April 2006

The Eclipse JUnit Test Recorder

Benjamin A. Mohlenhoff
Worcester Polytechnic Institute

Damon R. Bussey
Worcester Polytechnic Institute

THE ECLIPSE JUNIT TEST RECORDER

A Major Qualifying Project Report:

submitted to the faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Damon Bussey

and

Benjamin Mohlenhoff

Date: April 27, 2006

Approved:

1. Object-Oriented Design
2. Software Engineering
3. Eclipse Plug-in

Professor Gary Pollice, Advisor

Abstract

The Eclipse JUnit Test Recorder (EJUTR) is a plug-in for the Eclipse software framework that adds additional functionality to an existing Eclipse plug-in known as the Eclipse-Based Object Bench (E-BOB.) EJUTR allows users to easily create JUnit test cases based on their interactions with the Object Bench. Using E-BOB and EJUTR together allows the user to both interact with Java objects and implement JUnit test cases without requiring knowledge of either Java or JUnit syntax.

Acknowledgements

We would like to thank all those that made our project possible, in particular Professor Gary Pollice. Without his knowledge of the Eclipse platform, we would probably have not had a project in the first place. In addition, we would like to sincerely thank Liam Morley and Justin Braga for their initial work on the Eclipse-based Object Bench, the creation of which made our work possible.

Table of Contents

Table of Figures

1 Introduction

Man has always been a toolmaker. When confronted with a problem, the tendency of man is to seek out a tool to assist him in overcoming that problem. Of course, if no suitable tool for a given problem can be found, man creates one. Problems in software development are no different. We create tools to help us solve problems.

One of the latest software development tools is the Eclipse platform. The Eclipse platform is a powerful piece of software that was developed with the intention of creating a software system with an open architecture. The design of Eclipse consists of a small central core which provides basic functionality to the platform. The rest of the platform exists as a series of plug-ins which attach to that central core and provide additional layers of functionality. This multi-tiered, open architecture is one of the greatest strengths of the Eclipse platform because it allows software developers to easily leverage and utilize existing functionality while writing code to accomplish additional tasks. The platform features a full-fledged integrated development environment (IDE) for Java, and it has quickly become a standard across the software development industry due to the fact that it is free.

In 2005, a Major Qualifying Project (MQP) was sponsored by the Worcester Polytechnic Institute (WPI) which detailed the creation of an Eclipse-based Object Bench (E-BOB). E-BOB functioned as an Eclipse plug-in, and allowed the user to easily manipulate Java objects by setting variables and calling methods within those objects through the use of context menus and wizards. E-BOB functioned primarily as an educational tool because it allowed users to easily visualize and interact with Java objects without requiring knowledge of Java syntax. All interactions with E-BOB are through the use of context menus, wizards, and drag-and-drop operations. In addition to being an educational tool for Java beginners, E-BOB also appeals to more experienced Java programmers because it allowed those users to quickly and easily instantiate and test objects. However, one of the limitations of E-BOB is that while it allowed the user to easily manipulate Java objects, it offered no functionality allowing the user to record those manipulations. Often, users would execute the same interactions with the Object

Bench many times in a row while they were attempting to use it to test and debug their software.

Our project, the Eclipse JUnit Test Recorder (EJUTR), is an Eclipse plug-in that complements existing E-BOB functionality by adding additional unit testing features. EJUTR allows the user to record the manipulations of objects within E-BOB. The actions can then be written to a file such that the user could then easily view and execute those manipulations in the form of a JUnit test. A JUnit test functions as a series of actions and assertions that compare the expected result of some operation against the actual result of that operation. These tests help the user easily see which pieces of functionality within their objects execute correctly, and which pieces function incorrectly.

In addition to allowing the user to interact with Java objects without having knowledge of Java syntax, E-BOB and EJUTR together will allow the user to generate JUnit test cases without having knowledge of JUnit syntax. The user will be able to concentrate on deciding what specific functionality to test and how to accomplish those tests, instead of focusing on the specific format of JUnit test cases.

2 Background

2.1 Eclipse History

What is now known as Eclipse started out as a way for IBM to design multiple desktop tools all deriving from a common software base. Eclipse was designed to be a common platform for all of IBM's development products, as it would help IBM develop multiple tools without duplicating many common elements (D'Anjou, et al., xxix). By using Eclipse as a base, different tools developed by different parts of IBM would look similar to and integrate easily with each other.

Work on the Eclipse project began in 1998 when the IBM Software Group started work on a tools platform. In November 2001 IBM adopted an open source licensing for the Eclipse technology to encourage third parties to develop for the platform. Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft and Webgain created the Eclipse consortium and eclipse.org (About Us, par 6). The first major release of Eclipse took place in 2003. It was well received, but many developers saw Eclipse as IBM-controlled and as a result major vendors were reluctant to make contributions to Eclipse. In an effort to make Eclipse seem more independent, IBM formed the Eclipse Foundation. The Eclipse foundation is a not for profit organization with independent professional staff funded by dues collected from member companies. Eclipse was released under the Eclipse Public License, which gives the developers of its plug-ins world wide distribution rights that are royalty free (About Us, par 7).

Eclipse has become a very popular Java IDE. It is used in both educational and commercial environments. The Eclipse foundation has over 115 member companies and hosts over nine major open source projects. Between the spring of 2003 to the spring of 2004 usage of Eclipse in Europe, Asia, and North America grew by 75% (Growth, par 1). In North America growth was 90% (Growth, par 7). Today the Eclipse SDK has become one of the most widely used Java development environments and its popularity is still growing.

2.2 Eclipse Architecture

2.2.1 The Eclipse Platform

The Eclipse architecture, shown in Figure 1, consists of several parts. At the core of Eclipse is a system dependent runtime. The platform runtime is a small kernel that defines the plug-in infrastructure (Gamma, Beck 6). The rest of the Eclipse Platform is made up of a set of plug-ins that provides several major components. Theses components are SWT, JFace, the workbench, the workspace, and the team and help systems.

The Standard Widget Toolkit (SWT) provides a platform independent API for widgets and graphic library routines. Each SWT implementation is tightly integrated with a specific native window system. SWT provides a common interface by using native widgets when possible and emulating them when they do not exist in the native window system. The Eclipse UI is completely dependent on SWT. Java Swing and AWT are not used.

JFace is a UI toolkit implementation that uses SWT to provide common user interface (UI) functionality. JFace's implementation and API are platform independent. It provides the ability to create actions and views. Actions represent user commands. Different UI elements can be tied to the same action, allowing multiple ways for the user to accomplish the same task. Views display information about some object to the user. For example the Outline view displays member and method names about the object the user is has selected.

The workbench provides the basic structure of the Eclipse UI. The Eclipse UI is centered around two main types of components: views and editors. Editors allow the user to open, edit and save objects such as text files. Views represent panels within the Eclipse environment, and provide information to the user about various things. Eclipse also utilizes "Perspectives", which are collection of related editors and views. When a particular perspective is selected, the user interface of the Eclipse environment can change significantly.

The workspace portion of the Eclipse framework provides resource management for the rest of the platform. It contains sets of projects that map to user defined directories on the native system. Other plug-ins do not need to interact with the native

file system because the workbench provides different ways to access the workspace projects.

The help system provides several online books. It allows other plug-in to contribute help documents to the system. The team system allows users to easily synchronize update and commit projects to a CVS repository.

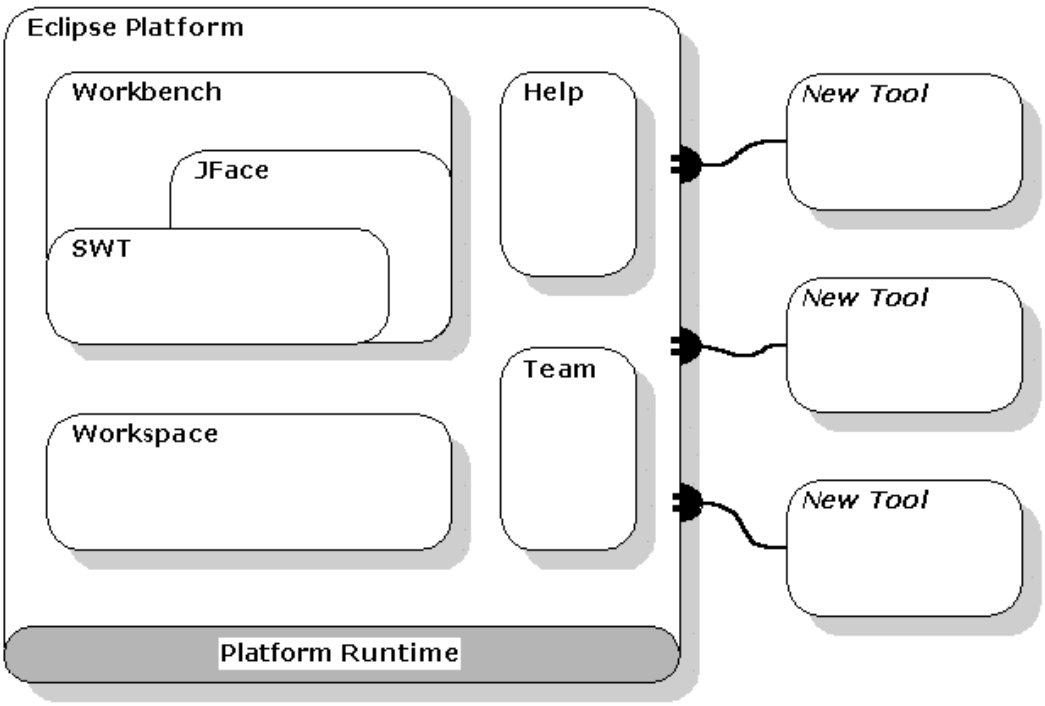

Figure 1 - The Eclipse Architecture (Technical Overview, 3)

2.2.2 Extension Points

Plug-ins add functionality to Eclipse through extension points. Each Eclipse plug-in contains a manifest file called "plugin.xml". The plug-in manifest contains information about what extension points the plug-in uses and provides descriptions of any extension points that the plug-in will provide to the rest of the system. A plug-in's extension points describe extensions that other plug-ins can use to add or modify functionality. For example, workbench extension points can enable plug-ins to add menus item, views, wizards or actions to the Eclipse UI. A plug-in can take advantage of multiple extension points from different plug-ins and also provide extension points that

any other plug-in can use. Figure 2 shows an example of a new plug-in taking advantage of extension points both inside the Eclipse framework and outside of it.

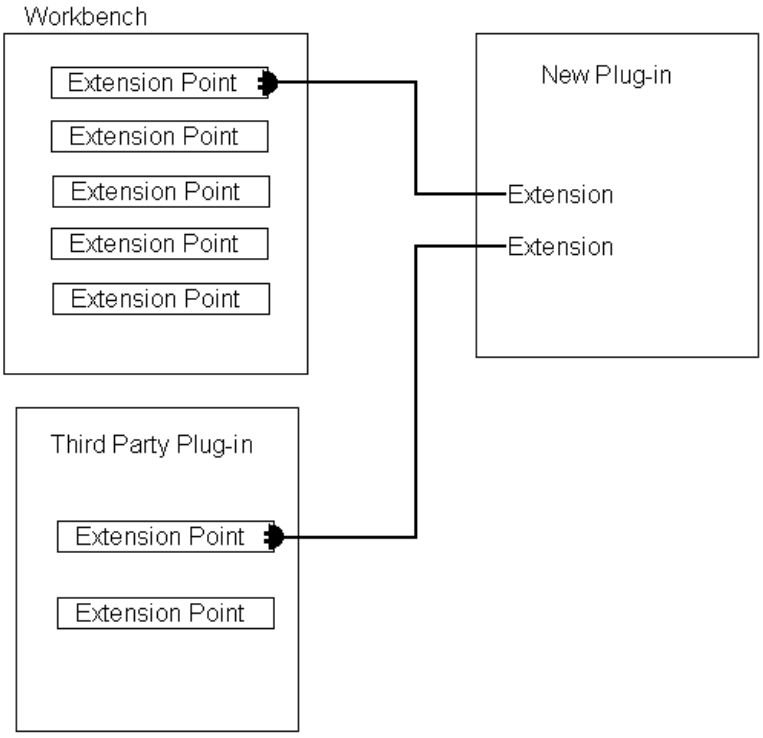

Figure 2 - Extension Points

When the Eclipse platform starts, the platform runtime inspects all of the installed plug-in manifest files. Not all plug-ins are started at runtime, but all of the extension points the plug in uses and the extensions it provide are read into a registry. Although the menu item and toolbar buttons described in plugin.xml are displayed when eclipse starts, the actual plug-in is not leaded until its functionality is needed. This is known as "lazy-loading" behavior, because plug-ins are not physically loaded by the Eclipse run-time until they are actually required. This helps to cut down on Eclipse's footprint within system memory.

2.2.3 JDT – Java Development Tooling

The Eclipse platform provides a "Rich Client" platform. This means that applications for many different purposes can be built on top of Eclipse without having to start from scratch. The application will run on any system the underlying platform will

run on. The Eclipse platform by itself is an "…IDE for anything, and for nothing in particular" (Technical Overview, 13).

The Java Development Tooling (JDT) plug-in turns Eclipse into a powerful Java development environment. The Eclipse application comes with the JDT plug-ins installed by default. Some of the many features the JDT adds to Eclipse include the ability to create and browse Java projects, Java editors with code highlighting, code refractory, code searching, a JCK-compliant Java compiler, the ability to run Java programs in a separate Java virtual machine and a powerful Java debugger (Technical Overview, 14-15). The JDT plug-ins provide the default Java perspective that Eclipse utilizes. The Java perspective provides the java editors, outlines and actions for developing Java code.

2.3 Unit Testing

Unit testing involves creating tests for a single unit, module, or piece of code. Each unit test should test a particular aspect of a particular piece of code. The goal of unit testing is to isolate each part of a program and verify that each individual part functions properly. The tests should take into account every aspect of the expected behavior of the code. A set of tests is written for each unit of code. These suites of unit tests allow a programmer to confidently edit code knowing that any side effects caused by the changes can be easily identified by running the tests. This process is known as regression testing, and automated unit testing makes regression testing simple.

JUnit is the definitive unit testing framework for Java. It was developed by Kent Beck and Erich Gamma. It is part of the xUnit framework, which was originally developed by Kent Beck. The xUnit framework is a set of unit testing frameworks for many different computer languages. Eclipse comes with a JUnit plug-in that lets user easily create new test case files and execute those files in the form of JUnit tests.

The JUnit libraries provide the TestCase class. Individual JUnit test cases must extend this class, which allows JUnit test runners the ability to run those tests and summarize the results. The setUp() and tearDown() methods can be overwritten in a TestCase class file, and provide an easy way to create and destroy test data for each individual test. For each test in a TestCase class the JUnit test runner calls the setUp()

method before each test method and the tearDown() method after each test method finishes.

The test case class also provides a number of assertion methods. The assertion methods are used to test the return values on methods in the class under test. In an assertion fails, then the test fails. Assertion methods usually take two arguments and compare them for equality. For example the method assert(boolean expected, Boolean actual) takes two boolean values and compares them.

2.4 E-BOB – The Eclipse-based Object Bench

The Eclipse-Based Object Bench is an Eclipse plug-in that allows user to create, inspect, and manipulate Java objects graphically within the Eclipse environment. It was developed in 2005 as a Major Qualifying Project (MQP) by Liam Morley and Justin Braga. This plug-in contributes the Object Bench view to Eclipse. A context menu is generated when a user right clicks on an object within the Bench and allows the user to inspect the values of fields within the object and invoke methods on that object. E-BOB displays dialogs that allow the user to specify parameters to method calls, if necessary.

The figure below shows the E-BOB Object Bench view. The two colored squares represent objects that have been instantiated. The name of the object appears above its type. Right-clicking an object will cause a context menu to appear. The "Inspect" menu item will bring up the Instance Properties view. The Instance Properties view shows current state of the object's internal fields. The right-click context menu will also display the methods of the selected object in submenus. Selecting methods within this menu will cause E-BOB to invoke that particular method on the selected object.

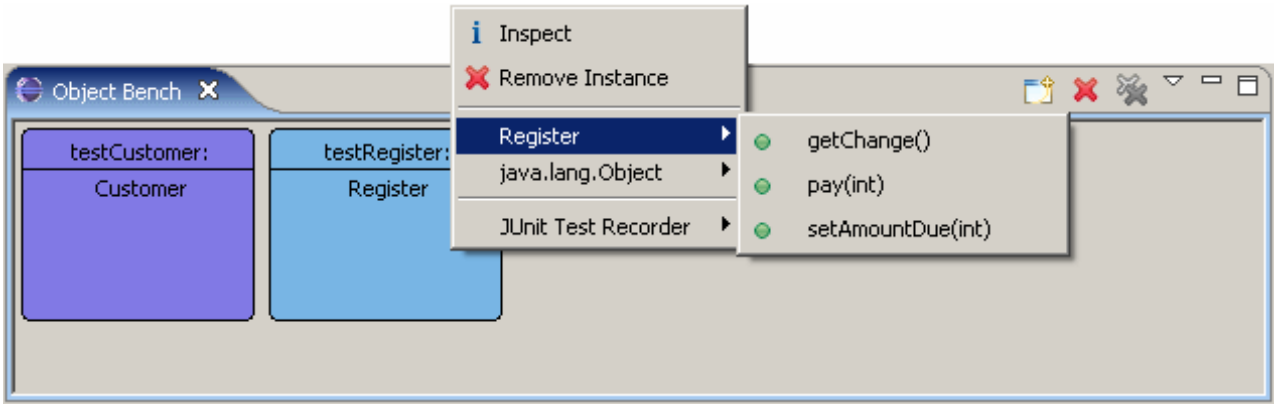

Figure 3 - The Eclipse-based Object Bench

The Conclusions and Further Work section of the E-BOB MQP report that there is no way to record actions performed on the Object Bench (37). This makes it difficult to use to use E-BOB to test the functionality of code. Each time a user wishes to test a method on an object, they must go through the steps of instantiating and setting the initial state of the object. These steps can not be recorded in order to allow the user to simply "play back" his operations with the Object Bench using the default E-BOB functionality.

3 Methodology

3.1 Research and Development

The first step in developing an Eclipse plug-in is to learn the Eclipse platform and determine which pieces of the platform the plug-in being developed needs to interface with. The platform itself is sufficiently complicated enough to make implementing a plug-in a non-trivial exercise, but fortunately there are a lot of resources available to assist a new Eclipse developer. There are a variety of tutorials and plug-in builders that enable even someone entirely new to Eclipse to generate simple plug-ins that attach to the Eclipse framework at commonly used extension points.

The first step was to create a default plug-in that would add a button to a bar, and a menu item to a context menu, in order to determine how difficult actually implementing an Eclipse plug-in is. There are a variety of plug-in tutorials and walkthroughs that are provided through the Eclipse Help system. Therefore, this was not a difficult task to accomplish. The only confusing part about Eclipse plug-in development is the necessity of having two instances of Eclipse executing at the same time. The first instance contains the code for the plug-in, while the second instance has physically loaded that code and is executing it. It is easy to get confused and forget which instance of Eclipse is the development environment and which is the test environment.

The EJUTR plug-in needed to interface with Eclipse and the E-BOB plug-in in several places. Therefore, one of the first tasks was to identify those areas and determine how to implement a plug-in which would attach to those areas without interfering with anything else. The first such area was that of the various popup menus within Eclipse. EJUTR needed to tie into the menus offered by E-BOB and add additional menu items that would control the functionality of EJUTR. The first step here was to figure out how to generate a plug-in that would add additional menu items to some of the default menus found throughout Eclipse. After that, pointing that menu-adding code towards the menus provided by E-BOB would be an easy task. In addition to popup menus, E-BOB utilizes a series of ‘Wizards’ in order to configure itself and to provide a usable and clean user interface for several of its operations, such as instantiating new objects or executing

methods. Therefore, it was important to determine how to create such wizards and how to tie in additional content into existing wizards.

EJUTR needed to generate JUnit test files, so it was therefore necessary to determine how best to use the Eclipse framework to create new files within the Eclipse workspace, and how to append additional content into existing files. There are a variety of methods for doing this, such as to modify the files directly by using methods found within the java.io package, which as its name suggests provides various input/output operations within Java. However, when dealing with files that need to exist within an Eclipse workspace, it is necessary to utilize specific Eclipse-provided methods when manipulating those files. Eclipse maintains a model of its currently open projects within system memory, so if the contents of the projects were to change on disk, those changes might not be reflected in that internal model. Therefore, in order to keep things synchronized, it was necessary to utilize those specific Eclipse-provided methods when creating or appending to JUnit files.

3.2 Modify E-BOB

The next step was to look at the code provided within E-BOB, and determine what (if any) changes needed to be made in order for EJUTR to function properly. Unfortunately, after looking at the code, it became apparent that E-BOB was not designed with future extensibility in mind. While certain information needed by EJUTR was contained within and processed by E-BOB, there was no way for EJUTR to physically access that information. Therefore, it was necessary to modify several components of E-BOB in order to expose additional information to EJUTR and any other future plug-ins that might require access to that information.

Since the purpose of EJUTR is to record manipulations of objects within the Object Bench with the intention of generating JUnit tests at some later point, it was important to ensure that EJUTR would be notified whenever objects were added to the bench, whenever objects were removed from the bench, and whenever the state of any of the objects within the bench changed. After examining the user interface of E-BOB, it became apparent that the only way to internally change the state of an object within the

Object Bench was to call some method within the object; it was not possible to modify the contents of the object directly. This helped to simplify the process of determining how to intercept information regarding state changes to objects, as it was only necessary to intercept method calls. Even then, it was only necessary to intercept just enough information in order to duplicate the method call later inside of a JUnit test.

E-BOB provides an "ObjectBenchListener" interface, which allows external classes to receive information regarding the changes of objects within the Object Bench. The initial version of this interface provided methods that were called whenever objects are added to (objectAdded()) or removed from (objectRemoved()) the bench. These calls would correspond to an object being either constructed or destroyed, and the corresponding calls to those methods within the interface were already coded into the behavior of E-BOB. However, the interface and E-BOB did not provide any way to notify its listeners of when methods within the objects in the bench were called, so it was necessary to modify the listener interface to provide this functionality. A "methodCalled()" method was added to the listener interface to accomplish this. This new method is called whenever E-BOB actually executes a method. In addition, changes were made to the "ExecuteMethodAction" class within E-BOB in order to trigger the new method that was added to the listener interface. The "ExecuteMethodAction" class is physically executed when the "Execute Method" menu item is clicked within the E-BOB context menu.

E-BOB utilizes several internally-defined types when it stores instantiations of objects within its bench. These internal types contain information regarding the objects within the bench that they represent, but it is not safe to pass references to those internal objects out through the listener interface. If references to the internal types were to be passed via the listener interface to some client code, it would be possible for that client code to modify the internal contents of the Object Bench. Since this is not desirable, E-BOB will now translate from its own internal types into new objects that were designed with the sole interest of passing out of the plug-in via the listener interface. Two objects were added to E-BOB to accomplish this, namely "MethodInvokeInfo" and "MethodInvokeParamInfo". By translating from E-BOB's internal types to these safe external types, code which implements the listener interface will be unable to modify the

internal contents of the Object Bench. Instantiations of these objects are passed via the listener interface whenever a call to "methodCalled" is made.

3.3 Construct EJUTR

After implementing the changes to E-BOB, work began on the EJUTR plug-in itself. The first task was to implement the listener interface and to ensure that it was possible to receive all of the information about method calls necessary to generate JUnit tests at some later point.

All that is needed to duplicate a method call within a test file is the method name, the object which contained the method, and a list of the parameters passed to that method call. Once all that information is available, it is a simple matter of parsing it into correct Java syntax and writing it to a test file. Of course, the file would later be parsed, compiled, and passed into JUnit, but that functionality is not the responsibility of EJUTR; EJUTR's responsibility is to merely log Object Bench interactions and write out a test file. Therefore, ensuring that the listener interface provided all of that information was the highest priority. The only potential issue with this strategy is that if complex objects are passed as parameters, all the information that the listener will be able to provide about them is their type and their variable name. However, this is not a problem because of how the E-BOB user interface is designed. Before it is possible to call a method that takes some other object as a parameter, it is necessary to explicitly instantiate that second object, and to initialize it either by means of a constructor or several method calls (most likely to 'setter' methods) that will configure the state of the object. These methods will be picked up and reported to EJUTR via the listener interface, so whenever a method requires an object as a parameter, the operations to define that parameter object will already be recorded. Therefore, the problem is avoided entirely.

The second step was to determine how best to internally store the recording of object additions, object removals, and method invocations. The internal storage scheme needed to be able to handle several different types of objects such as object additions, object removals, and JUnit assertions. The internal storage scheme needed to provide an

indeterminate amount of storage, as well. For example, there could be only one record stored, or one million records stored.

The third step was to implement a scheme for writing the contents of the internal record to a test file. EJUTR needed to know exactly how to parse and write out each element within the internal recording to comply with standard Java syntax.

The final step of implementing EJUTR was to design and implement the GUI components of the plug-in. Several components were necessary, such as graphical elements that would allow the user to begin recording his actions, and elements that would allow the user to end the recording process. In addition, these GUI elements needed to be displayed at a certain point within E-BOB's execution (for example, displaying the Assertion Wizard after E-BOB performs a method call on an object, but before the results of that call are returned to the user.)

More information about the specific implementation details of EJUTR can be found in the Results and Findings chapter.

3.4 Distribution

The final problem to solve on this project was how to handle the distribution of EJUTR, and the modified version of E-BOB. Fortunately, Eclipse provides methods to easily export code into a JAR (Java Archive) file, which it can then utilize as a plug-in. In addition, the modified E-BOB code will be submitted to the developers of E-BOB through www.sourceforge.net, which hosts the project.

4 Results and Findings

4.1 EJUTR Software Design

The overall software design of the Eclipse JUnit Test Recorder is not overly complex; however, it is important to understand how every component interacts with every other component within the plug-in before examining each individual object in great detail. Each component has specific responsibilities and tasks that it must accomplish, but the most important thing to consider in the design of the system is how each component will communicate with other components.

The first piece of functionality within EJUTR is the functionality that attaches to E-BOB and receives information about object additions, object removals, and method invocations from the Object Bench. The next step is to use the information that EJUTR receives to form some sort of record that will allow EJUTR to later generate syntactically correct Java code within a JUnit test case. The next step is to store those records within some sort of internal scheme which will allow EJUTR to later access those records and generate complete JUnit test cases. These three steps occur in sequence whenever EJUTR is set to record the users interactions with the Object Bench, which is triggered through menu items that EJUTR contributes to E-BOB's context menus (specifically, the "Record New Test" menu item.) The functionality described in the previous set of steps is provided by the ebob_observer and logger packages. The next phase of functionality that EJUTR provides is the ability to go back and generate JUnit test case files based on the contents of the action log. This functionality is initiated when the user selects the "Stop Recording" menu item that EJUTR contributes to E-BOB's context menus. EJUTR then iterates through the contents of the internal action log, and generates a String of properly formatted Java code, which is then written to the JUnit test case that the user specified when the process of recording was initiated. The functionality to achieve these steps is located in the logger and file_operations packages within EJUTR. All the menu items that are contributed to the E-BOB menus are located within the actions packages, and all of the wizards that are used to interface with the user are contained within the wizards package.

EJUTR makes use of the Singleton design pattern, as there are several

components with the EJUTR plug-in that should exist only once within Eclipse. It would not make sense for multiple instances of the action log (where the actions within EJUTR are physically stored) to exist, for example. The Singleton design pattern ensures that only one element of classes with utilize the pattern exist within a software system through the use of a private constructor method, and all external objects that wish to interact with a singleton must first obtain a handle to that object by using some method that returns a handle to the singleton instance of the object. The classes within EJUTR that implement this pattern are EjutrPlugin, EBOBListener, and EBOBActionLogger.

The EJUTR plug-in consists of several packages, all of which contain objects. Each package corresponds to a specific subset of the functionality within EJUTR. For example, the logger package contains code that relates to how EJUTR physically stores and provides external objects access to the internal action log, and the wizards package contains the code that displays wizards to the user (such as the Assertion or Record New Tests wizards.)

If you would like more information about specific methods, fields, or classes within the EJUTR plugin, please consult the EJUTR Javadoc, which can be viewed on the EJUTR web site, or generated by running the Javadoc utility on the EJUTR source code tree. Table 1 contains the names of the Java packages within EJUTR, and a brief description of each.

Table 1 - Package Structure of EJUTR

Package Name	Description
edu.wpi.ejutr	Contains plug-in code which interfaces with the Eclipse framework.
edu.wpi.ejutr.actions	Contains the physical elements displayed within context menus.
edu.wpi.ejutr.debug	Contains debug code used to test EJUTR.
edu.wpi.ejutr.ebob_observer	Contains code that communicates and interfaces with E-BOB.
edu.wpi.ejutr.file_operations	Contains code that interfaces with the physical file system.
edu.wpi.ejutr.logger	Contains code that records and stores E-BOB's actions.
edu.wpi.ejutr.testobjects	Contains objects used to test EJUTR.
edu.wpi.ejutr.wizards	Contains the graphical wizards that are displayed to the user.

4.1.1 edu.wpi.ejutr

The edu.wpi.ejutr package contains code which implements the Eclipse-defined interface required of all plug-ins. This package contains the code that physically plugs in to the Eclipse architecture in order to make all the other functionality within EJUTR accessible from within the Eclipse framework. The only class contained within this package is the EjutrPlugin class, which implements several Eclipse-defined interfaces. These interfaces provide specific methods that all plug-ins must implement, in order to guarantee that Eclipse is able to communicate with the plug-in. These methods include start() and stop() methods, which are called when the plug-in is loaded or unloaded (as Eclipse allows plug-ins to be dynamically loaded and unloaded while the rest of the system is running.) In addition, the EjutrPlugin class provides methods allowing other objects to get a handle to the plug-in instance itself (necessary for certain Eclipse functions) and a handle to E-BOB's plug-in instance. It is necessary for EJUTR to have a handle to the currently running instance of the E-BOB plug-in so that EJUTR can access member objects and interfaces of E-BOB, such as the ObjectBenchListener interface provided within E-BOB.

4.1.2 edu.wpi.ejutr.actions

The edu.wpi.ejutr.actions package contains the physical objects that appear throughout the context menus that EJUTR interfaces with. The code within each of these objects is executed whenever a context menu item is clicked. The objects themselves can appear in any of the context menus within Eclipse, as the location of each is defined with the "plugin.xml" file found within the EJUTR plug-in. Therefore, if it was later necessary to add EJUTR menu items to additional menus within Eclipse, the "plugin.xml" file would need to be modified, and additional elements within that file would point to objects within the actions package.

The RecordNewTestMethod class is the physical object that appears within certain E-BOB and EJUTR context menus as the "Record New Test Method" item. When clicked, it performs certain background operations such as clearing all current objects from E-BOB's internal representation of its Object Bench, and then instantiates

and displays the RecordNewTestMethod wizard found within the edu.wpi.ejutr.wizards package. The RecordNewTestMethod class waits until that wizard is closed, and if the wizard is closed using the "Finish" button, the RecordNewTestMethod class populates the global EJUTRTargetFileInfo object with the contents of the fields within the wizard. If the wizard is closed via the "Cancel" button, then nothing is stored, and no further behavior occurs.

The RecordNewTestMethod class then removes all current objects from the Object Bench. It is necessary to remove all objects from the Object Bench because there is no way to get the complete and total state of each object through the ObjectBenchListener interface, and therefore it would not be possible to guarantee that the state of an object within the generated JUnit test case file would match the state of the object existing within the Object Bench when the recording is started. The simplest way to fix this problem was to ensure that when the recording started, all objects would be removed from the Object Bench such that the user would then re-construct the objects necessary for the JUnit tests, and all state-changing method calls would then be logged.

The final action of the RecordNewTestMethod class is to attach the instance of the EBOBListener that is contained within EJUTR to the E-BOB plug-in, so that EJUTR will be notified of any changes to the internal state of E-BOB. Until EJUTR is actually recording, it is not notified of any E-BOB events because it simply does not need to know.

The StopRecording class contained within the edu.wpi.ejutr.actions package is physically displayed within both EJUTR and E-BOB context menus as the "Stop Recording" item, and its purpose is exactly the opposite of the RecordNewTestMethod object. The purpose of this class is to physically stop EJUTR from recording additional interactions with the Object Bench, and to determine what sort of action must be taken with the current contents of EJUTR's action log.

When an instantiation of this class is triggered, it displays a confirmation dialog to the user which determines which one of three actions the user desires. There are three possible actions that EJUTR can perform when the StopRecording class is triggered. The first action occurs when the user chooses to stop recording and copy the current contents of the log to the JUnit test case that was specified when the recording was started. The second action occurs when the user chooses to stop recording and discard the current

contents of the action log. The third and final action occurs when the user wishes to continue recording without discarding the current contents of the action log or writing those contents to a file.

If the first option is chosen by the user, then the current contents of the action log are written to the JUnit test case that was specified when the recording was started. This occurs by calling the dumpTo() method within the EJUTRFileWriter object with the name of the desired target method. The EBOBListener object is then detached from the Object Bench, and the contents of the EJUTRTargetFileInfo object are removed, as EJUTR is no longer recording anything.

If the second option is called by the user, the EBOBListener is removed from the Object Bench, and the contents of the EJUTRTargetFileInfo object are removed. Nothing is written to the JUnit test case specified when the recording was started, and

If the third option is chosen by the user, nothing happens. The state of the action log and the Object Bench does not change at all. The process of recording additional interactions with the Object Bench continues as if the "Stop Recording" menu item was never selected at all.

The RunTests class is not currently used within EJUTR. The class was initially designed to provide the functionality to easily execute JUnit tests from within EJUTR, but it was later determined to be unnecessary. It is already quite simple to execute JUnit tests through the menu items that are already provided by the JUnit plug-in itself.

4.1.3 edu.wpi.ejutr.debug

The edu.wpi.ejutr.debug package contains code representing objects that were used to debug the system, such as printing to the console or displaying debug pop-up dialogs.

The DebugDialog class allows other objects to print messages to the console. This was used while developing several components within EJUTR. The DebugDialog class also enables other objects to either enable or disable the debug printing behavior.

The TestFileWriter class was used while EJUTR was in development. It was inserted into the context menus of EJUTR and EBOB and explicitly triggered the

dumpTo() method of the EJUTRFileWriter object without needing to go through the normal steps in the execution of EJUTR, such as attaching/detaching the EBOBListener, or populating the action log with interactions. This class is not used within EJUTR during its normal execution process.

4.1.4 edu.wpi.ejutr.ebob_observer

The edu.wpi.ejutr.ebob_observer package contains the code responsible for interfacing with E-BOB through the use of the ObjectBenchListener interface. The package itself contains only one class, the EBOBListener, which implements that interface.

The EBOBActionListener implements the ObjectBenchListener interface found within E-BOB. This object gets physically attached to and detached from the Object Bench within E-BOB whenever a test recording is started or stopped (respectively.) The EBOBActionListener class is implemented as a singleton in order to guarantee that only one instance of this listener exists within EJUTR. It was necessary to utilize the singleton design pattern while implementing this class in order to make the attach() and detach() methods static, so that they could be called by different objects external to the edu.wpi.ejutr.ebob_observer package. It would not have made sense to have the EBOBListener instantiated when it was attached by the RecordNewTestMethod object, because it would then be difficult to utilize from a corresponding StopRecording object.

The ObjectBenchListener interface within E-BOB specifies that there are three methods that a class that wishes to behave as an ObjectBenchListener must implement. The first of those methods is the objectAdded() method. The objectAdded() method is triggered through E-BOB's listener interface whenever an object is instantiated within the Object Bench. Both the fully-qualified name of the object as well as the particular name of the instance of that object are passed to all listeners attached to the Object Bench. For example, if an Integer object named "int1" were added to the Object Bench, the data passed through the listener interface would be "java.lang.Integer" as the fully-qualified object name, and "int1" as the instance name of the object. This information is stored within EJUTR's action log, and is later written to a JUnit test file by EJUTR.

However, a weakness of the objectAdded() method is that it is triggered after the object was physically instantiated within E-BOB. By the time the method is called and the listeners are notified, there is no longer any information available within E-BOB that indicates which particular constructor was used when instantiating that object. In Java, it is possible for classes to contain several different constructor methods. Constructors are special methods that are responsible for physically instantiating an object, and generally constructor methods will set the initial contents of the constructed object to some particular state. If there are no constructors explicitly defined within a class, Java will utilize a "default constructor" is used, which merely instantiates the object as a subclass of Java's base java.lang.Object class. In this case, no class-specific initialization is performed. The problem is that, given some object, it is impossible to determine exactly which of the available constructors were used to create that object. Even the Java Reflection API is not able to provide that functionality. Therefore, EJUTR is not able to tell which constructor was used to instantiate a given object. Therefore, EJUTR assumes that the default constructor was used whenever an object is added to the Object Bench and the objectAdded() method is triggered. This works because the default constructor is always available for every Java object. Currently, however, there is no way to guarantee that the state of an object that was constructed within the Object Bench will match the state of an object within EJUTR's action log unless the default constructor is utilized, which will affect the outcome of tests that are recorded and "played back" later. Please see the "Recommendations and Future Work" section of this paper for our recommendations on how to correct this problem.

The second method within the ObjectBenchListener interface specified by E-BOB is the objectRemoved() call. This method is triggered whenever an object is removed from the Object Bench, which is analagous to a "delete" operation. EJUTR stores the fact that the object was removed within its internal action log.

The objectAdded() and objectRemoved() methods within the ObjectBenchListener interface were initially defined when work started on the EJUTR project. However, there was no way for E-BOB to notify its listeners when methods were called on objects within the Object Bench. Therefore, it was necessary to add a third method to this interface that would be triggered by E-BOB whenever it executed a

method within an object contained within the Object Bench. The new method is called methodCalled(), and it is triggered whenever a method is executed within E-BOB.

The methodCalled() method takes two types of objects as parameters, namely "MethodInvokeInfo" and "MethodInvokeParamInfo". These objects are constructed within E-BOB and contain information about the method that was called. The actual code for these objects is placed within the edu.wpi.ebob.model package (along with the code for the ObjectBenchListener interface.) The MethodInvokeInfo class contains various pieces of information about the physical method that was triggered within E-BOB, such as the fully-qualified object name of the object on which the method was called, and the instance name on which the method was called. The MethodInvokeParamInfo class contains information regarding the parameters (if any) that were utilized in the method call. The information contained within these classes is saved within EJUTR's internal action log for later test case generation.

After recieving the information about the specific method that was called within E-BOB, EJUTR then examines the return type of that method. If the return type is not "void", then the Assertion Wizard is displayed, which allows the user to specify if they would like to add a JUnit Assert call to EJUTR's internal action log. If the return type is "void", then no further action is taken.

4.1.5 edu.wpi.ejutr.file_operations

The edu.wpi.ejutr.file_operations package contains the code responsible for writing the contents of EJUTR's internal record to specified methods within JUnit test case files. The only class contained within this package is the EJUTRFileWriter class. The EJUTRFileWriter class simply iterates through the contents of EJUTR's internal action log and writes them to the JUnit test case that was specified when the recording was started. The EJUTRFileWriter class utilizes the to_String() method of each object within EJUTR's internal action log to generate syntactically correct Java code that is then written to the specified method and file. In addition, the EJUTRFileWriter class will check the import list within the JUnit test case that it is writing to. If the EJUTRFileWriter class detects that the junit.framework.Assert class is not imported,

EJUTR will automatically add the corresponding import to ensure that compilation errors that are caused by a missing JUnit import will never be introduced into an existing file.

4.1.6 edu.wpi.ejutr.logger

The edu.wpi.ejutr.logger package contains the code responsible for storing and retrieving the user's interactions with E-BOB. It was determined that EJUTR would initially need to provide the capability of storing four distinct types of E-BOB events: method calls, object additions, object removals, and assertions. The first three types of events correspond to events that are triggered through E-BOB's ObjectBenchListener interface, and the fourth event is triggered internally by EJUTR whenever the AssertionWizard is displayed and is successfully closed. The contents of the edu.wpi.ejutr.logger package were designed while keeping this fact in mind.

The core class of the edu.wpi.ejutr.logger package is the EBOBActionLogger. This class is responsible for physically maintaining a list of recorded actions within EJUTR, and for providing access to that internal list of actions to other classes (such as the EJUTRFileWriter class.) The EBOBActionLogger class is implemented as a singleton in order to ensure that there exists at most one internal action log within EJUTR. Therefore, the class provides methods for other objects to get handles to both the EBOBActionLogger object itself and the internal action log.

The EBOBActionLogger class utilizes a simple Java Vector to store the internal action log within EJUTR. In addition, the class provides an easy way for external objects to access the contents of the internal action log through the use of the iterator() method. The functionality to access specific elements within the internal action list is intentionally not provided. This behavior will help to ensure that elements of the internal action log will always be accessed in the order that they were recorded.

There are several methods within the EBOBActionLogger class that are called in order to actually create a record of an event within the internal log. These methods correspond to the four types of events that were identified when the requirements of the logger package were initially enumerated. The methods are logMethodCall(), logObjectAddition(), logObjectRemoval(), and logAssertion(). Each of these methods

take certain pieces of information as parameters and construct a specific type of object (which implements the EJUTRLogEntry interface.) The constructed object is then inserted into the internal action log. For example, the information passed into the EBOBActionLogger class for the logMethodCall() method corresponds to the name of the method called, the object on which it was called, and the list of parameters.

The type of object that is constructed for each of the four types of events is dependant on the event itself, but each of the objects that are physically added to EJUTR's internal action log all implement the EJUTRLogEntry interface. The most important component of the EJUTRLogEntry interface is the to_String() method, which is called by the EJUTRFileWriter class when the contents of the internal action log are written to a JUnit test case. The to_String() method provides a way for EJUTR translate from the internal storage object to a representation of that object within a properly-formatted JUnit test case file. Therefore, it is the responsibility of objects that implement the EJUTRLogEntry interface to implement the to_String() method in a fashion that returns a properly-formatted, syntactically correct String representing the object. In addition, the interface specifies numbers that uniquely identify each type of EJUTRLogEntry object. If it is ever necessary to add additional types of EJUTRLogEntry objects, the interface would need to be modified by adding additional unique numerical identifiers for each new type of object. These unique identifiers allow external objects that are iterating through the internal action list to easily identify the type of object within the internal action log without necessitating the use of either Java reflection or the “instanceof” operator.

The LogAssertion class implements the EJUTRLogEntry interface and is inserted into EJUTR's action log whenever an assertion is declared using EJUTR's Assertion Wizard. Currently, the only type of JUnit assertion that is recorded is Assert.assertTrue(), which evaluates the statement it is given in terms of being either true or false. However, every other type of Assert call can be translated into the assertTrue() call, so this isn't much of a limitation. In order to generate a String that represents an Assert call that is syntactically correct in terms of both Java and JUnit syntax (in terms of the assertTrue() call), several pieces of information are necessary, namely, the left-hand-side of the evaluation, the comparison operator, the right-hand-side of the evaluation, and whether or

not the entire evaluation should equate to being either true and false. These pieces of information are stored within the LogAssertion class, and when the to_String() method is triggered, the information is translated into syntactically correct Java syntax.

The LogMethodCall class implements the EJUTRLogEntry interface and is inserted into EJUTR's action log whenever the EBOBListener class is notified of a method call within E-BOB. The to_String() method generates a syntactically perfect method call using information such as the calling object, the name of the method being called, and string representations of each parameter necessary within the method call. For example, if the calling object was a Register object, the instance name of that object was "reg1", the method name called was "someMethod", and the parameters were some other object named "cash2" and an integer value of 5, the corresponding String that will be generated is "reg1.someMethod(cash2, 5);".

The LogObjectAddition class implements the EJUTRLogEntry interface and is inserted into EJUTR's action log whenever an object is instantiated within the Object Bench. The to_String() method generates a string representing syntactically correct Java code which will instantiate a new instance of the given object using the fully-qualified object name, and the instance name of the object. As explained above, the default constructor is used here while translating from the LogObjectAddition object to the corresponding Java code.

The LogObjectRemoval class implements the EJUTRLogEntry interface and is inserted into EJUTR's action log whenever an object is removed from the Object Bench. However, due to the nature of the Java garbage collector, there is no need for any real code to be generated and placed into a JUnit test case that deals with the removal of an object from the Object Bench. The user interface of E-BOB will ensure that no further calls using the object that was removed can occur, so there is no need for EJUTR to take any special steps when dealing with the removal of an object; it is enough that the garbage collector within the Java run-time will deal with objects that are no longer used.

The EJUTRTargetFileInfo class is a singleton class which contains information about the current recording session. If EJUTR is not currently in the process of recording, then the fields within the singleton object are empty. The fields within the EJUTRTargetFileInfo object are populated with information whenever a recording

session begins, within the RecordNewTestMethod class in the edu.wpi.ejutr.actions package. The same fields are cleared in the StopRecording method in the edu.wpi.ejutr.actions package, after the contents of EJUTR's action log have been written to the correct JUnit test case.

4.1.7 edu.wpi.ejutr.testobjects

The edu.wpi.ejutr.testobjects package contains several objects that were used while testing the functionality of EJUTR and the communications interface between EJUTR and E-BOB. These objects are not used when EJUTR executes normally.

4.1.8 edu.wpi.ejutr.wizards

The edu.wpi.ejutr.wizards package contains the physical wizards that are displayed to the user, such as the Assertion Wizard, which allows the user to specify whether or not to include a JUnit assertion, and the Record New Test Method wizard, which allows users to designate where they would like the recorded test methods to be written to.

Both of the EJUTR wizards extend the Wizard class, which is itself an implementation of the IWizard interface, and override only the necessary methods. The wizard page class represents one "step" in a wizard. Both EJUTR wizards contain one wizard page. The wizard page uses functionality from the Abstract Windowing Toolkit (AWT) and JFace packages to display widgets and dialogs. Both of these packages are part of the Eclipse framework. Methods were added the wizard classes so the user inputs can be accessed after the wizards are displayed.

4.1.8.1 The Record New Test Method Wizard

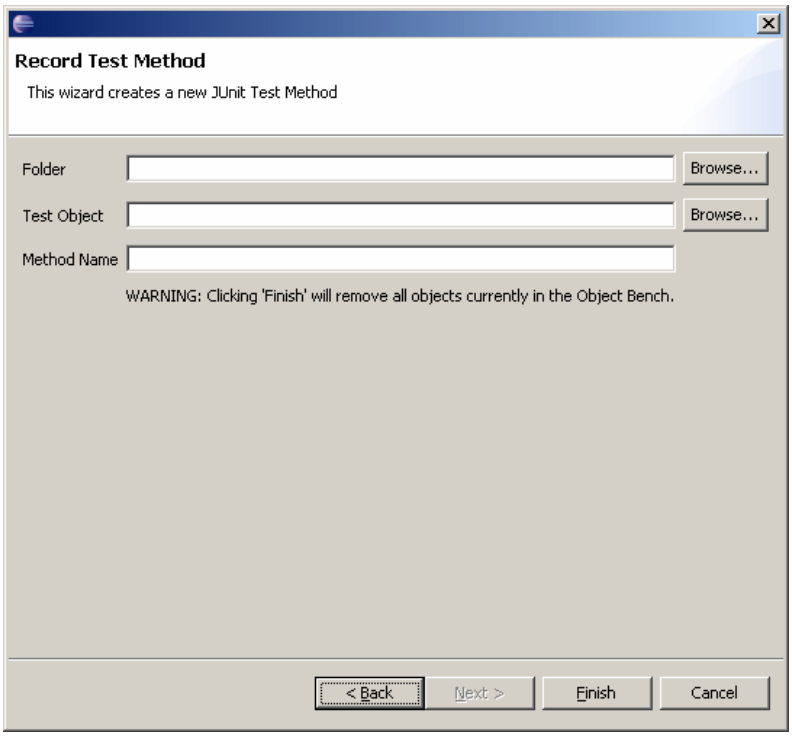

Figure 4 - The Record New Test Wizard

The RecordNewTestMethod wizard is displayed to the user and collects information about which folder, JUnit test case file, and test method to write the contents of the EJUTR action log to. The wizard is physically instantiated within the RecordNewTestMethod action within the actions package.

The Record New Test Method wizard within EJUTR is registered with Eclipse thorough the org.eclipse.ui.newWizards point. The newWizards extension point can be used to tell Eclipse about the existence of a new user-defined wizard. When it is used Eclipse will allow the user to run the wizard any time by clicking on "File->New->Other…". A dialog will appear showing a list of all wizards that have registered with Eclipse. Registering the Record New Test Method Wizard with Eclipse makes EJUTR easier to use because the wizard is listed in a standard place where the user can find it.

The "Folder" text field lets the user enter the folder where the test case resides. The "Browse" button to the right of the text field will bring up a dialog that will let users select a folder within any currently open project. Once the selection is made and the

"OK" button is pressed the "Folder" text field is automatically filled in. If the user types in an invalid folder, a corresponding error message will appear at the top of the wizard, and the "Finish" button will become disabled until an acceptable value is entered into the text field.

The "Test Object" text field lets the user fill in the name of the JUnit test case file that the new method will be written to. The "Browse" button will bring up a dialog that will let the user filer the files in the folder entered into the folder text field. Once a file is selected and the "OK" button is pressed, the text field will automatically be filled in with the name of the file. The wizard checks to make sure that the selected file extends the TestCase class. If it does not an error message will appear at the top of the wizard.

The "Method Name" text field indicates the name of the new test method. The wizard check to make sure that the name starts with the string "test" and that the provided method name is a valid. If it is not, an error is displayed within the wizard, informing the user of the need to correct the name of the method. When the error message is displayed, the "Finish" button is disabled, so that execution cannot proceed using an invalid method name.

4.1.8.2 The EJUTR Assertion Wizard

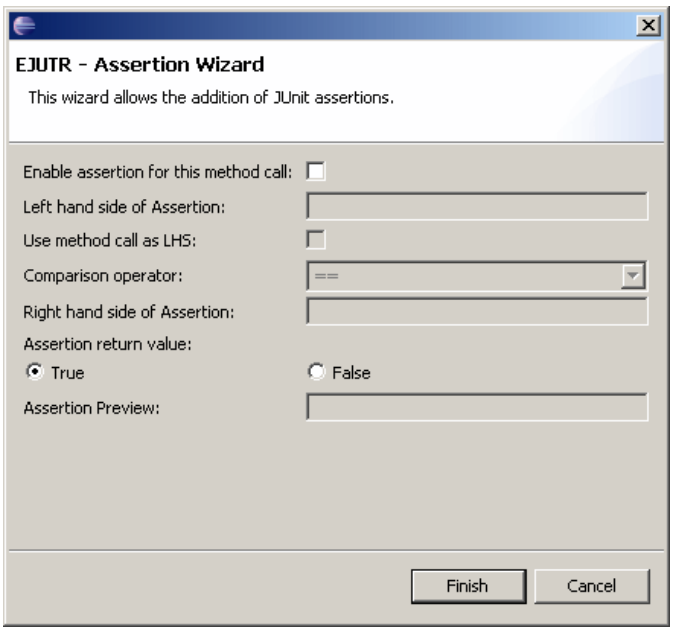

Figure 5 - The Assertion Wizard

The Assertion Wizard is displayed to the user whenever EJUTR is recording and a non-void method call is executed within E-BOB. Its purpose is to collect information about whether or not the user desires a specific JUnit assertion call to be made as a result of the method call that was previously executed.

The UI elements of this wizard lets the user compare two values in an assert call. The two values will be joined by a comparison operator, and the overall expression will be evaluated in terms of being either true or false. Checking the "Enable assertion for this method call" check box will add an assertion to EJUTR's action log; if the check box is unchecked then nothing will be written to EJUTR's action log when the 'Finish' button is pressed. The "Left hand side of Assertion" text field indicates the left-hand-side of the boolean expression that will be inserted into the assertion call. The "Use method call as LHS" check box will cause the return value of the method that was just executed within E-BOB to be used as the LHS of the boolean JUnit expression. The "Comparison operator" combo box lets the user choose between standard comparison operators (==, !=, >, >=, <, <=) and select the appropriate one that will be used to compare the two values. The "Right hand side of Assertion" text field is used to record the expected return value

of the method that is being tested. The “Assertion return value” radio buttons are used to indicate whether the boolean expression is expected to be true or false. When the “Finish” button is clicked, an assertion is created using the information entered into the wizard. The assertion is then added to EJUTR’s action log.

5 Conclusions

The first conclusion that we can draw from the EJUTR project is that the Eclipse framework is both very powerful and very complex. We were able to implement the EJUTR plug-in itself with somewhere between 950 and 1000 lines of code, which is remarkable when one considers all of the things that the plug-in does and what it interfaces with, however, the line-count figure does not include the code that was modified within E-BOB. There is a lot of functionality within the Eclipse framework, which provided us (as Eclipse plug-in developers) the opportunity to spend more time thinking about the behavior and design of our particular plug-in as opposed to "re-inventing the wheel" and generating code to provide some functionality that is already included within the Eclipse framework. This is very powerful as a development tool, but seems to generate a new, unique set of development problems. Instead of needing to figure out how to build a particular tool to accomplish some task, Eclipse developers are instead given what amounts to a gigantic toolbox and an online instruction manual explaining the contents of that toolbox. There is probably some piece of code within the Eclipse framework to solve almost any development issue, but the problem becomes finding that particular piece of code or that particular tool within the massive amount of functionality that is provided by the Eclipse framework. Fortunately, there exists an entire library of books that detail every facet of the Eclipse framework, but as with anything else, the more time one spends utilizing and developing code with the framework, the better acquainted one will become with all of the individual tools and pieces of functionality that the platform provides.

As powerful as the Eclipse framework is, we have just barely scratched the surface of it. While the EJUTR project was a good way to get involved with Eclipse plug-in development, there are many more ways with which to work within the Eclipse framework that we have not yet attempted. EJUTR interfaced with several extension points within Eclipse such as the Wizards extension points and the right-click context menu extension points, but there exist many more such extension points that could be used to provide additional functionality to the Eclipse platform. It may be worthwhile in

the future to review those additional extension points to see what additional functionality EJUTR and E-BOB could provide to the Eclipse platform.

The Eclipse-based Object Bench was a very well-received plug-in within the Eclipse community, and we hope that with the inclusion of the EJUTR project E-BOB will become even more valuable to those developers and students that currently utilize it.

6 Recommendations and Future Work

The Eclipse JUnit Test Recorder project accomplished a lot of tasks, but there is still plenty of work to be done in the future. The contents of this chapter will detail additional work that the EJUTR development team feels could be done in order to increase the usability of the EJUTR plug-in.

6.1 The Constructor Issue

As explained earlier, there is a problem with the ObjectBenchListener interface defined within E-BOB that occurs when objects are instantiated using a constructor other than the default constructor. The problem is that the ObjectBenchListener class was originally designed to only signal the fact that an object was added to the Object Bench, and as such there is no information available within the interface other than a handle to the instantiated object. Unfortunately, there exists no way within Java to determine which particular constructor was used to instantiate a given object when all that is given is the object itself. Therefore, it is currently impossible for EJUTR to determine which of many possible constructors were used when an object is added to the Object Bench, and therefore it can only assume that the default constructor was used. Each individual constructor methods can have different behavior from the other available constructor methods, and therefore it is both possible and likely that the state of the instantiated object will differ markedly depending on which constructor was initially used. Therefore, if a constructor other than the default constructor is utilized within E-BOB, EJUTR will record the action as having instantiated the same object using the default constructor. This may cause disparities between the state of the object in the Object Bench and the state of the object within EJUTR's internal action log, which is not a desirable situation in terms of generating accurate JUnit test code.

The underlying problem here is more complex than it initially seems, however. When the objectAdded event is fired within E-BOB and a new object is instantiated and added to the Object Bench, the object has already been constructed somewhere further up the stack. The information about the particular method invocations that were utilized

while instantiating that object (ie. the particular constructor that was utilized) is no longer available in any form. Therefore, fixing this problem will be more complex than merely altering the ObjectBenchListener interface to pass additional information to its implementers, as the information simply no longer exists. It would require making significant changes to the order that methods are called within E-BOB, and when making such changes there always exists a risk of disabling some other piece of functionality unexpectedly.

While reflecting upon this problem, it became apparent that perhaps the best way to implement the E-BOB to EJUTR interface was to not modify E-BOB's code at all. Instead, it may be a better idea to extend the ObjectBenchListener interface into a new class and add additional functionality to the extension. Another idea was to create an entirely new collection of listener objects as well as the corresponding functionality to drive those objects. The new listener list would behave similar to the current implementation of the ObjectBenchListener interface, but it would allow for more significant changes to be made without fear of breaking some other functionality within E-BOB. E-BOB actually uses the ObjectBenchListener interface within itself to keep track of certain types of events, which was always a limiting factor when considering how much of the interface it was possible to modify. The new listener list could specifically provide the information that EJUTR requires rather than forcing EJUTR to parse certain pieces of information copied from the internal structures used within E-BOB. In addition, it would be possible to add the calls to trigger the listener behavior within E-BOB in positions within the code base that are advantageous to the type of behavior that EJUTR needs the listener interface to provide, rather than what E-BOB requires.

6.2 Additional Types of Assertions

One of the major limitations of EJUTR is that it will only allow the user to add a single type of JUnit assertion call, namely, the Assert.assertTrue() call. This assertion checks to see if the expression passed to the call as a parameter evaluates to a Boolean value of true or not. While all other JUnit assertions can be translated into an assertTrue()

call, it may be desirable for users to be able to select specific types of assertion calls rather than forcing them to always use the assertTrue() call.

If additional types of assertions were to be added, it would be necessary to add additional objects that implement the EJUTRLogEntry interface to the edu.wpi.ejutr.logger package. The additional objects would each represent a specific type of assertion call. Additional unique numerical identifiers would need to be added to the EJUTRLogEntry interface, as well. The EBOBActionLogger class would need additional methods to handle the additional types of assertion objects that could be added to EJUTR's internal action log. A second option would be to modify only the LogAssertion class, and instead add logic to the class that would allow EJUTR to specify individual types of assertion calls instead of assuming everything is an Assert.assertTrue() call like it does currently. Of course, the Assertion Wizard will need to be modified to handle the additional types of assertion calls.

6.3 More Comparison Operators

EJUTR currently provides various types of comparison operators within the Assertion Wizard in the form of a drop-down combo box. However, by default, only the numerical comparison operators are shown. It may be desirable in the future for EJUTR to examine the objects that the method call which triggered the Assertion Wizard was invoked on, and provide additional comparison operators within the drop-down combo box. These additional comparison operators could be Boolean methods within the objects, for example.

6.4 UI Component Usability Studies

The wizards and various user interface components within EJUTR were not designed with usability in mind. That does not mean that they are difficult to use, but merely that the UI components were never tested in terms of their overall usability. It may be desirable in the future to evaluate how difficult it is for an average user to utilize

EJUTR. In particular, the Assertion Wizard needs to be examined. This could possibly be accomplished by conducting a series of usability studies.

6.5 External Help Plug-in

E-BOB is accompanied by a full-fledged Help plug-in that explains in great detail exactly how to utilize the various components of E-BOB. It might be a good idea in the future to either include a similar help plug-in for EJUTR, or to extend the E-BOB help plug-in and add sections detailing how to use EJUTR. However, this seems to be dependant on how intertwined EJUTR and E-BOB eventually become, a factor which is unknown at this point. It is hoped that EJUTR will eventually become part of the standard E-BOB distribution, but that has not been decided yet.

6.6 Localization

E-BOB utilizes a method of storing strings for its graphical elements within a separate file, which are read in at run-time and used to write text onto specific graphical widgets. This approach has many advantages, one of which is that it allows users to easily translate the plug-in GUI into different languages by simply changing the corresponding value of the string in the text file, rather than modifying the code directly. If E-BOB and EJUTR were to be combined, it would be necessary for EJUTR to implement a similar scheme in order to keep things consistent.

7 Bibliography/Works Cited

About Us: History of Eclipse. 2006. The Eclipse Foundation. 18 Apr. 2006. <http://www.eclipse.org/org/#history>.

Clayberg, Eric and Dan Rubel. Eclipse: Building Commerical-Quality Eclipse Plug-ins. Ed. Erich Gamma, Lee Nackman, and John Wiegand. Boston: Addison-Wesley, 2004.

D'Anjou, Jim, Scott Fairbrother, Dan Kehn, et al. The Java Developer's Guide to Eclipse. Boston: Addison-Wesley, 2005.

Eclipse IDE Sees Strong Growth Worldwide. 24 May 2004. ADVISOR. 17 Apr. 2006. <http://advisor.com/doc/14139>.

Eclipse Platform Technical Overview. Feb. 2003. IBM Corporation. 17 Apr. 2006. <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.

Gamma, Erich and Kent Beck. Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Ed. Erich Gamma, Lee Nackman, and John Wiegand. Boston: Addison-Wesley, 2004.

Horstmann, Cay. Object Oriented Design & Patterns. Hoboken: John Wiley & Sons, Inc., 2004.

Morley, Liam and Justin Braga. "The Eclipse-Based Object Bench." Major Qualifying Project Report. Worcester Polytechnic Institute, 2005.

Appendix A: EJUTR Installation Instructions

This appendix explains how to install the EJUTR plug-in. Like many Eclipse plug-ins, EJUTR depends on other plug-ins to function properly. Several dependencies must be satisfied before the EJUTR can be used. EJUTR requires three external plug-ins. This number does not include the JDT and JUnit plug-ins, which are installed by default in most Eclipse configurations.

Installing E-BOB.

The EJUTR plug-in requires the E-BOB plug-in. The official E-BOB site is http://ebob.sourceforge.net, and the original version of the plug-in can be downloaded from there.

However, the E-BOB plug-in was modified during the development of EJUTR. The modifications have not been yet been committed to the official E-BOB project. The modified E-BOB plug-in must be used, or EJUTR will be unable to function. The modified E-BOB plug-in can be obtained from http://users.wpi.edu/~bigben/ejutr/

The E-BOB plug-in itself has two dependencies.

Java EMF Model (JEM)

The JEM plug-in is part of the Visual Editor project. E-BOB relies on the JEM plug-in in order to execute methods and instantiate objects on a separate instance of the Java Virtual Machine.

The dependency can be fulfilled by either installing the JEM runtime plug-in or the entire Visual Editor project.

Make sure to download the correct version of the JEM plug-in for your individual version of Eclipse.

The project page for the visual editor is:
http://www.eclipse.org/vep/WebContent/main.php

Eclipse Modeling Framework (EMF)

JEM requires the Eclipse Modeling Framework in order to operate properly. This plug-in can be obtained from http://www.eclipse.org/emf/

Installing EJUTR

The EJUTR plug-in and the modified E-BOB plug-in can be found at http://users.wpi.edu/~bigben/ejutr/ . Specific installation instructions can be found on that site.

Appendix B: EJUTR Usage Walkthrough

The following is a walkthrough of recording a simple test case with E-BOB and EJUTR. This walkthrough assumes that the Java perspective is being used within Eclipse (it's the default way that Eclipse appears to the user.) The example creates a test for a simple Register class. The class has three methods. One method allows the user to set the amount due, one method allows the user to indicate how much the customer has paid, and one method calculates the change due to the customer.

Step 1 - Make the Test Case file

The first step is to create a JUnit test case. This can be done in several ways, the easiest of which is to use the "New JUnit Test Case" button on the main toolbar, or in the right-click context menu of the Package Explorer.

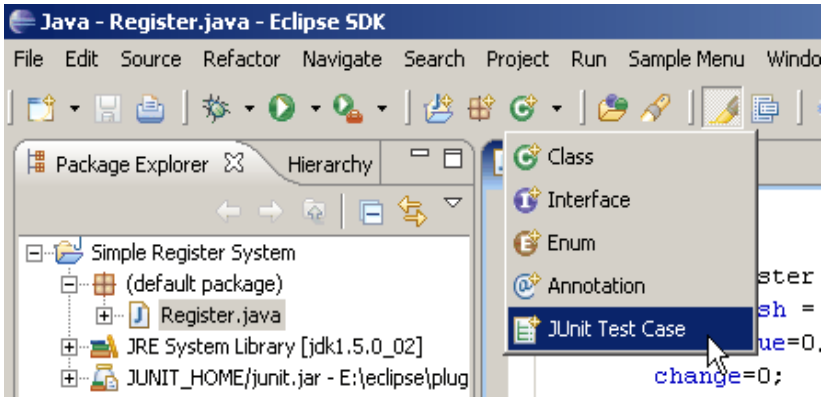

Figure 6 - Creating a JUnit Test Case

If the JUnit library (junit.jar) does not currently exist within the project's build path, a dialog will appear asking you if you wish to add it. Add the libraries. This will bring up the New JUnit Test Case Wizard.

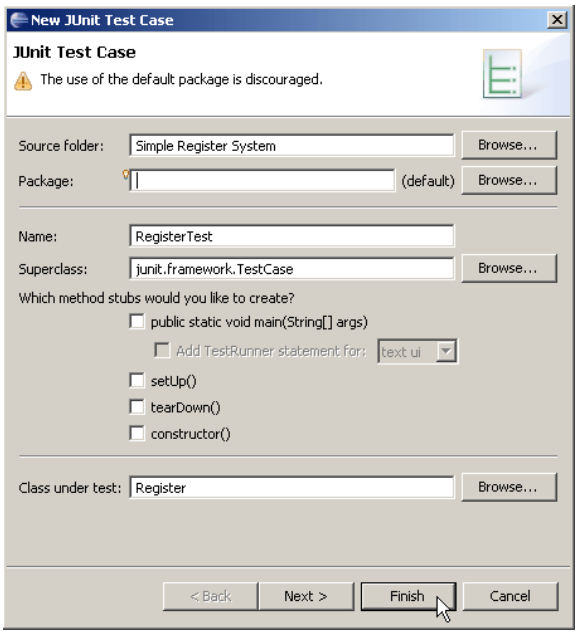

Figure 7 - The JUnit Test Case creation wizard

This wizard gives you many options, but for the purpose of this walkthrough most will be left the way they are. Specify the destination for the new test case file by indicating the desired source folder and package. Sometimes tests are stored in a separate package, or even a separate project, but in this case the new test case in going in the same package as the class that is going to be tested. The name field indicates the desired name of the new test case class. It has been set to RegisterTest. Clicking 'Finish' will create a new JUnit test case file.

Step 2 - Start the Recording

After the JUnit test case file has been created and the JUnit library has been added to the build path, recording can begin. There are several ways to start recording a new test method. The easiest way is to use the E-BOB right-click context menu. To begin recording a new JUnit test case, right click on the E-BOB View, select "Record New Test Method" in the "JUnit Test Recorder" submenu.

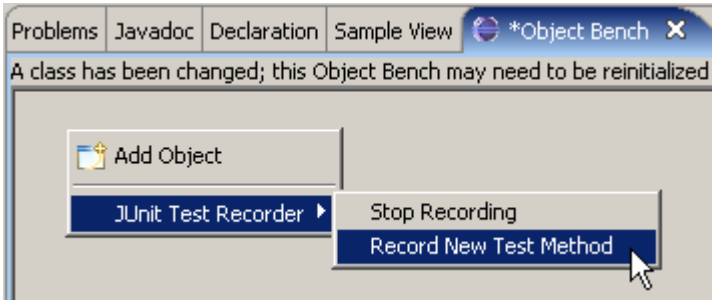

Figure 8 - E-BOB's right-click context menu

The “Record New Test Method” wizard will appear. The ‘Folder’ and ‘Test Object’ fields are used to indicate the location of the test case that the test method will be recorded to. The ‘Folder’ field indicates the folder that the test case resides in, and the Test Object Field indicates the test case.

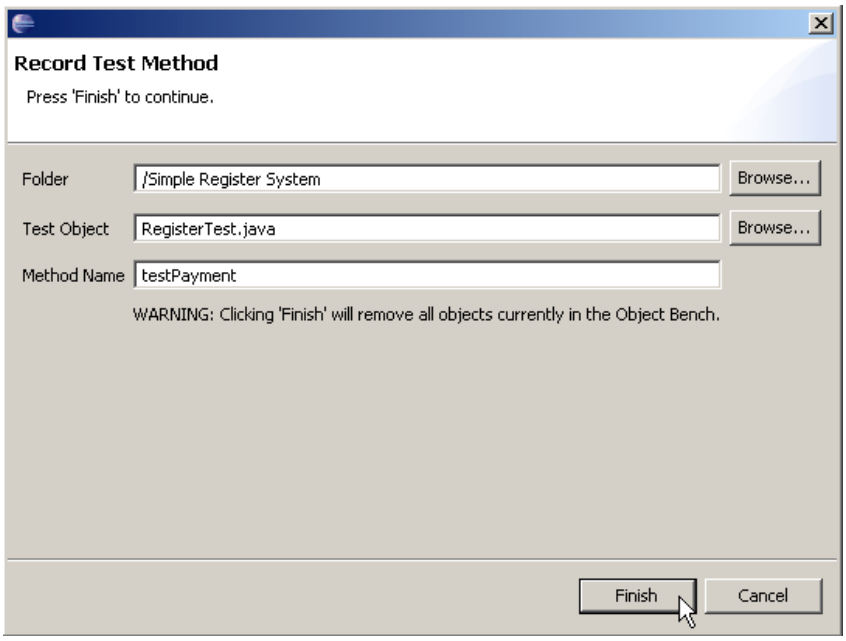

Figure 9 - The Record New Test Case wizard

After the ‘Finish’ button is clicked, all interactions with E-BOB will be recorded by EJUTR. The contents of the Object Bench will be emptied, and EJUTR and E-BOB will both start from scratch.

Step 3 – Utilize E-BOB

The next step is to utilize E-BOB to instantiate objects, call methods, and set variable fields within those objects. E-BOB provides multiple ways of doing this.

To manipulate an object, right-click on its representation within the Object Bench and use the context menus to call methods. In this example, the setAmountDue() method is used to set the amount due to 20. The customer payment is set to 15 using the pay() method.

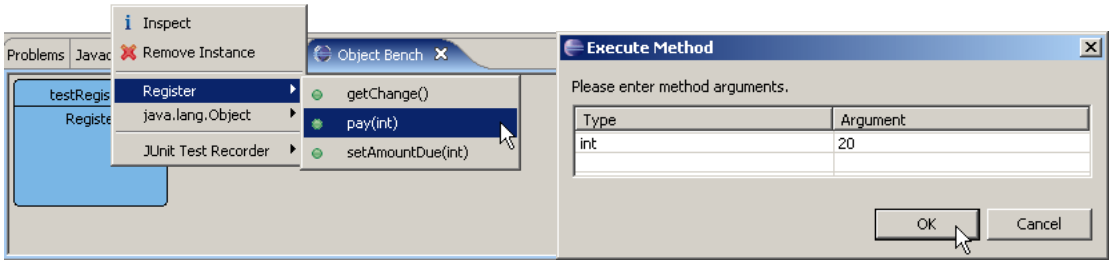

Figure 10 - Executing the 'pay' Method

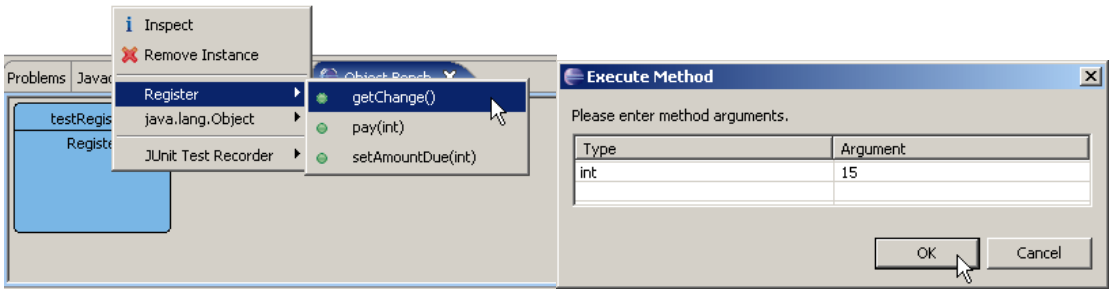

Figure 11 - Executing the 'getChange' Method

Step 4 – Generate Assertions

The next step is to set up a JUnit assertion call. Use E-BOB to manipulate the objects within the Object Bench to some desired state. Then, utilize the Assertion Wizard to generate a JUnit assertion call. In this walkthrough only one test will be created. In practice multiple tests are created for each test method.

Each test involves a method with a return value. The actual return value of the method is compared to the expected return value of the method. In this walkthrough the return value of the getChange() method will be tested. When E-BOB is used to call the getChange() method the EJUTR plug-in will display the Assertion Wizard.

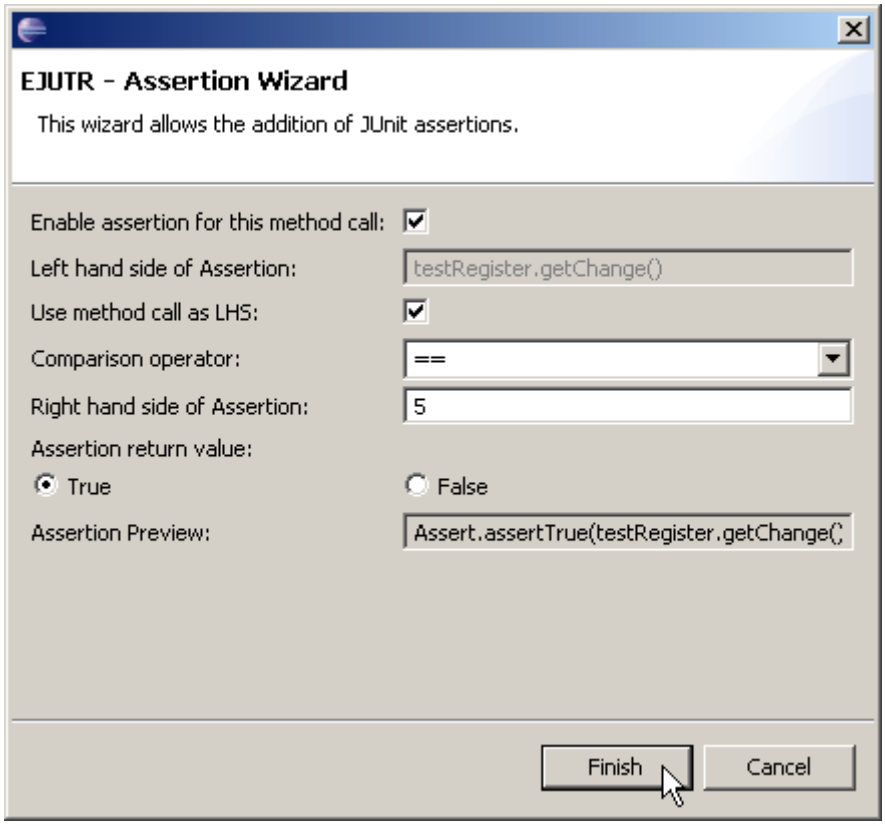

Figure 12 - The EJUTR Assertion Wizard

Click the “Enable assertion for this method call” check box.

Click the “Use method call as LHS” check box. This set the left had side of the comparison to the return value of the method. In this walkthrough the == operator is used to compare the two test values. The combo box provides other operators that can be used in different situations. The right had side of the test expression is set to the expected return value of 5. We want to assert that the value is equal to the excepted return value of 5 so the assertion return value is set to true.

Step 5 - Stop the recording

After the desired tests have been completed it is time to end the recording. Once again there are multiple ways of doing this but the easiest is right click on the Object Bench and chose “Stop Recording” EJUTR submenu and choose “Yes – Stop and Save Recording.”

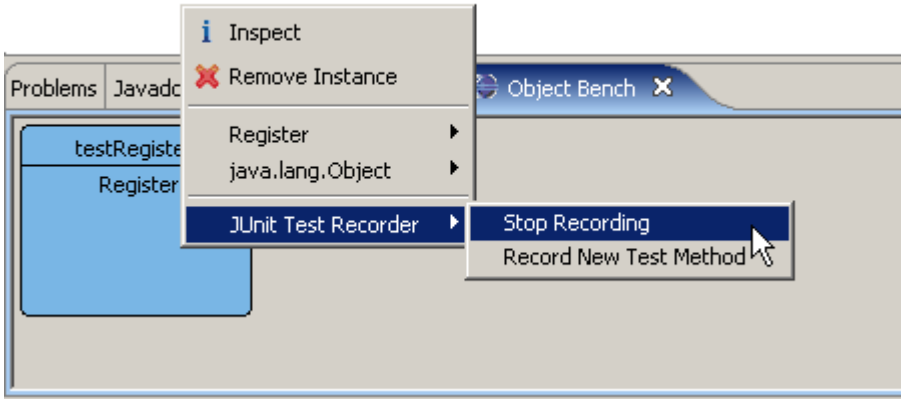

Figure 13 - Stopping EJUTR

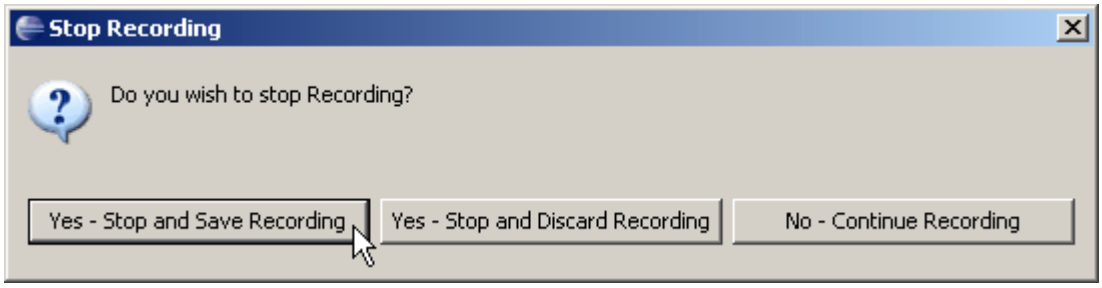

Figure 14 - The "Stop Recording" Dialog

Step 6 - The test method is generated

Once recording has stopped, you will be presented with three options. The “Yes – Stop and Save Recording” option will allow you to stop recording and copy the current contents of EJUTR’s action log to the designated JUnit test case. The “Yes – Stop and Discard Recording” option will allow you to stop recording and discard the current

contents of EJUTR's action log, which is useful if you want to start over. The third option, "No – Continue Recording" allows you to exit out of this dialog box and continue recording as if you had never clicked the "Stop Recording" button in the first place.

In this example, we click the "Yes – Stop and Save Recording" button. As a result, the following code is generated and placed within the designated JUnit test case.

```
import junit.framework.TestCase;
import junit.framework.Assert;

public class RegisterTest extends TestCase {

        public void testPayment() {
        // object addition
        Register testRegister = new Register();
        // method call
        testRegister.setAmountDue(15);
        // method call
        testRegister.pay(20);
        // method call
        testRegister.getChange();
        Assert.assertTrue(testRegister.getChange()==5);
        }
}
```

This concludes the EJUTR walkthrough.