col);

           //       Change

           d = S.years_balance[year] - S.years_balance[year - 1];
           table.setValueAt (new Currency (d, mincols, fracs, pd), row, ++col);

           //       Taxes Payed
```

```
            d = S.years_taxes_payed[year];
            table.setValueAt (new Currency (d, mincols, fracs, pd), row, ++col);

            //      Net Interest

            d = S.years_interest[year];
            table.setValueAt (new Currency (d, mincols, fracs, pd), row, ++col);
         }
         for ( ; year <= highest_row_used; year++) {
            row = year - 1;
            col = 0;

            //      Year

            table.setValueAt (new Double (0.0), row, col);

            //      Balance

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);

            //      Change

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);

            //      Taxes Payed

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);

            //      Net Interest

            table.setValueAt (new Currency (0.0, mincols, fracs, pd), row, ++col);
         }
         highest_row_used = S.years;
         Model.fireTableDataChanged();

         // write script for cell changes now
         RunPressed = false;
   }
   public void restore_background () {
         table.setBackground (parent.parent.Cwhite);
   }

}

class SaveTableModel extends AbstractTableModel {

   final String[] columnNames = {
         "Year",
         "Savings Balance",
         "Change",
         "Taxes Payed",
         "Net Interest",
   } ;

   Object data[][];
```

```
SaveTable parent;


SaveTableModel (SaveTable target){
        parent = target;
}

public int getColumnCount() {
        return columnNames.length;
}

public int getRowCount() {
        return data.length;
}

public String getColumnName(int col) {
        return columnNames[col];
}

public Object getValueAt(int row, int col) {
        return data[row][col];
}

public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
}

//      implement this method if table is editable

public boolean isCellEditable(int row, int col) {
        //          Note that the data/cell address is constant,
        //          no matter where the cell appears onscreen.

        if (col < 1)
           return false;
        else
           return true;
}

//      implement this method if table data can change

public void setValueAt(Object value, int row, int col) {
        if (row < 0 || row > 100)
           return;
        if (col < 0 || col > 4)
           return;
        data[row][col] = value;

        // CHANGE IN A CELL GETS THIS FUNCTION TO BE CALLED
        // Write script only when the RUN button is NOT pressed.


        //System.out.println("cell value changed");
        if (parent.RunPressed == false)
           this.fireTableCellUpdated(row, col);
```

```
    }

}


// we cannot change cells in this table by clickin' on it.  There is no
// accessible action, but since there may be additional actions attached
// in the future, we shall implement AccessibleAction anyways.
class BothTable extends JPanel {
   AAJTable              table;
   JScrollPane    scrollPane;
   BothTableModel        Model = new BothTableModel (this);
   TablePanel            parent;
   Both                  B;
   boolean               trace;
   int                   highest_row_used = 0;

   // this boolean is for the model to know when to write script
   // Must not write script when RUN button is pressed.
   static boolean RunPressed = false;

   //      constructor

   BothTable (TablePanel target) {

         // Always set accessible name.
         getAccessibleContext().setAccessibleName("BothTable");

         //          save ("pointer" to) parent object

         parent = target;
         trace = parent.trace;
         if (trace == true)
            System.out.println ("In BothTable () (constructor)");

         B = parent.B;
         Model.data = new Object[100][5];

         Initialize_Model ();

         //          create subpanel to contain table

         setLayout(new GridLayout(1,1,5,5));

         //          build table

         table = new AAJTable(Model);
         table.getAccessibleContext().setAccessibleName("Table");
         table.setPreferredScrollableViewportSize(new Dimension(450, 50));
         table.getColumnModel().getColumn(0).setPreferredWidth(1);
         table.setDefaultRenderer(Currency.class, parent);
         CSH.setHelpIDString(table, "Both.table");

         //          Create the scroll pane; insert table

         scrollPane = new JScrollPane(table);
```

```
        scrollPane.getAccessibleContext().setAccessibleName("ScrollPane");

        scrollPane.getViewport().getAccessibleContext()
           .setAccessibleName("ViewPort");

        //      Put the scroll pane on screen

        add(scrollPane);
}

private void Initialize_Model () {
        int     i, j;

        if (trace == true)
           System.out.println
                ("In BothTable.Initialize_Model ()  (writes Both Table's data to 0)");

        for (i = 0; i < 100; i++) {
           for (j = 0; j < 5; j++) {
                Model.data[i][j] = new Double (0.0);
           }
        }
}

void update () {
        int     year;
        int     row;
        int     col;
        boolean pd = false;     //      print '$'
        int     fracs = 2;      //      fraction digits to print
        int     mincols[] = new int[4];
        int     x;

        // do not write script for cell changes now
        RunPressed = true;

        //      Set Formatting

        for (int i = 0; i < 4; i++)
           mincols[i] = 0;

        for (year = 1; year <= B.years; year++) {
           x = Currency.get_minColumns(B.save_balance[year]);
           if (x > mincols[0])
                mincols[0] = x;
           x = Currency.get_minColumns(B.loan_balance[year]);
           if (x > mincols[1])
                mincols[1] = x;
           x = Currency.get_minColumns(B.net_taxes[year]);
           if (x > mincols[2])
                mincols[2] = x;
           x = Currency.get_minColumns(B.net_interest[year]);
           if (x > mincols[3])
                mincols[3] = x;
        }
```

```
        for (int i = 0; i < 4; i++)
           mincols[i] += 3;          // for '.12', the CENTS

        if (trace == true)

           System.out.println
                 ("In BothTable.update ()  (write Loan,Save objects' data to Both Table)");

        for (year = 1; year <= B.years; year++) {
  row = year - 1;
           col = 0;
           table.setValueAt (new Double (year), row, col);
           table.setValueAt (new Currency
                                 (
                                  //B.save_balance[year], mincols[0], fracs, pd),
                                  B.save_balance[year], pd),
                                 row, ++col);
           table.setValueAt (new Currency
                                 (
                                  //B.loan_balance[year], mincols[1], fracs, pd),
                                  B.loan_balance[year], pd),
                                 row, ++col);
           table.setValueAt (new Currency
                                 (
                                  //B.net_taxes[year], mincols[2], fracs, pd),
                                  B.net_taxes[year], pd),
                                 row, ++col);
           table.setValueAt (new Currency
                                 (
                                  //B.net_interest[year], mincols[3], fracs, pd),
                                  B.net_interest[year], pd),
                                 row, ++col);
        }
        for ( ; year <= highest_row_used; year++) {
           row = year - 1;
           col = 0;

           table.setValueAt (new Double (0.0), row, col);
           for (int j = 0; j < 4; j++) {
                 table.setValueAt (new Currency (0.0, mincols[j], fracs, pd),
                                       row, ++col);
           }
        }
        highest_row_used = B.years;
        Model.fireTableDataChanged();

        // write script for cell changes now
        RunPressed = false;

  }
  public void restore_background () {
        if (parent.parent.controlpanel.LoanPanel.text_changed == false &&
           parent.parent.controlpanel.SavePanel.text_changed == false)
           table.setBackground (parent.parent.Cwhite);
  }
}
```

```
class BothTableModel extends AbstractTableModel {

    final String[] columnNames = {
            "Year",
            "Savings Balance",
            "Loan Balance",
            "Net Taxes",
            "Net Interest",
    } ;

    Object data[][];

    BothTable parent;


    BothTableModel (BothTable target){
            parent = target;
    }

    public int getColumnCount() {
            return columnNames.length;
    }

    public int getRowCount() {
            return (100);
    }

    public String getColumnName(int col) {
            return columnNames[col];
    }

    public Object getValueAt(int row, int col) {
            return data[row][col];
    }

    public Class getColumnClass(int c) {
            return getValueAt(0, c).getClass();
    }

    //      Implement This Method If Table Is Editable

    public boolean isCellEditable(int row, int col) {
            //      Note that the data/cell address is constant,
            //      no matter where the cell appears onscreen.

            return (false);
    }

    //      Implement This Method If Table Data Can Change

    public void setValueAt(Object value, int row, int col) {
            data[row][col] = value;

            // CHANGE IN A CELL GETS THIS FUNCTION TO BE CALLED
            // Write script only when the RUN button is NOT pressed.
```

```
        //System.out.println("cell value changed");
        if (parent.RunPressed == false)
            this.fireTableCellUpdated(row, col);
    }

}
```

AccessLS/AccessLS.java

```
import javax.swing.*;
import javax.swing.event.*;
import javax.accessibility.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import java.io.*;
import com.sun.java.accessibility.util.*;

/*  This program accesses LoanSave application.  It replays the script
   on LoanSave.  */

// implements GUIInitializedListener to wait till the SimpleExit window
// pops up.  does not extend JFrame; run by the LoanSave application

public class AccessLS implements  GUIInitializedListener{

   public JFrame frame;
   Accessible accessible;
   static Accessible child;
   JButton button;
   File f;
   static String contentLine;
   BufferedReader bufr;
   int actNum;


   // must wait till the SimpleExit GUI is initialized in the VM

   public AccessLS(){
         if (EventQueueMonitor.isGUIInitialized()){
            createGUI();
         } else {
            EventQueueMonitor.addGUIInitializedListener(this);
         }
   }


   public void guiInitialized() {
         createGUI();
   }


   public void createGUI() {

         WindowListener l = new WindowAdapter() {
            public void windowClosing(WindowEvent e) { System.exit(0);}
         };

         frame = new JFrame("Accessing LoanSave");
         frame.getContentPane().setLayout(new FlowLayout());
         frame.addWindowListener(l);
         button = new JButton("Play Script");
         frame.getContentPane().add(button);
```

```
// when pressed Play Script button, it gets the script file, parses
// each command, get the accessible object, and perform appropriate
// actions.

button.addActionListener(new ActionListener(){
   public void actionPerformed(ActionEvent e) {
         AccessibleContext context;
         AccessibleAction accessibleAction;

         Accessible accessibleTemp;


         // grab windows and get accessible components

         Window[] wins = EventQueueMonitor.getTopLevelWindows();
         for (int i = 0; i < wins.length; i++) {
            accessibleTemp = Translator.getAccessible(wins[i]);
            if (accessibleTemp != null) {

                  // only grab the "LoanSave" accessible component

                  if (accessibleTemp.getAccessibleContext()
                     .getAccessibleName() == "LoanSave"){

                     accessible = accessibleTemp;}
            }
         }


         // if LoanSave application is not running, exit.

         if (accessible.getAccessibleContext().getAccessibleName()
            != "LoanSave"){

            System.out.println("Could not grab LoanSave application.");
            System.exit(0);
         }


         // parse the script file and do accessible action

         try{
            f = new File("/u/youngkim/1999/LoanSave/script");
            contentLine = "";
            if (!f.exists()){

                  System.out.println
                     ("Could not find /u/youngkim/1999/LoanSave/script");
            }
            else {
                  int index;

                  bufr = new BufferedReader(new FileReader(f));
                  String in = new String();
```

```
String line = new String();
while ((line = bufr.readLine()) != null)
   in = in.concat(line + "\n");
bufr.close();

index = in.indexOf('\n');


// go through every line

while (index != -1){
   contentLine = in.substring(0, index);
   in = in.substring(index+1);

   child = accessible;
   //System.out.println(contentLine);


   // gotta get rid of the first name since we're
   // already in it.

   String childName = getNextChildName();


   // traverse down the accessible tree to find the
   // correct widget accessible
   /* Monkey application is helpful in debugging */

   while ((childName = getNextChildName()) != null){
         String cn;
         System.out.println(childName);
         context = child.getAccessibleContext();
         int count = context.getAccessibleChildrenCount();
         for (int i = 0; i < count; i++) {

            if (context.getAccessibleChild(i).
                  getAccessibleContext().getAccessibleName()
                  == null)

                  cn = "null";

            else
                  cn = context.getAccessibleChild(i)
                     .getAccessibleContext()
                     .getAccessibleName();

            System.out.println("C="+childName+" AN="+cn);
            if (childName.equals(cn)){
                  child = context.getAccessibleChild(i);
                  System.out.println("assigning child");}

         }
   }


   //System.out.println("before parsing action number");
```

```
                        actNum = getActionNumber();

                        getAndWriteArgument();

                        System.out.println
                              (child.getAccessibleContext().getAccessibleName());

                        //System.out.println("before doing accessible action");

                        if (child.getAccessibleContext().getAccessibleAction()
                              ==null)

                              System.out.println("There is no accessible action!");

                        else
                              child.getAccessibleContext().getAccessibleAction()
                                 .doAccessibleAction(actNum);

                        //System.out.println("after callin' AA");

                        index = in.indexOf('\n');
                     }
                     //                              in.close();
                  }
               }
               catch (Exception ioe) {
                  System.out.println(ioe);
               }
            }
         });
         frame.setSize(400, 200);
         frame.show();
}


// Returns the next child name, using the static contentLine
// basically after this function call, the child name gets cut
// from contentLine.

public String getNextChildName(){

      if (contentLine.startsWith(".Row=")){
         int endMsg = contentLine.indexOf(':');

         child.getAccessibleContext().setAccessibleDescription
               (contentLine.substring(1, endMsg));

         contentLine = contentLine.substring(endMsg+1);
         return(null);}

      if (contentLine.startsWith(":")){
         contentLine = contentLine.substring(1);
         return(null);}

      contentLine = contentLine.substring(1);
      int endName = contentLine.indexOf('.');
```

```
        int rowIndex=contentLine.indexOf("Row=");
        int colonIndex=contentLine.indexOf(":");

        if ((endName == -1) || ((endName > rowIndex) && (rowIndex != -1)))
           endName = rowIndex;

        if ((endName == -1) || ((endName > colonIndex) && (colonIndex != -1)))
           endName = colonIndex;

        String name = contentLine.substring(0, endName);

        // we do not use endName+1 because we want to save :(colon)
        contentLine = contentLine.substring(endName);
        System.out.println(contentLine);
        return name;
}


// Returns the action number from contentLine.  contentLine must have
// action number (can be "") in the beginning followed by a :(colon)

public int getActionNumber(){
        System.out.println("contentLine received = "+ contentLine);
        if (contentLine.startsWith(":")){
           contentLine = contentLine.substring(1);
           return(0);}

        int endNum = contentLine.indexOf(':');
        System.out.println(endNum);
        int actNum = Integer.parseInt(contentLine.substring(0, endNum));
        System.out.println(actNum);
        contentLine = contentLine.substring(endNum+1);
        return actNum;
}


// contentLine must have the argument (can be "") in the beginning followed
// by //(double slash) or EOL.
// If there's no argument, this function does nothing.
// If there is one, it parses it out and writes it at accessible action
//    description so that LoanSave application can access it.
//    For a table argument, since the accessible action description
//    contains "Row=" and "Col=", it should just concatenate argument
//    with "Arg =".

public void getAndWriteArgument(){
        System.out.println("inside arg parse: "+contentLine);
        if (contentLine.startsWith("//")) return;
        int endArg = contentLine.indexOf("//");
        contentLine = contentLine.substring(0, endArg);
        System.out.println("arg value= "+contentLine);
        AccessibleContext ac = child.getAccessibleContext();
        System.out.println("previous description:"+ac.getAccessibleDescription());
        if (ac.getAccessibleDescription() != null){
           if (ac.getAccessibleDescription().startsWith("Row="))
```

```
                ac.setAccessibleDescription(ac.getAccessibleDescription().concat
                                        ("Arg="+contentLine));

            else
                ac.setAccessibleDescription(contentLine);
        }
        else
                ac.setAccessibleDescription(contentLine);

        System.out.println("done writing argument");
    }

    public static void main (String s[]){
        String vers = System.getProperty("java.version");
        if (vers.compareTo("1.1.2") < 0) {
            System.out.println("!!!WARNING: AccessLS must be run with a " +
                               "1.1.2 or higher version VM!!!");
        }
        new AccessLS();
    }
}
```