Automatic JUnit Creation Tool

An Exploration in High Level Process Driven Automatic Test Case Creation

A Senior Project presented to
the Faculty of the Computer Science Department
California Polytechnic State University, San Luis Obispo

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Science in Software Engineering

By William Whitney
Advised by Dr. John Dalbey
June, 2010

1 CONTENTS

2 ABSTRACT

Many software developers do not enjoy writing unit test code. Often their excuses range from testing is slow to testing is hard. Yet perhaps test derivation has to be neither. The aim of this senior project is to examine the current state of unit test creation for the Java programming language. In particular, inefficiencies with the JUnit test framework regarding test derivation are analyzed. Ultimately, a JUnit test creation tool is created that provides a high-level process for test derivation.

3 INTRODUCTION

The Automatic JUnit Creation Tool rose out of a desire to produce a senior project that not only would benefit its author but potentially other developers as well. In order, to meet this objective the initial brain storming sessions revolved around programming. In particular, the portions of programming that are undesirable. The ideas examined included enhancing IDE auto complete features, code formatting, and revision control. Yet the problem of testing source code stood out above the rest.

4 WHAT IS WRONG WITH TESTING?

In order, to answer what is wrong with software testing we must first narrow the problem. The first narrowing step was the selection of the Java programming language. We selected the Java language because it is popular and has wide tool support. If a solution exists that makes testing easier chances are it is implementation in Java. Yet even after selecting a language there are still many types of testing. Once again, we had to narrow the problem by selecting unit testing as the primary focus. We made this choice because unit testing is a very common developer activity where system and integration testing happens more sporadically. Amazingly, unit testing in Java is still a wide topic of discussion. We choose to narrow unit testing even more by constraining ourselves to the JUnit framework. With all these constraints in place, we then examined the problems associated with JUnit test creation in Java.

4.1 HEAVY EMPHASIS ON CODING RATHER THAN TESTING

One of the first problems we observed with the JUnit testing framework is that it heavily relies on coding [1]. This means developers must do three things while trying to write a unit test. First, they must look at their source code; second, they must think about the testing methodology they want to apply, finally they must think about how they are going to code the JUnit test. This process is cognitively demanding on the programmer because they have to think about three things at once. When in reality it would be much nicer if programmers could simply focus on the code they are testing and the testing technique they are using.

4.2 UNPREDICTABLE TEST DERIVATION PROCESS

One of the next problems discovered with JUnit testing is that developers are simply provided a set of assert statements [2]. This leads each developer to devise his or her own test creation process. A good test derivation process would help developers quickly create test cases that are correct and consistent. Ideally, a tool would exist that fulfills this requirement. The tests derived by this tool would guarantee the application of boundary value analysis, branch and path testing. This minimum guarantee would assure the tool creates rigorous tests. No longer would unit test quality depend on the individual developer but rather on the quality of the tool.

4.3 SLOW TEST CREATION

The last major problem identified with the test creation process is that it is slow [3]. This complaint seems mainly to arise out of the fact that each developer must relearn the JUnit testing framework and reinvent the process they are going to use each time they make a JUnit test. Writing tests in the JUnit framework also requires developers to write code, which is a labor-intensive task that goes far beyond checking test inputs and outputs. Ideally, developers would have a tool with a graphical user interface that allowed them to quickly open their Java source file and derive a test for it without having to worry about the JUnit testing framework.

5 IDEAL SOLUTION FOR JAVA JUNIT TESTING

The next step undertaken was the definition of an ideal tool. We knew from the analysis above that the JUnit testing framework is overly reliant on coding, provides very little implicit process and is time consuming to use. Our ideal criterion not only aims to eliminate these weak spots but also provide developers wide access to the tool.

5.1 FIND METHOD TEST BRANCHES AUTOMATICALLY

The ideal tool must find all branches for each method within the target Java source file. This requirement will ensure that at a minimum each test created by the tool is minimally rigorous. Also finding branches can be a boring and redundant task for developers. If the ideal tool automatically found each execution branch then developers are able to perform very little work and still test each branch.

5.2 PROVIDE GUI WIZARD TO INPUT TEST VALUES

The ideal tool must provide a GUI to input test values and specify expected results. A GUI based test derivation tool would shield developers from the JUnit testing framework details. The GUI will also enable developers to work at the problem domain level of abstraction. This will reduce cognitive load since developers will no longer have to worry about coding the test harness at the same time they are deriving the test.

5.3 GENERATE TESTS AUTOMATICALLY

The ideal tool must be able to generate JUnit tests from the provided input and expected output defined in the graphical user interface. This feature will benefit developers by automatically generating JUnit tests that are modular, well documented and correct. This feature will also save developers time because one test input and expected result will automatically transform into several lines of JUnit test code.

5.4 PROVIDE CROSS PLATFORM OPERATION

The ideal tool should be cross platform. Developers often work in many different environments so any tool they use should travel with them. In addition, the Java programming language is cross platform thus the tool should work wherever the code does.

5.5 OPEN SOURCE & FREE

Since developers will likely use this tool as a part of a larger development process, it is imperative that they are able to improve it. In addition, the underlying JUnit framework is open source so any tool that aims to improve the quality of testing for everyone should share a similar license.

6 BACKGROUND RESEARCH

6.1 RANDOOP

The first solution we found in our search was Randoop.

This tool can:

- Generate unit tests
- Captures behavior of existing code
- Produce random test data
- Automatically execute tests

While this tool does automatically create JUnit tests, it only does so to capture the existing behavior of your code. This might be useful if you have a large body of legacy code you need to get under test. Yet it does not provide the process driven approach we require. In particular, we need a tool that provides a GUI wizard for test input values and one that can find method branches automatically.

Randoop's website: `http://code.google.com/p/randoop/`

6.2 JTEST

The next tool we examined was JTest by Parasoft.

This tool can:

- Generate automatic unit tests to create a regression baseline
- Generate test cases for corner cases in your code that might cause errors
- Automatically generate test case stubs

This tool also provides a feature called test case "tracing". This feature automatically derives test cases for your classes, provides a GUI to edit the input values, and expected result. Yet this tool is not free or open source thus, it does not fully meet our criteria. We need a JUnit testing solution that allows free distribution and developer modification.

JTest's website: `http://www.parasoft.com/jsp/products/jtest.jsp`

6.3 COVIEW

The final tool considered was CoView.

This tool can:

- Create JUnit Tests
- Create Mock Objects
- Measure Path and Branch Coverage

This tool does help developers create tests that cover test paths and branches. Yet the tool is not completely free and is not open source. While this tool meets many of the required criteria it still does not fully raise the level of abstraction the programmer works at. Manual editing by programmers is still required to customize the tests generated by CoView.

CoView's website: `http://www.codign.com/`

6.4 BACKGROUND RESEARCH CONCLUSION

Overall, our search for existing technologies failed to turn up any solutions that adequately met our needs. The closest solution is CoView however, as mentioned above this tool is still not completely free and does not fully raise the level of abstraction for test derivation. The lack of a good existing tool lead us to the creation of our tool "The Automatic JUnit Creation Tool" this tool aims to closely model our ideal tool and reduce the problems found in the JUnit test framework defined above.

7 DEVELOPMENT PROCESS

The first step taken in the development of this tool was process selection. We selected a spiral development strategy with an incremental delivery plan. We made this choice because of the exploratory nature of the new tool. At the time, we were attempting to design something we had never attempted before. Thus, we wanted a process that focused on both exploring the problem domain as well as producing verifiable results. The spiral portion allowed us to add new requirements during the course of the quarter while the incremental delivery meant we were able to verify existing work easily.

7.1 CREATION OF A STORYBOARD AS REQUIREMENTS

We discussed many of the project requirements verbally during our senior project meetings. The domain of test tool creation was a dive into uncharted territory. We needed to capture requirements however; we did not want to use the traditional SRS format. Instead, we needed a more visual and concrete way of capturing requirements. We selected the storyboard method. With this method, we were able to visualize how the tool was supposed to work and run through all the different user scenarios. If we found something that needed to be reworked we simply took out a pen and made the changes right on storyboard. Appendix A contains our initial storyboard document.

7.2 CREATION OF INITIAL USER GUIDE AS REQUIREMENTS

As a way to refine the requirements a user guide was create for the then nonexistent tool. Once we had the user guide, we proceeded with thought experiments to convince ourselves that the tool really would work. This activity was very successful because we performed much of the intense development work in the storyboard stage. The user guide simply spelled out more explicitly many of the requirements that were implicit with the storyboard. Our initial user guide is contained in appendix B.

7.3 CREATION OF SCHEDULE

All the work on the Automatic JUnit Creation Tool took place as a part of a schedule. This was an amazing feat because we were using a spiral development model with incremental delivery. This required that the schedule define operations from a high level. As you can see from the schedule below we used big picture terms like "Finalize design". This allowed us to iterate over the requirements knowing that during week 5 of winter quarter we had to deliver the artifact. Ultimately, this schedule worked very well. We defined it during week 2 of winter quarter and deviated very little from it by the end of spring quarter.

Winter 2010	**Spring 2010**
Week 2 Jan 11-17 • Create Schedule • Storyboard initial requirements • Define Initial non-functional requirements • Define a process Week 3 Jan 18 - 24 • Revise and finalize user manual requirements • Non-functional Requirements • Product Information Version 1 Week 4 Jan 25 - 31 • Create initial design Week 5 Feb 1 - 7 • Finalize design Week 6 Feb 8 - 14 • Code first feature set Week 7 Feb 15 - 21 • Code second feature set Week 8 Feb 22 - 28 • Code third feature set Week 9 Mar 1 - 7 • Test • Solicit developer feedback Week 10 Mar 9 - 14 (Product Release) • Launch as open source project • Create product distribution site	Week 1 Mar 29 - 4 Apr • Developer Getting Started Guide Week 2 Apr 5 - 11 • Solicit user feedback for enhancements Week 3 Apr 12 - 18 • Refine feedback into requirements Week 4 Apr 19 - 25 • Recruit developers who want to help implement requirements Week 5 Apr 26 - May 2 • Implement requirements Week 6 May 3 - May 9 • Paper Outline Week 7 May 10 - May 16 • Second Paper Outline Week 8 May 17 - May 23 • First Paper Draft Week 9 May 24 - May 30 • Second Paper Draft Week 10 May 31 - Jun 6 • Submit material to library and CSC department

Figure 1: Initial Schedule Created Winter Quarter

7.4 ARCHITECTURE DESIGN DOCUMENT

After we had, the initial requirements defined and schedule laid out we proceeded to create a minimal architecture and design document. This document was minimal because we knew many of the requirements would evolve. Yet we still wanted a basic design in place that would make future requirements easy to implement. We defined the interactions between the Java grammar parser and GUI elements defined in our storyboard. Our architecture document is located in appendix C.

7.5 CODING

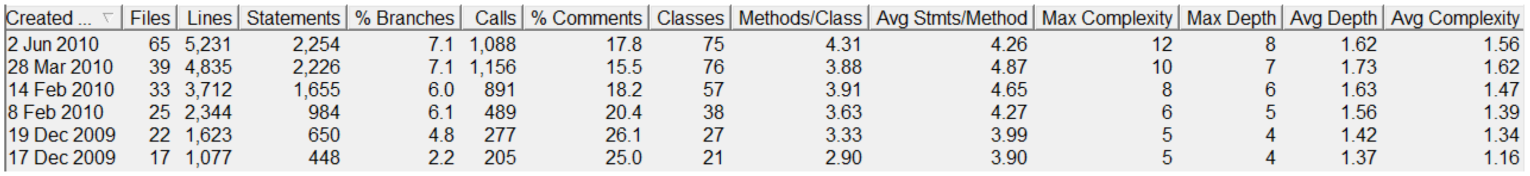
Nearly all the coding took place during winter quarter. After we defined the initial requirements, design, and schedule, we began spirally designing and implementing each requirement. This spiral phase occurred during the winter quarter weeks 5 - 8. Normally we would select a use case from the user guide or story board to implement for the next week. Next, we would create a small informal design and then implement the feature. The next week during our senior project meeting, we would try the feature and write down any improvements we might make. We would then deliver these improvements the following week along with our next new feature.

7.6 METRICS

Metrics are often a good measure of process and software quality. Since most of the work was done using a spiral process with incremental delivery. It was important to understand how our code base is evolving over time.

7.6.1 OVERALL METRICS

The chart below shows several software metrics at key intervals in the development cycle. We see our code base starting small with about 1000 lines and then growing to about 5000 lines. With the first 1000, lines coming from an initial prototyping effort before enrollment in the senior project class. This initial code revolved around trying different techniques to extract execution branches.



Created ...	Files	Lines	Statements	% Branches	Calls	% Comments	Classes	Methods/Class	Avg Stmts/Method	Max Complexity	Max Depth	Avg Depth	Avg Complexity
2 Jun 2010	65	5,231	2,254	7.1	1,088	17.8	75	4.31	4.26	12	8	1.62	1.56
28 Mar 2010	39	4,835	2,226	7.1	1,156	15.5	76	3.88	4.87	10	7	1.73	1.62
14 Feb 2010	33	3,712	1,655	6.0	891	18.2	57	3.91	4.65	8	6	1.63	1.47
8 Feb 2010	25	2,344	984	6.1	489	20.4	38	3.63	4.27	6	5	1.56	1.39
19 Dec 2009	22	1,623	650	4.8	277	26.1	27	3.33	3.99	5	4	1.42	1.34
17 Dec 2009	17	1,077	448	2.2	205	25.0	21	2.90	3.90	5	4	1.37	1.16

Figure 2: Project Metrics for Java Source Code

Given these metrics, we wanted to ensure that the average complexity and average depth remained low. Maintaining a low average depth and low average complexity creates a modular system. This was important for us because we did not always know what our next requirements would look like.

7.6.2 NUMBER OF JAVA STATEMENTS

This figure shows the code growth curve. This curve helps determine when we have completed our requirements. As you can see, we added most of our code from January to March. Yet from March to June, the slope quickly falls off. This means we were successful in implementing all of our initial requirements winter quarter and did not have much new development work spring quarter.

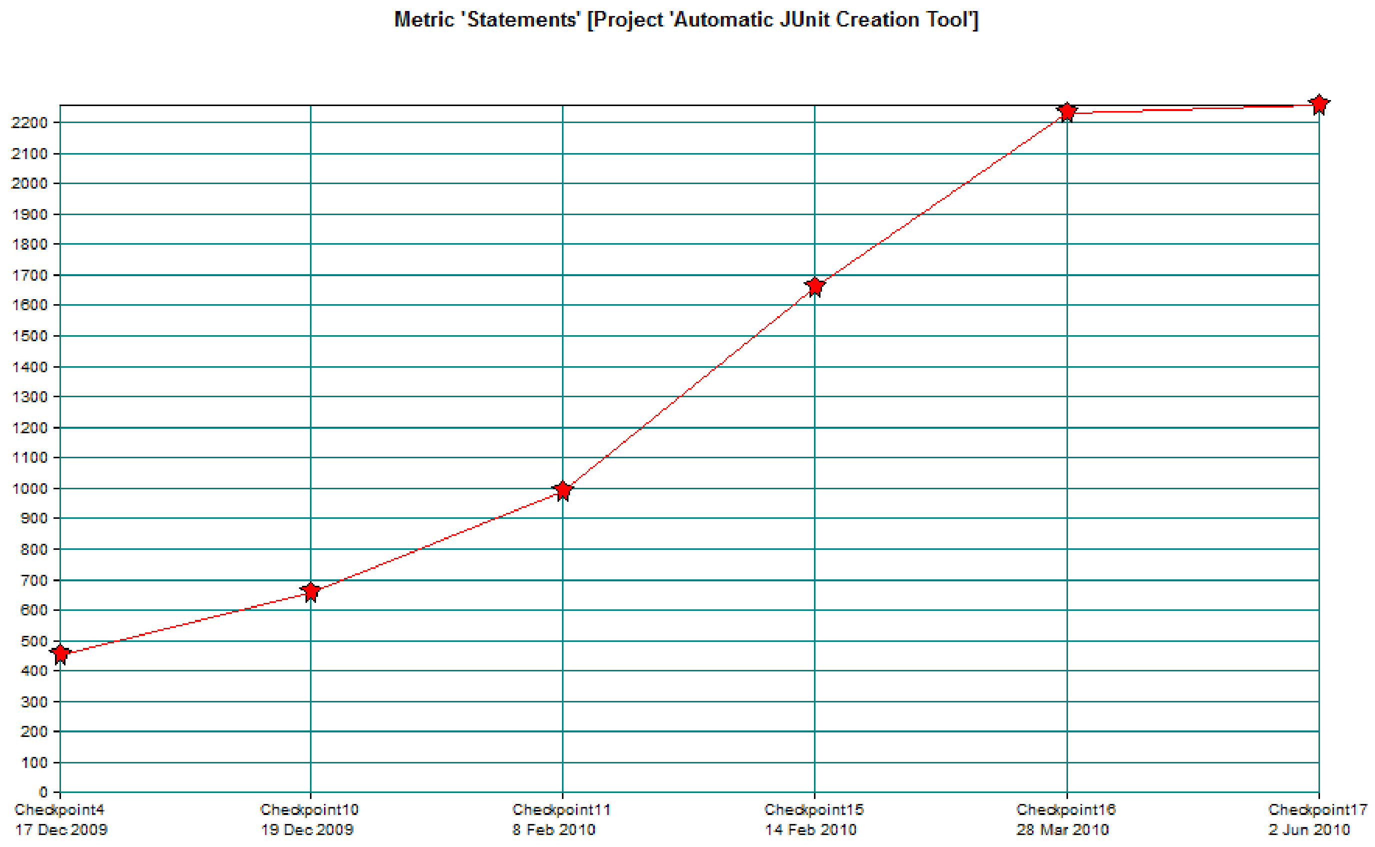


Figure 3: Number of Java Statements vs. Time

8 TOOL OVERVIEW

The following section describes The Automatic JUnit Creation Tool.

8.1 ANTLR TOOLKIT TO INTERPRET JAVA SOURCE CODE

The major technical feature of the Automatic JUnit Creation Tool is its ability to detect execution branches. This was a hard technical problem. Initially we threw around several ideas including manual parsing, counting if statements and looking for block depth. The solution chosen was the ANTLR Toolkit. This toolkit allows you to specify a language grammar such as Java, C++, or any other grammar and then attach your own interpretation code. At the points in the grammar where we find a control structure such as an `if`, `else` or `while` we simply record at what depth we found it.

Imagine we have the following Java code:

```
public class TestClass
{

    public int getSpeedingTicketAmount(int speed)
    {
        if (speed > 50)
        {
            if (speed > 80)
            {
                return 500;
            }
            return 100;
        }
        return 0;

    }
}
```

Figure 4: Java Class to Determine Speeding Ticket Amounts

The branch-parsing module will interpret this code as follows:

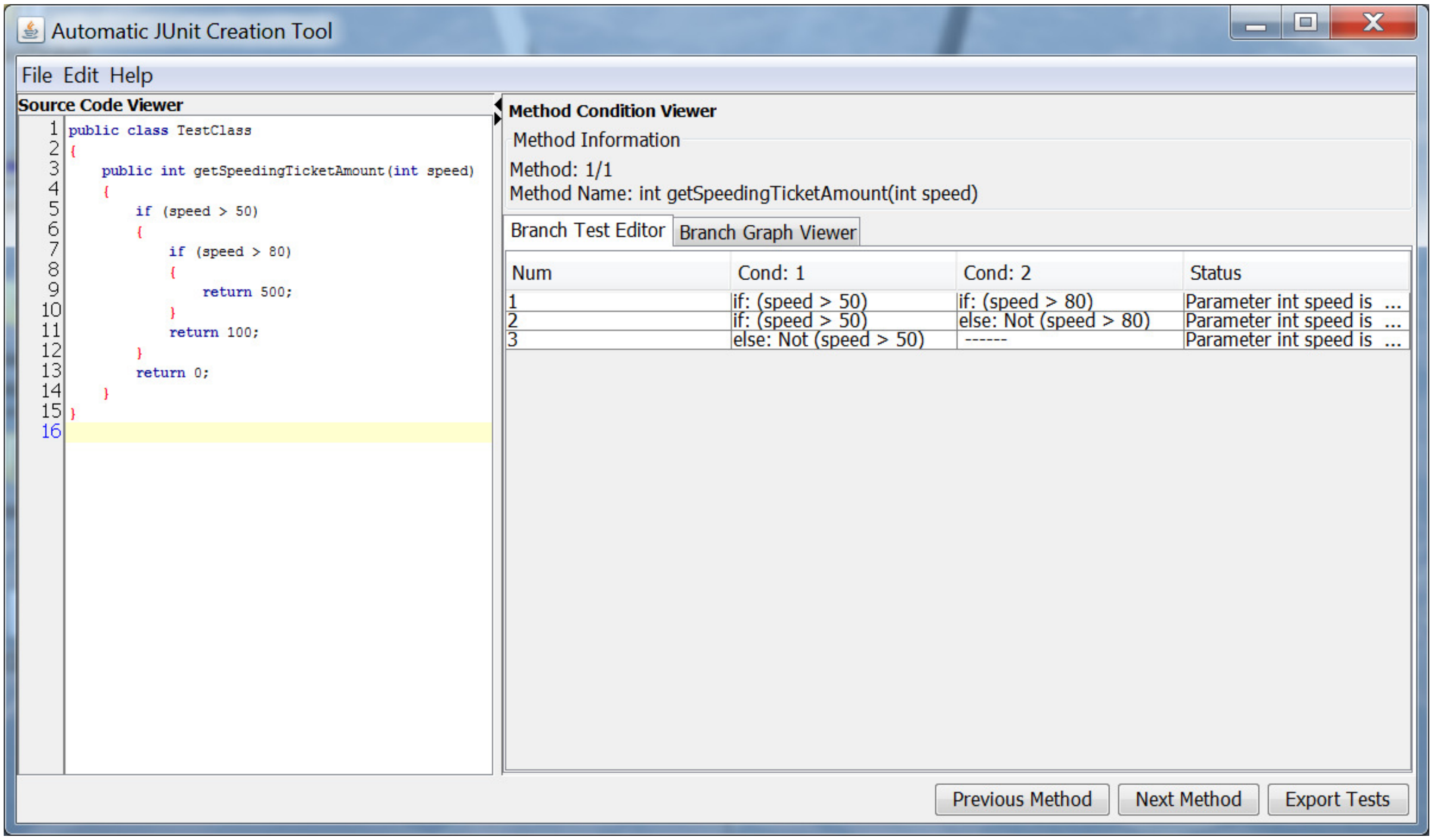
```
If: (speed > 50) Level: 0
If: (speed > 80) Level: 1
Else: Not (speed > 80) Level: 1
Else: Not (speed > 50) Level: 0
```

Figure 5: ANTLR's Evaluation of Figure 4 Source Code Using our Plug-in

Since the ANTLR tool kit moves through the code in a depth first pattern, we are able to reconstruct each branch even nested ones. Ultimately, this toolkit saved a lot of time because it was not necessary to write our own parser or interpreter. We were simply able to attach our own interpretation code that the ANTLR toolkit calls when it locates a particular language feature.

8.2 METHOD CONDITION VIEWER

The next major feature of the tool is the method condition viewer. This feature shows developers all the execution branches available for a method. This view also gives developers feedback regarding how many tests are complete.



```
public class TestClass
{
    public int getSpeedingTicketAmount(int speed)
    {
        if (speed > 50)
        {
            if (speed > 80)
            {
                return 500;
            }
            return 100;
        }
        return 0;
    }
}
```

Method: 1/1
Method Name: int getSpeedingTicketAmount(int speed)

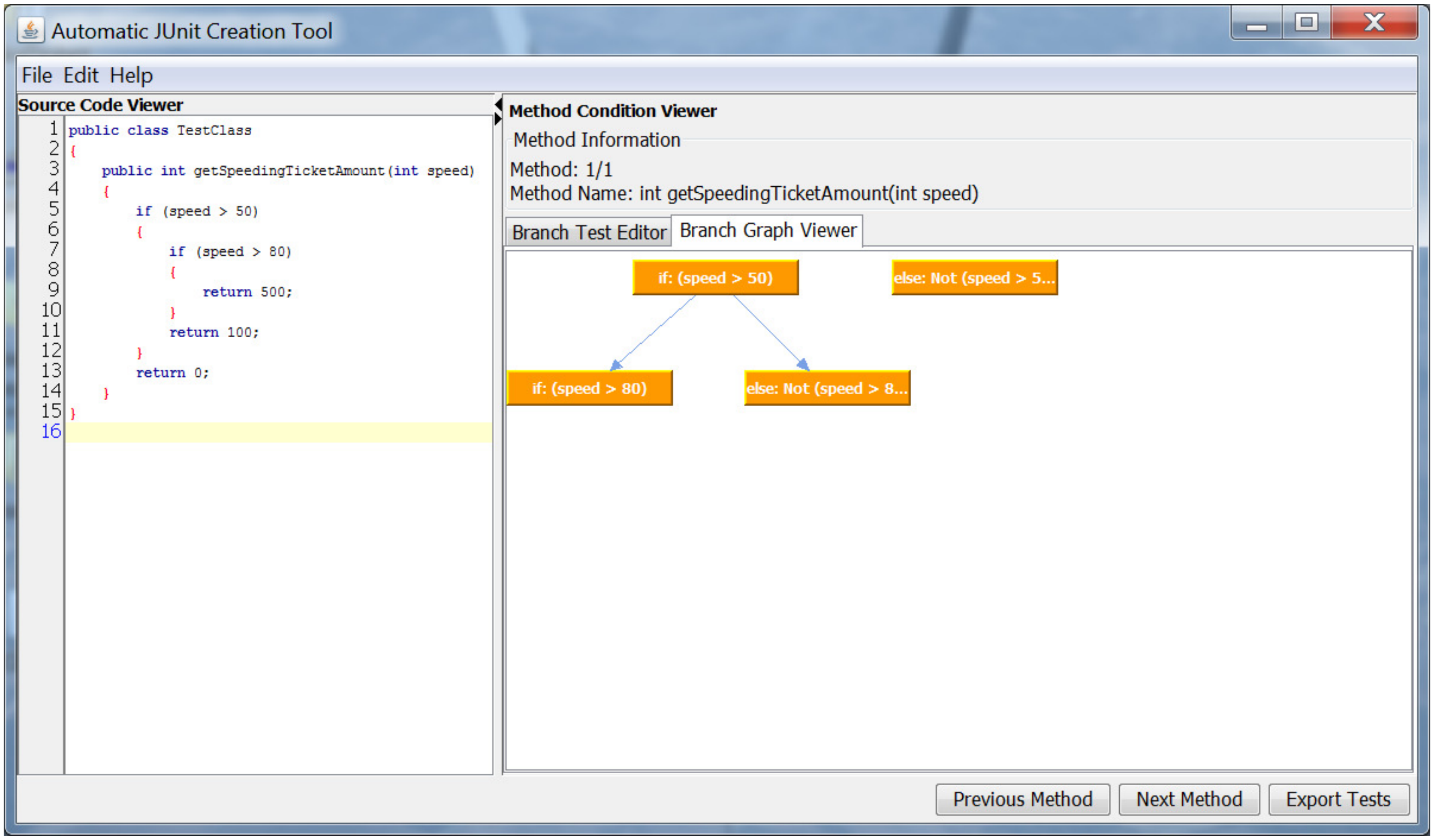
Num	Cond: 1	Cond: 2	Status
1	if: (speed > 50)	if: (speed > 80)	Parameter int speed is ...
2	if: (speed > 50)	else: Not (speed > 80)	Parameter int speed is ...
3	else: Not (speed > 50)	------	Parameter int speed is ...

Figure 6: Method Condition Viewer with Branch Test Editor Tab Selected

The method condition viewer meets the first requirement of an ideal tool. The ability to find method test branches automatically. In the image above, we have opened a very simple class that contains one method. This method calculates the fee amount for a speeding ticket based on driver speed. The branch test editor to the right of the source code, lists all the execution branches in the code.

8.2.1 BRANCH GRAPH VIEWER

Supplementary to branch viewing in the table format the tool also provides a graphical representation. We included this feature spring quarter because of user feedback requesting it. It is simply another view for the branch test editor. Yet this view shows in more detail how branches relate to each other.



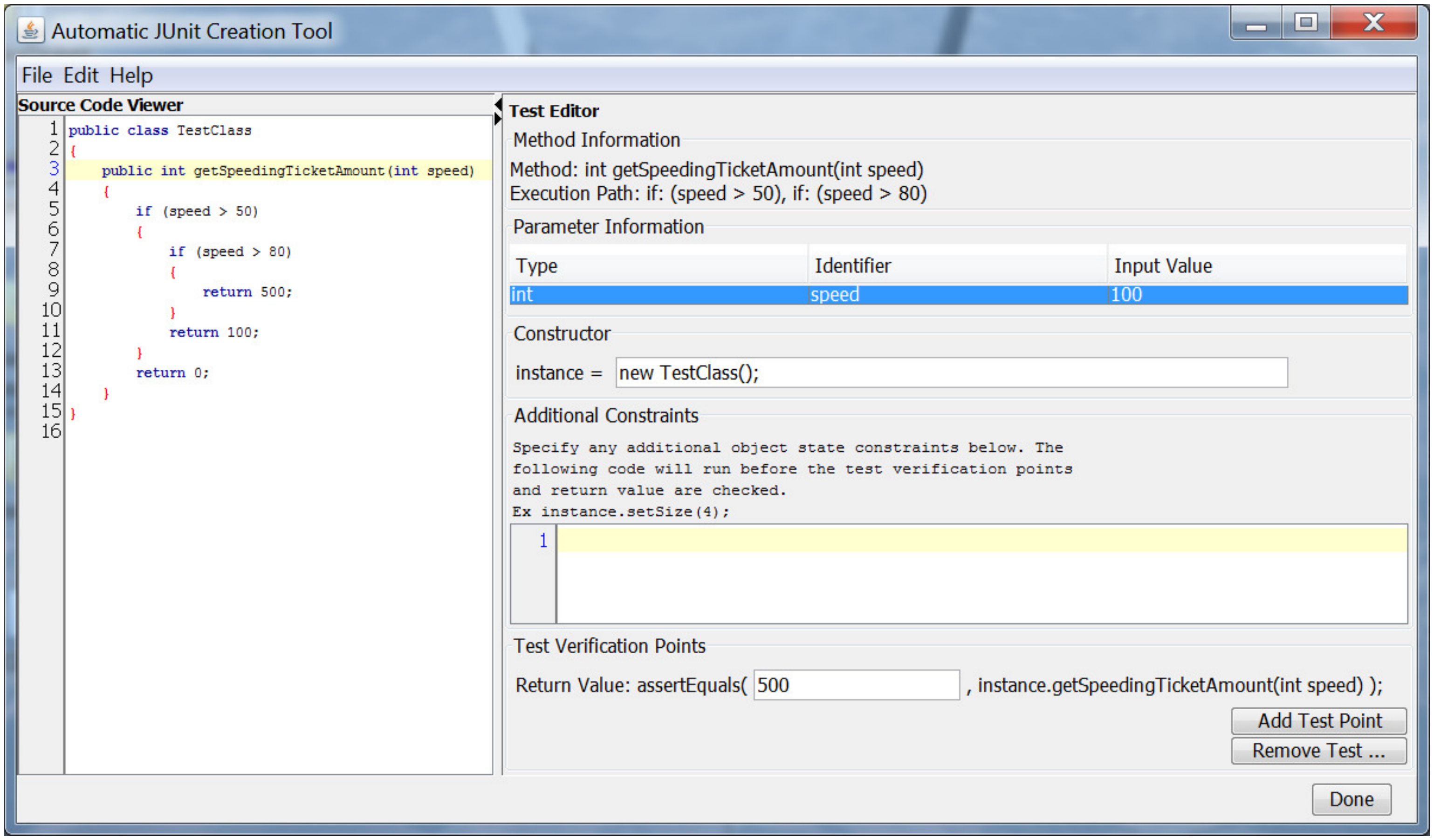
```
public class TestClass
{
    public int getSpeedingTicketAmount(int speed)
    {
        if (speed > 50)
        {
            if (speed > 80)
            {
                return 500;
            }
            return 100;
        }
        return 0;
    }
}
```

Method: 1/1
Method Name: int getSpeedingTicketAmount(int speed)

Figure 7: Method Condition Viewer with Branch Graph Viewer Tab Selected

8.3 TEST CREATION

Perhaps the most important feature of the Automatic JUnit Creation Tool is the test creator. This feature allows a developer to create a JUnit test for a particular execution branch. One of the key design considerations involved with this feature is that developers are able to create tests without thinking about the underlying JUnit test framework. They are simply able to focus on the code they are testing.



```
public class TestClass
{
    public int getSpeedingTicketAmount(int speed)
    {
        if (speed > 50)
        {
            if (speed > 80)
            {
                return 500;
            }
            return 100;
        }
        return 0;
    }
}
```

Method Information

Method: int getSpeedingTicketAmount(int speed)
Execution Path: if: (speed > 50), if: (speed > 80)

Parameter Information

Type	Identifier	Input Value
int	speed	100

Constructor

instance = new TestClass();

Additional Constraints

```
Specify any additional object state constraints below. The
following code will run before the test verification points
and return value are checked.
Ex instance.setSize(4);
```

Test Verification Points

Return Value: assertEquals(500 , instance.getSpeedingTicketAmount(int speed));

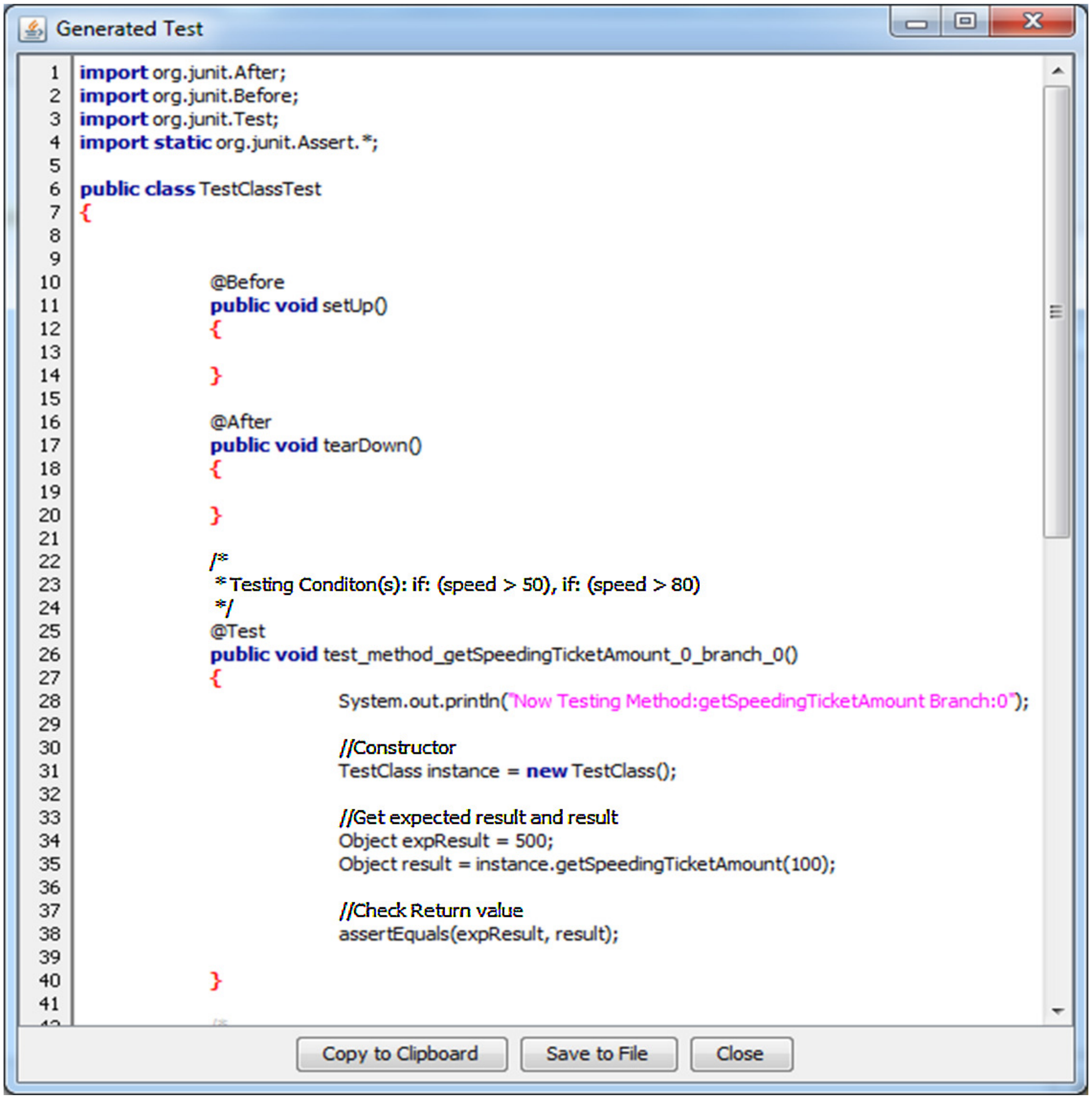
Figure 8: Test Editor Populated with Input Value of 100 and Expected Result of 500

As you can see from the image above all a developer has to do is provide a test input value and the expected result. Above we have provided an input value of 100 and an expected output of 500. We chose the input value 100 because it triggers the branch `if: (speed > 50), if (speed > 80)` which should return 500.

The test editor feature meets the ideal test tool requirement "provide a GUI wizard to input test values". In the above window, all a developer has to think about is the input test value they want to use and its expected result. This shields developers from nearly any thought about what the underlying JUnit framework is doing.

8.4 JUNIT TEST EXPORT

The final feature of the Automatic JUnit Creation Tool is the test export feature. This allows developers to export well-documented test cases that work in the JUnit testing framework.



```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class TestClassTest
{


		@Before
		public void setUp()
		{

		}

		@After
		public void tearDown()
		{

		}

		/*
		 *Testing Conditon(s): if: (speed > 50), if: (speed > 80)
		 */
		@Test
		public void test_method_getSpeedingTicketAmount_0_branch_0()
		{
				System.out.println("Now Testing Method:getSpeedingTicketAmount Branch:0");

				//Constructor
				TestClass instance = new TestClass();

				//Get expected result and result
				Object expResult = 500;
				Object result = instance.getSpeedingTicketAmount(100);

				//Check Return value
				assertEquals(expResult, result);

		}
```

Figure 9: Test Export Viewer

This feature provides two key benefits. First, each branch has its own JUnit test method. This makes isolating failures easy since a single branch error will cause a single failure. Second, each test case contains comments explaining what branch you are currently testing.

This feature meets the "generate tests automatically" requirement of the ideal test tool. The JUnit test export feature allows developers with just a few input values and an expected output to generate five lines of JUnit framework code. Coding is a labor intensive and error prone process thus this feature also reduces the amount of time required to create JUnit tests.

9 SOLICITATION OF USER FEEDBACK

We determined very early in the creation of this tool that many developers had never used anything like it. This meant it must be extremely easy and intuitive to use. Much time went into the user interface during the storyboard and user manual stages. To ensure wide accessibility real user feedback was collected.

9.1 METHODOLOGY

Since the open source community would eventually inherit the tool user feedback exercises were performed using the open source project page. More specifically most exercises involved users simply downloading the tool and trying it out. The users that participated in the data collection exercises were all students in the Cal Poly Computer Science Department.

We asked each student the following:

- Could you start the tool?
- Did it find branches for your Java source file?
- Were you able to create a JUnit test?
- What parts of the tool do you not like?
- What part of the tool do you want to change?

Many of these questions were open ended however, these questions tried to capture the unstructured use that might occur if an individual developer found the open source tool downloaded it and then tried it on their own code.

9.2 USER FEEDBACK

After several user feedback exercises we started to notice a definite pattern. Most users wanted the same thing. These repeated requests included:

- Syntax highlighting for the source code
- The ability to prevent blank JUnit tests from being exported
- Highlighting for the currently selected method in the code viewer
- The ability to save over an existing file

9.3 OVERALL RESULT

Overall, the user feedback activity helped verify that the tool was intuitive and easy to use. Almost every developer thought the tool was useful and many said they could see themselves using the tool to create tests in the future.

10 OPEN SOURCE RELEASE

At the end of winter quarter 2010, the Automatic JUnit Creation Tool was ready for release. The tool had been through some user feedback and worked well enough it could be put out as a beta release.

10.1 CREATION OF SOURCEFORGE PROJECT

Initially we chose Source Forge to host our project with almost no thought. Mainly because Source Forge already hosts over 230,000 projects [4]. It is common for Source Forge to distribute over 3 million open source downloads a day [4]. Source Forge also has a trusted name and large user base.

The first step was the creation of a Source Forge project page. During this step, I had to create a Source Forge user name and then create a project. In order, to create the project I had to write short description of the Automatic JUnit Creation Tool. I also had to provide some information about the computing platform it runs on.

The next step was actually configuring the project page. At this point, I turned on the subversion hosting module so that I could move all my code to the Source Forge version control system. I then also turned on the file hosting and made a release. Almost immediately after making the first release the project started to experience downloads.

The Automatic JUnit Creation Tool is located at:
`https://sourceforge.net/projects/amaticjunittool/`

10.2 SOURCE FORGE LIMITATIONS & INVESTIGATION OF ALTERNATE HOSTING

While Source Forge did immediately seem like the one stop shop to host your open source project within a few weeks its limitations and drawbacks were discovered. The developer user interface for Source Forge is extremely slow and confusing to use. For instance, it is complicated to view source code diffs between commits, manage and track defects and access the project wiki.

After struggling with the limitations of Source Forge an investigation into other open source hosing platforms was performed. In particular, GitHub was examined. What is great about GitHub is that you get a true feel of community because it is all centered around users. In addition, each developer is free to branch anyone's code at any time. The tools for managing defects, managing project status and performing code diffs were also much better. Yet GitHub still was not perfect because it did not do distribution well. That was the one area where Source Forge really led the way.

Now unsatisfied with both Source Forge and GitHub I looked at what I could do to make Source Forge better. It was determined that Source Forge allowed for the Trac project management framework. This essentially solved all the initial complaints about Source Forge with its built in wiki, ticket system and web based repository browser. Once Trac was installed, everything else on Source Forge was disabled leaving users only the option of going to Trac.

Overall, this research into alternate open source hosing platforms played a big role in the way Source Forge is used for the Automatic JUnit Creation Tool. The project page for the tool leverages Source Forge's best asset the distribution framework while avoiding the built in project management tools through the use of Trac.

10.3 COMMUNITY INVOLVEMENT WITH TOOL

After 11 weeks, the open source community has downloaded just fewer than 300 copies of the Automatic JUnit Creation Tool. No feedback or inquires have been made.

11 OPEN SOURCE EXPERIENCE AND OBSERVATIONS

The open source launch of the Automatic JUnit Creation Tool led to many observations about open source software in general.

11.1 UNCERTAIN QUALITY

One observation made while working in the open source world is that many releases are of undetermined quality. Much of the formal release process that exists for corporations selling boxed software simply does not exist in open source. Taking the Automatic JUnit Creation Tool as an example, we simply made new releases once we had enough new features at an acceptable quality.

We could have had a formal release process but our team is small. Many of the other teams on Source Forge are also small. This likely means much of the software released to the open source community is of uncertain quality.

11.2 INACTIVE PROJECTS

Another observation we made is that projects become inactive. The Automatic JUnit Creation Tool for instance needed syntax highlighting. We searched Source Forge to find an acceptable open source library. Once found we examined the project page and noticed that no activity had occurred in over 200 days. While the code did work, we discovered that the original programmers left many features unimplemented. The take away point being that most developers donate their time to open source projects so do not expect their projects to stay active forever. In the case of the syntax highlighting project the core developers may have implemented Java syntax highlighting and then stopped because it met their needs.

11.3 THIRD-PARTY DISTRIBUTION

Another observation made after our open source release is how fast other sites will begin to redistribute your software. Nothing within the GPL V3 license says you cannot redistribute the tool, so many sites did. These sites ranged from Softpedia to less reputable sites hosting open source downloads to get advertisement revenue. This rapid redistribution has several implications for developers. First, you may not always be the single source of distribution. Say you find a major bug and create a new build. Simply replacing the link on your download page does not guarantee all third-party distribution sites will have the new build. It may take several months to propagate to all the third-party sites. Yet this third-party network does help developer as well. This external network acts as free advertizing spreading the word about your tool much faster than what might have been otherwise possible.

12 FUTURE WORK

We designed and implemented the entire Automatic JUnit Creation Tool during winter and spring quarter. Yet because of this effort, we did identify some areas that future developers may want to research.

12.1 DEPENDENCY INJECTION

The next major technical challenge the Automatic JUnit Creation Tool will need to solve is dependency injection. The tool currently works great for classes that use mainly primitive language types. However, if a class depends on other classes with complex behavior things are not as smooth. We need to verify the effects the class under test has on its dependencies. Future versions of the tool would integrate an existing tool kit like Easy Mock into the graphical user interface. This would enable developers to easily break all dependencies and test any class as if it was comprised of all primitive types.

12.2 NETBEANS PLUG-IN SUPPORT

Another major improvement would be the transition to a plug-in for the Netbeans or Eclipse development environment. This would remove one layer of complexity by allowing developers to stay within their current workflow to create tests. They would also be able to run tests from inside the tool because the Netbeans or Eclipse project would have access to all the project configuration settings needed to compile a complex project.

13 CONCLUSION

Overall, this senior project was a success. In the very early stages, we were able to identify several key problems with the JUnit testing framework. We then defined an ideal solution and verified that no other tools existed that fulfilled our requirements. After the search for existing technology turned up empty handed we then proceeded to use a spiral development model with incremental delivery to build the entire application. By the end of winter quarter, we had a product stable enough for release to the open source community so we created a Source Forge project page. During spring quarter, we conducted many user feedback exercises to determine how usable our new product was. Many of the Cal Poly students that tried our product thought it was very intuitive and said they could see themselves using the tool in the future. We also identified several areas of future work that will make the tool even more useful. Ultimately, this senior project produced a working open source tool that solves many of our identified problems with JUnit testing framework.

14 REFERENCES

1. "Introduction". JTestCase. 6-2-2010 <http://jtestcase.sourceforge.net/introduction.html>.
2. Static code analysis, Louridas, P. *Software, IEEE* On page(s): 58 - 61 , Volume: 23 Issue: 4, July-Aug. 2006
3. Pakkirisamy, L.Babu. "Coarse-Grained Unit Testing". Dr Dobbs. 6-2-2010 <http://www.drdobbs.com/java/222601023 >.
4. "About Source Forge". SourceForge.net. 6-2-2010 <https://sourceforge.net/about>.

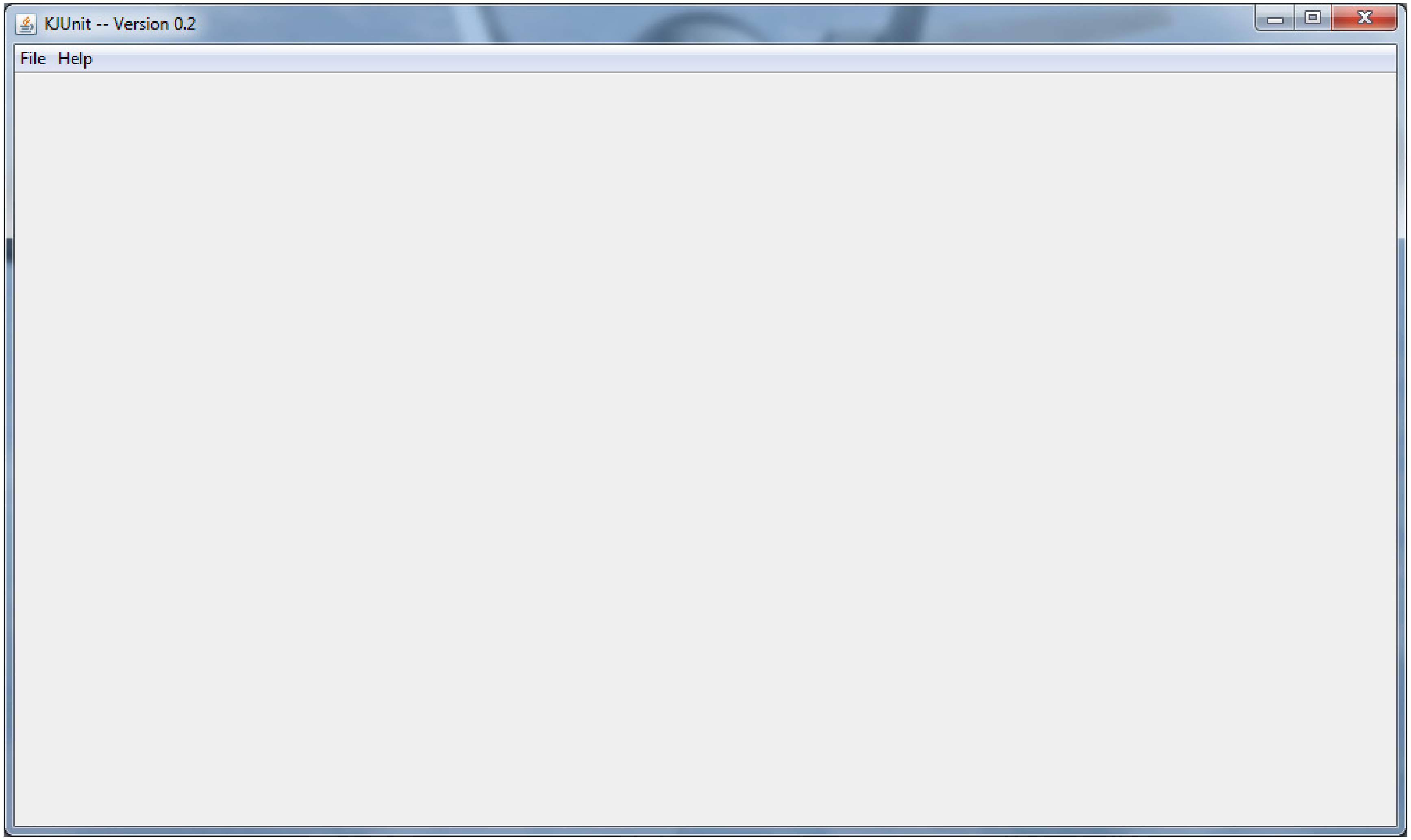
Appendix A

Automatic JUnit Creation Tool

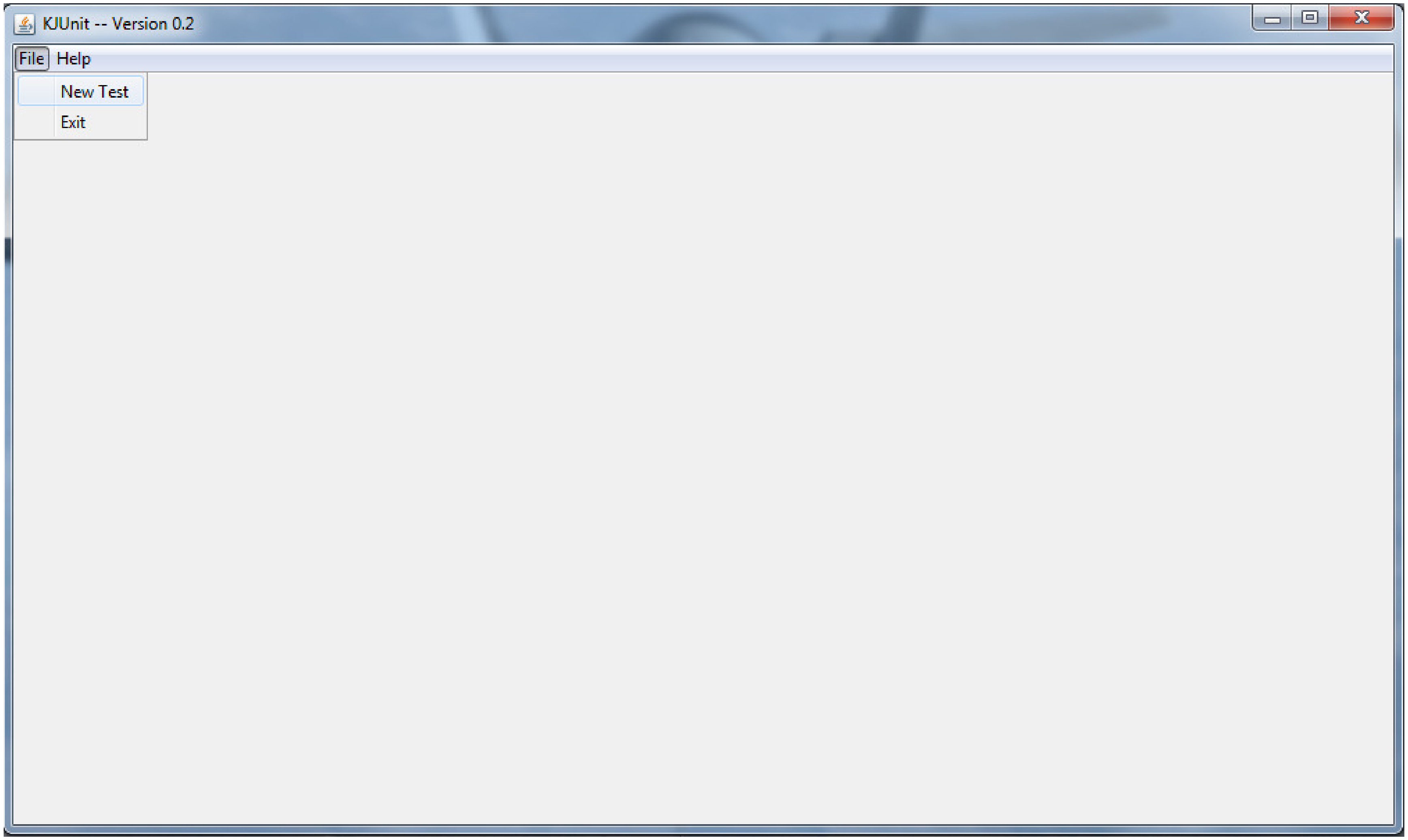
Story Board

William Whitney

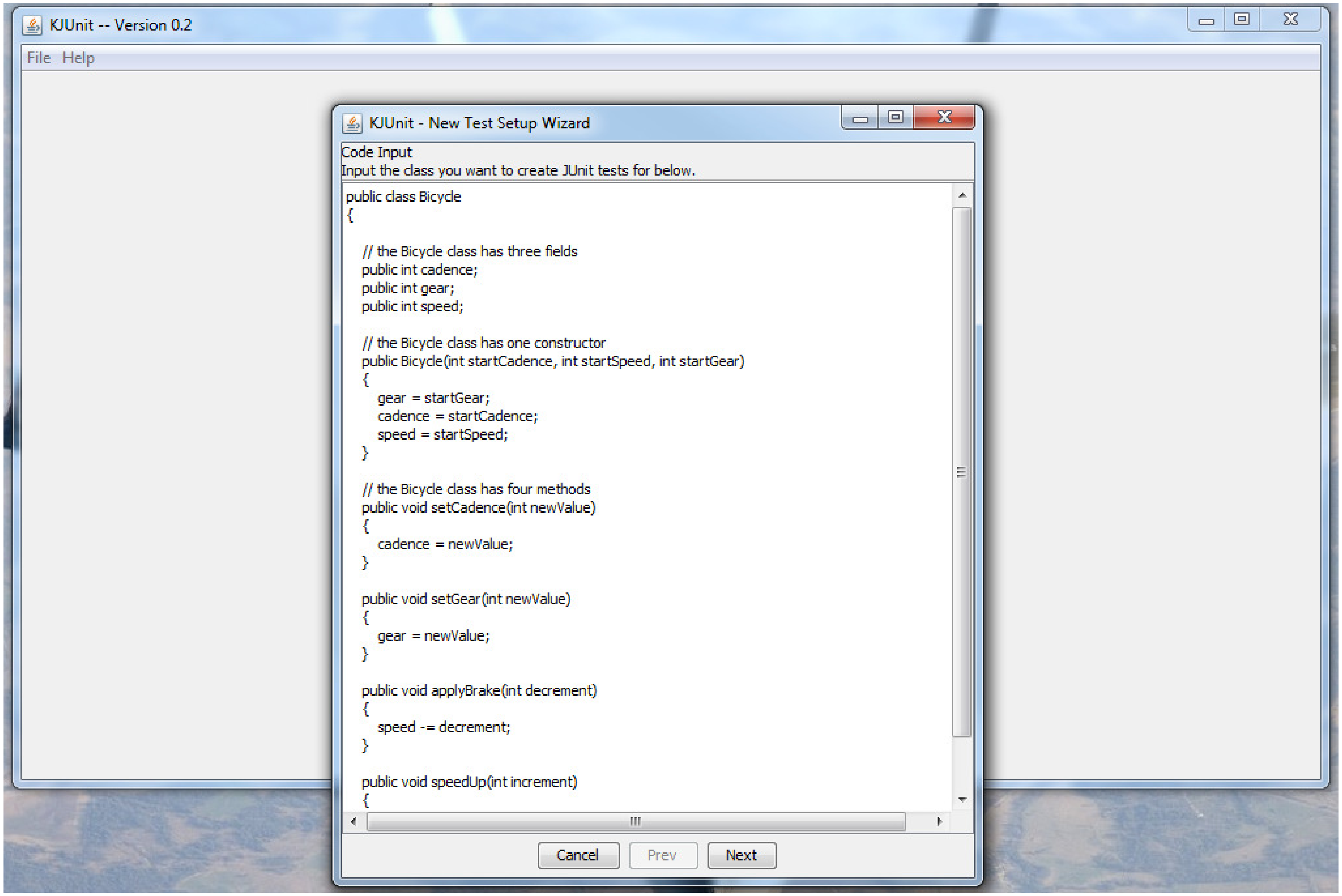
Senior Project



Initial Program Start Up



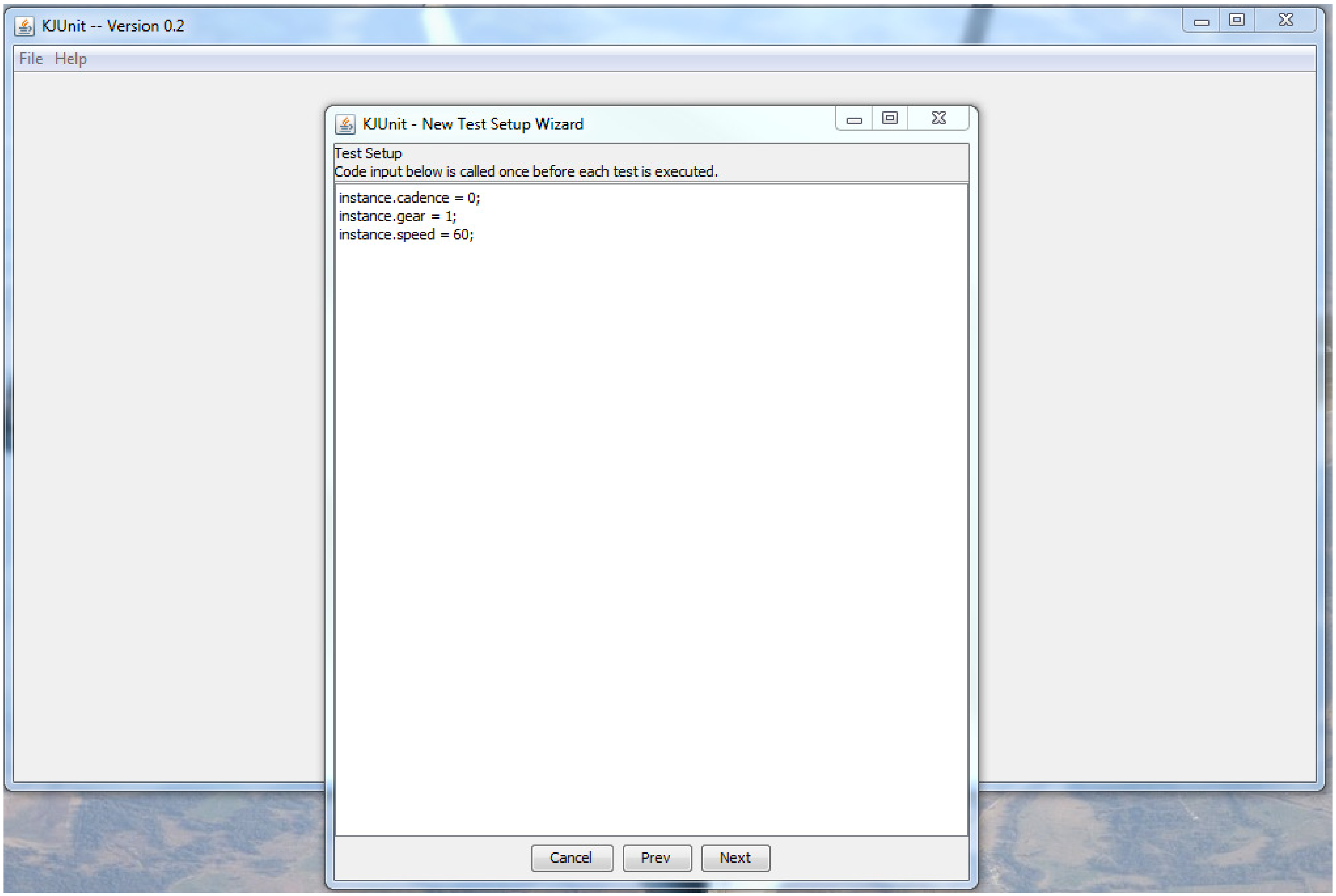
Click File -> New Test



Code Input
Input the class you want to create JUnit tests for below.

```
public class Bicycle
{

   // the Bicycle class has three fields
   public int cadence;
   public int gear;
   public int speed;

   // the Bicycle class has one constructor
   public Bicycle(int startCadence, int startSpeed, int startGear)
   {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }

   // the Bicycle class has four methods
   public void setCadence(int newValue)
   {
      cadence = newValue;
   }

   public void setGear(int newValue)
   {
      gear = newValue;
   }

   public void applyBrake(int decrement)
   {
      speed -= decrement;
   }

   public void speedUp(int increment)
   {
```

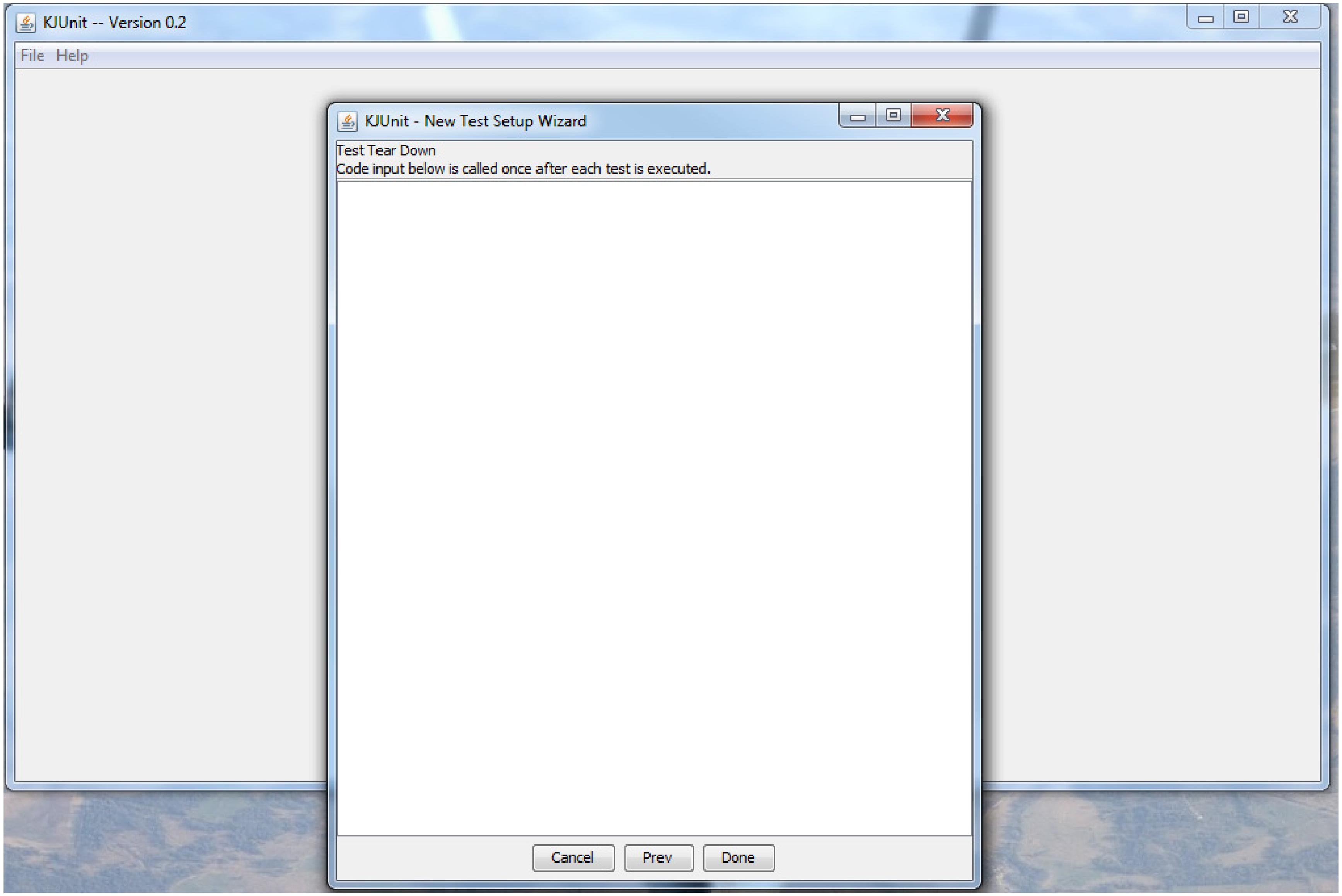
Window Asking For Java Class



Test Setup
Code input below is called once before each test is executed.

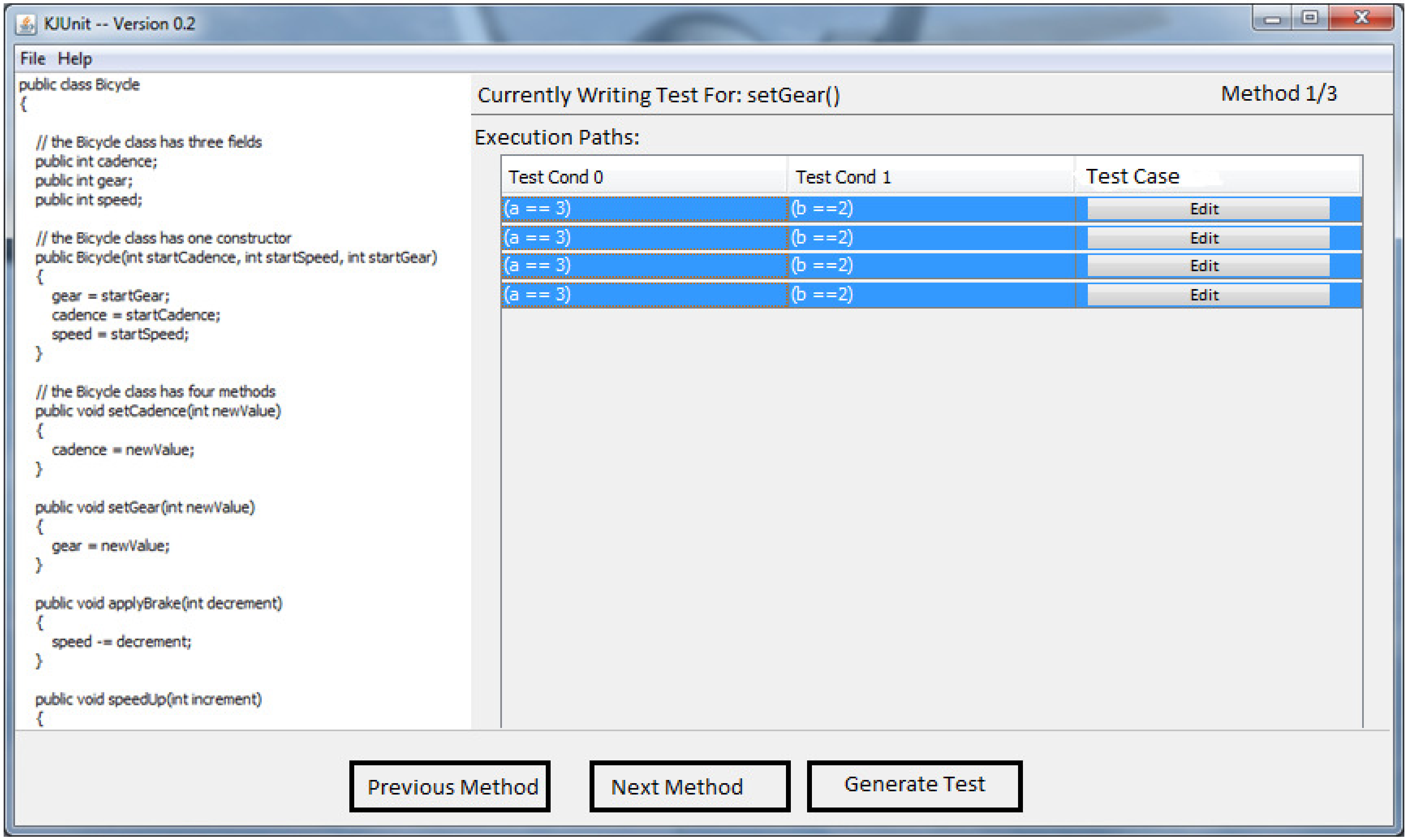
```
instance.cadence = 0;
instance.gear = 1;
instance.speed = 60;
```

Input Test Setup Information



Test Tear Down
Code input below is called once after each test is executed.

Input Test Tear Down Information

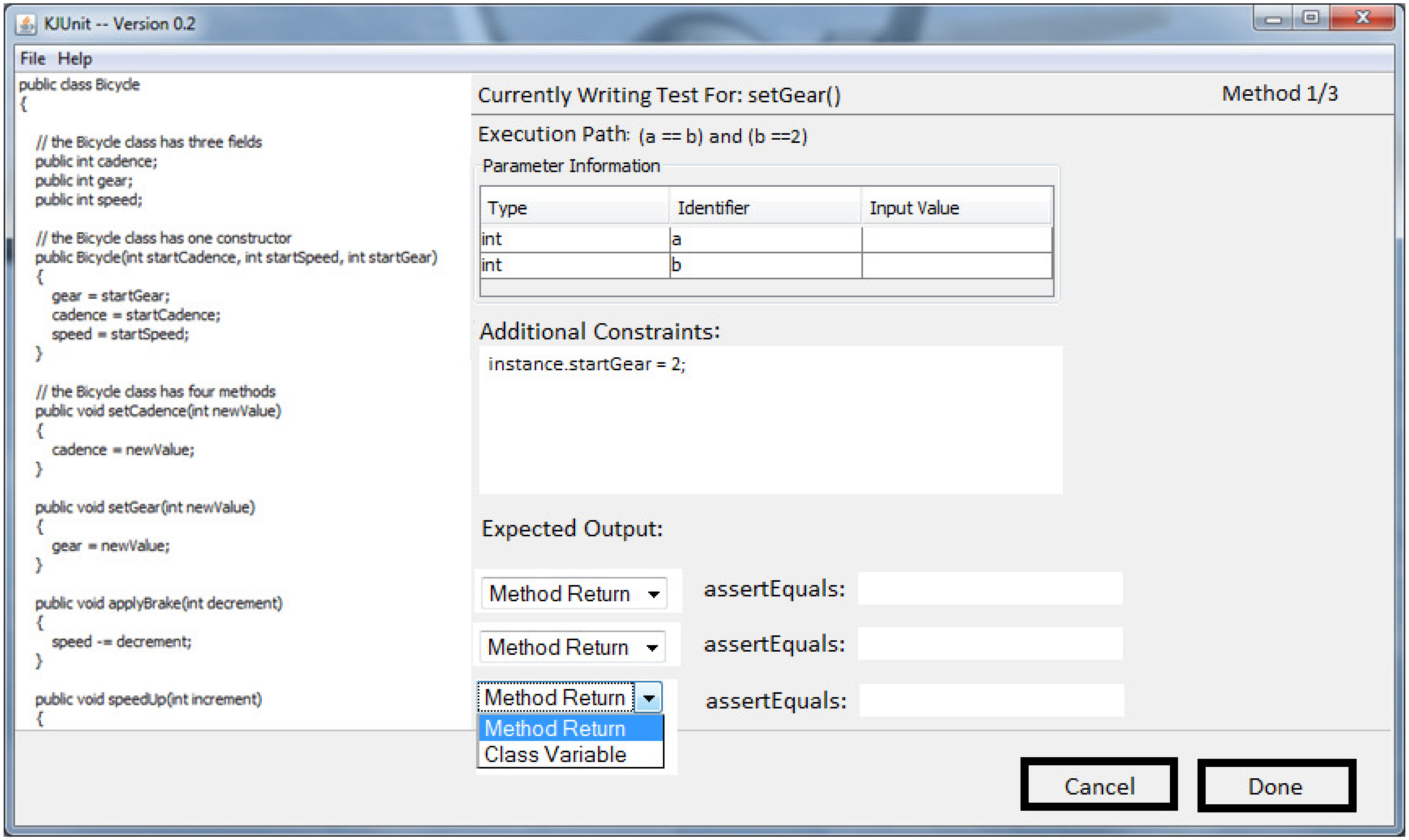


```
public class Bicycle
{

  // the Bicycle class has three fields
  public int cadence;
  public int gear;
  public int speed;

  // the Bicycle class has one constructor
  public Bicycle(int startCadence, int startSpeed, int startGear)
  {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
  }

  // the Bicycle class has four methods
  public void setCadence(int newValue)
  {
    cadence = newValue;
  }

  public void setGear(int newValue)
  {
    gear = newValue;
  }

  public void applyBrake(int decrement)
  {
    speed -= decrement;
  }

  public void speedUp(int increment)
  {
```

Test Cond 0	Test Cond 1	Test Case
(a == 3)	(b ==2)	Edit
(a == 3)	(b ==2)	Edit
(a == 3)	(b ==2)	Edit
(a == 3)	(b ==2)	Edit

Shows the Various Test Branches.

Each Test Branch effectively represents a self-contained test for the method. If the edit button is pressed the test editor will appear.



```
public class Bicycle
{

   // the Bicycle class has three fields
   public int cadence;
   public int gear;
   public int speed;

   // the Bicycle class has one constructor
   public Bicycle(int startCadence, int startSpeed, int startGear)
   {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }

   // the Bicycle class has four methods
   public void setCadence(int newValue)
   {
      cadence = newValue;
   }

   public void setGear(int newValue)
   {
      gear = newValue;
   }

   public void applyBrake(int decrement)
   {
      speed -= decrement;
   }

   public void speedUp(int increment)
   {
```

Currently Writing Test For: setGear()

Execution Path: (a == b) and (b ==2)

Parameter Information

Type	Identifier	Input Value
int	a	
int	b	

Additional Constraints:

```
instance.startGear = 2;
```

Expected Output:

Method Return	assertEquals:	
Method Return	assertEquals:	
Method Return (Method Return / Class Variable)	assertEquals:	

This screen details building a test for a specific execution path.

The parameter information box allows values to be set for methods that take parameters. The additional constraints box allows programmers trigger the current execution path even if it contains something like a class member variable. The Expected output section allows programmers to define the success criteria of the test in terms of class member variables and method return value.

Appendix B

Automatic JUnit Creation Tool

User Manual

CSC 491

Winter 2010

Student: William Whitney
Advisor: Dr. John Dalbey

Contents

1 Creating a New Test

This section describes how to test a new Java class using the Automatic JUnit Creation Tool.

1.1 Importing Existing Source File

The Automatic JUnit Creation Tool requires a Java Class file to get started.

1. **Click** ‘File’ to show the file menu.
2. **Click ‘**New Test’ to start the new test wizard.
3. **Click** ‘Browse” to pull up the source file selection box.
4. **Navigate** to your Java Source File Path.
5. **Click** ‘Open’ to capture the path to your Java Class.
6. **Click** ‘Next’ to move to the test setup panel.

1.2 Defining a Test Setup Procedure

Test may be defined with a block of code that will be executed once before each test. This section describes how to define a test setup method. This feature is useful if you want to setup your test object to a certain initial state before each test.

1. **Complete** ‘Step 1.1 Importing Existing Source File’.
2. **Manipulate** class instance variable ‘testObj’ .

Note: The Automatic JUnit Creation tool will automatically create an object called testObj. This object is the basis of all testing. It will be instantiated using the default constructor for the main class. Yet this behavior can be overridden (See Figure 1).

Method Test Setup:
```
testObj = new Calculator(Calculator.Postfix);
testObj.push(“3”);
testObj.push(“1”);
testObj.push(“+”);
```

Figure 1: Test Setup Example

3. **Click** ‘Next’ to navigate to the test tear down panel

1.3 Defining a Test Tear Down Procedure

Tests may be specified with a set of statements called once after each test completes. This feature is useful if you want to reverse a certain action after each test finishes.

1. **Complete** ‘Step 1.2 Defining a Test Setup Procedure”.
2. **Manipulate** class instance variable ‘testObj’ (See Figure 2).
3. **Click ‘**Finish’ to exit wizard.

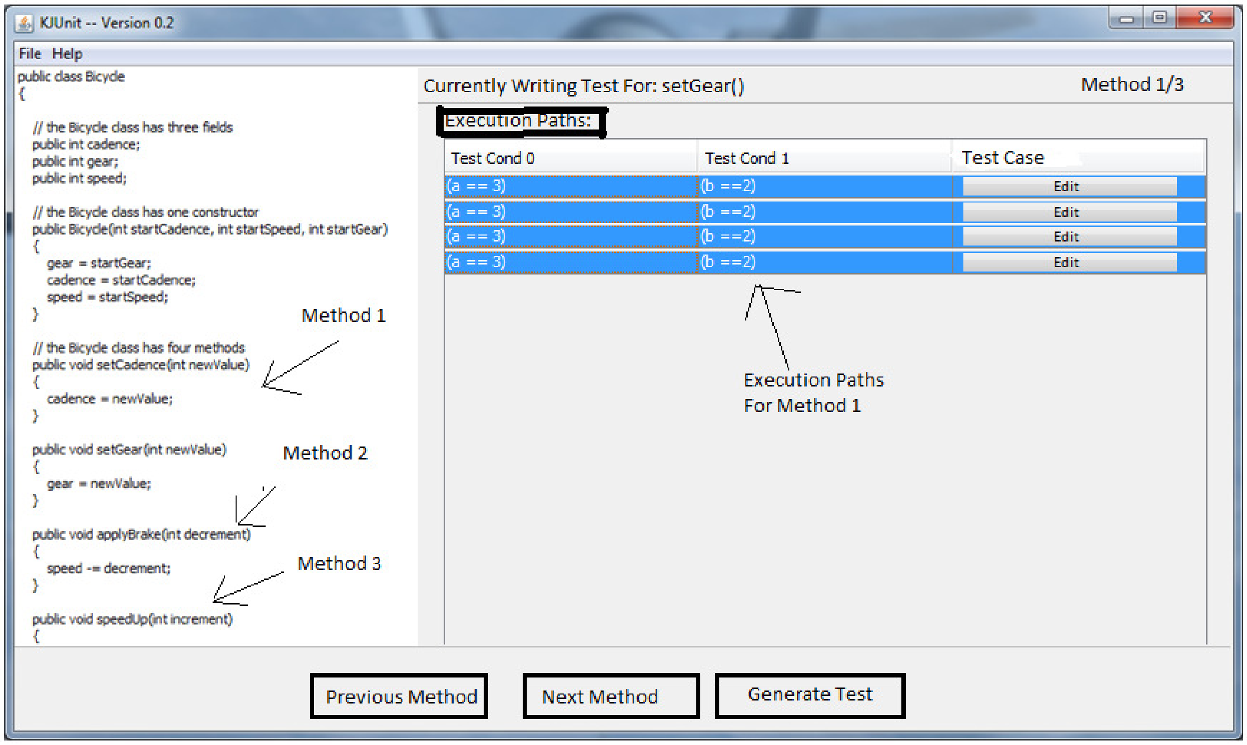
Example:

```
testObj.query(“DELETE * FROM cities”);
testObj.query(“DROP TABLE users”);
```

Figure 2: Test Tear Down Example

2. Viewing Method Execution Paths

After the completion of the new test wizard, the Automatic JUnit Creation Tool will analyze your class. This analysis will find all execution paths though your source code. An execution path is a specific way to enter and terminate a method.



```
public class Bicycle
{

  // the Bicycle class has three fields
  public int cadence;
  public int gear;
  public int speed;

  // the Bicycle class has one constructor
  public Bicycle(int startCadence, int startSpeed, int startGear)
  {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
  }

  // the Bicycle class has four methods
  public void setCadence(int newValue)
  {
    cadence = newValue;
  }

  public void setGear(int newValue)
  {
    gear = newValue;
  }

  public void applyBrake(int decrement)
  {
    speed -= decrement;
  }

  public void speedUp(int increment)
  {
```

Test Cond 0	Test Cond 1	Test Case
(a == 3)	(b ==2)	Edit
(a == 3)	(b ==2)	Edit
(a == 3)	(b ==2)	Edit
(a == 3)	(b ==2)	Edit

Figure 3: Execution Path Editor

2.1 Changing Methods

The execution paths shown within the Automatic JUnit Creation tool are for the currently selected method (See Figure 3).

- **Click** 'Next Method' button to view execution paths for the next method.
- **Click** 'Previous Method' button to view execution paths for the previous method.

2.2 Viewing Execution Path Detail

The execution path tab shows all ways the currently selected method can enter and terminate (See Figure 3). This view provides a panel with a read only version of your code and a table within the execution path tab showing each execution branch for the current method.

- **Look** in the 'code panel' to see a read only version of your code.
- **Look** at the 'method status' bar to view the method name and return type.
- **Look** at the 'method status' bar to see which method number you are currently working on.
- **Look** at each row within the 'Execution Path' table to see an execution branch.
- **Look** at each column within the 'Execution Path' table to view the test branch conditions.

3. Creating Execution Path Tests

Each branch within the Automatic JUnit Creation Tool represents an individual JUnit test. This section describes how to create a test for a specific branch.

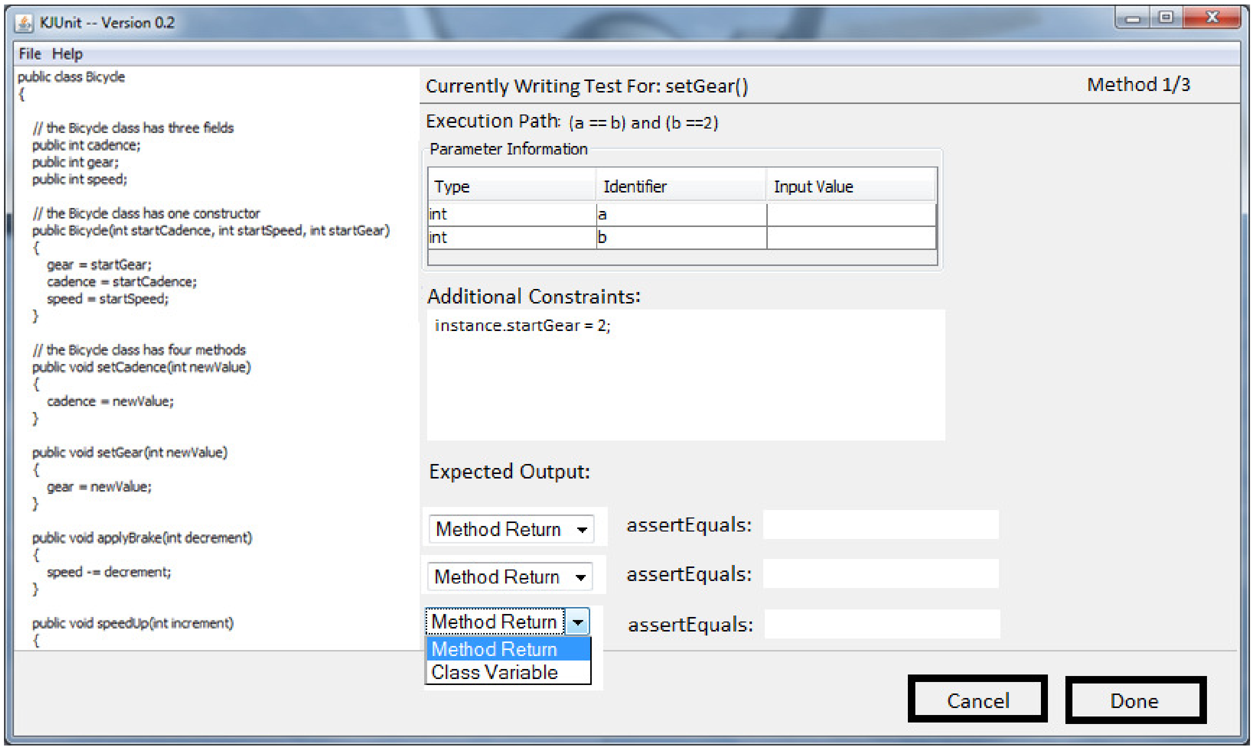
3.1 Editing an Execution Path

Follow the directions bellow to edit an execution branch.

1. **Click** the 'Execution Paths' tab (Figure 3).
2. **Click** the 'edit' button to modify a specific execution path (Figure 3).

3.2 Setting Method Input Values

Each execution path represents a particular path triggered by a specific set of inputs. This means that all inputs defined on the ‘Execution Path Test Creator’ page (Figure 4) must make the given test condition true. Method input parameters will often satisfy this condition.



```
public class Bicycle
{
   // the Bicycle class has three fields
   public int cadence;
   public int gear;
   public int speed;

   // the Bicycle class has one constructor
   public Bicycle(int startCadence, int startSpeed, int startGear)
   {
      gear = startGear;
      cadence = startCadence;
      speed = startSpeed;
   }

   // the Bicycle class has four methods
   public void setCadence(int newValue)
   {
      cadence = newValue;
   }

   public void setGear(int newValue)
   {
      gear = newValue;
   }

   public void applyBrake(int decrement)
   {
      speed -= decrement;
   }

   public void speedUp(int increment)
   {
```

Currently Writing Test For: setGear() Method 1/3

Execution Path: (a == b) and (b ==2)

Parameter Information

Type	Identifier	Input Value
int	a	
int	b	

Additional Constraints:

instance.startGear = 2;

Expected Output:

Method Return assertEquals:

Method Return assertEquals:

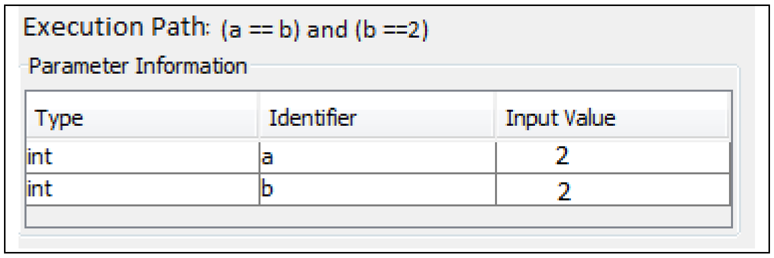
Method Return assertEquals:

Method Return
Class Variable

Figure 4: Execution Path Test Creator

1. **Complete** ‘Step 3.1 Editing an Execution Path’.
2. **Observe** the logic contained in the current execution path.
3. **Locate** any method parameter inputs.
4. **Input** a method parameter value for each variable that helps satisfy the execution path.

Example:



Execution Path: (a == b) and (b ==2)

Parameter Information

Type	Identifier	Input Value
int	a	2
int	b	2

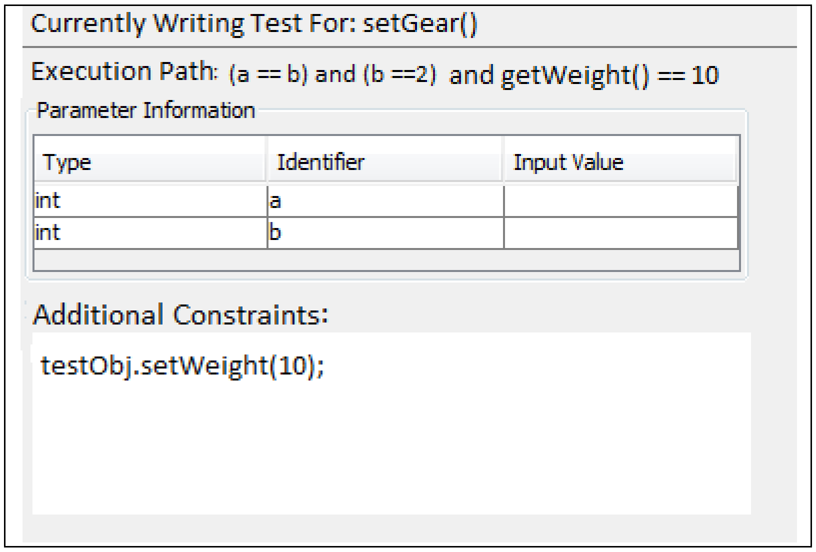
Figure 5: Method Input Parameter Table

3.3 Setting Additional Test Inputs

In many cases, not all test branches can be triggered by method inputs alone. Class instance variables or object state may be involved. The Additional Test Inputs feature allows users to specify inputs that rely on object state.

1. **Complete** 'Step 3.2 Setting Method Input Values'.
2. **Observe** the logic contained in the current execution path.
3. **Locate** any parameters dependent on object state.
4. **Input** code manipulating testObj that will trigger the correct object state

Example:


Currently Writing Test For: setGear()

Execution Path: (a == b) and (b ==2) and getWeight() == 10

Parameter Information

Type	Identifier	Input Value
int	a	
int	b	

Additional Constraints:

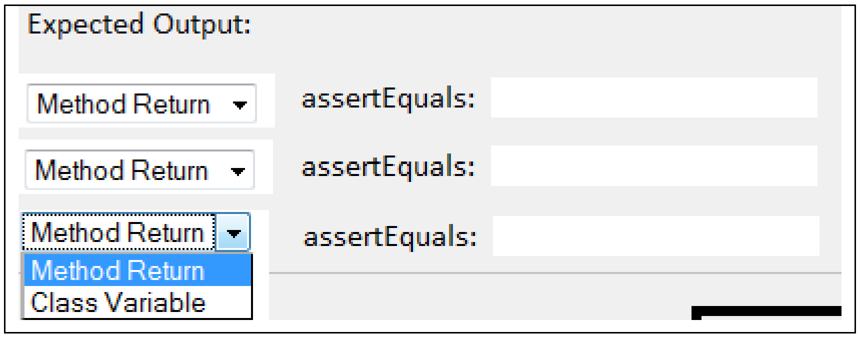
```
testObj.setWeight(10);
```

Figure 6: Showing Test Input Dealing With Class State

3.4 Capturing Expected Behavior

After specifying the correct input to trigger the current execution path the expected output must be captured.

1. **Click** 'drop down box' under 'Expected Output' to specify the type of test point you want.
2. **Input** 'expected value' in the assertEquals box to define test point


Expected Output:

Method Return assertEquals:

Method Return assertEquals:

Method Return assertEquals:

Figure 7: Expected output window

4. Generating JUnit Tests

After all test branches for all methods have been specified, you are ready to generate a JUnit test file.

4.1 Generating a JUnit Test

1. **Click** ‘Generate Test’ within the Execution Path Editor view (Figure 3).
2. **Choose** a location on your computer to save the file.
3. **Click** ‘Save’

```
@Test
public void testSmartSub()
{
      /*
       * Testing Conditon: (num1 > num2)
       */
      assertEquals(testObj.smartSub(20, 10), 10);

      /*
       * Testing Conditon: Not(num1 > num2), (num2 < num1)
       */
      assertEquals(testObj.smartSub(10, 20), 10);

      /*
       * Testing Conditon: Not(num1 > num2), Not(num2 < num1)
       */
      assertEquals(testObj.smartSub(5, 5), 0);

}
```

Figure 8: Sample JUnit Test Ouputs

Appendix C

Senior Project

Automatic JUnit Creation Tool

Architecture Overview

CSC 491

Winter 2010

Student: William Whitney
Advisor: Dr. John Dalbey

Table of Contents

1. Architecture

This section describes the technology stack for the Automatic JUnit Creation tool and some of the key design elements.

1.1 Technologies

The Automatic JUnit Creation tool will utilize the following technologies:

- Java run time environment 1.6
- Java swing for user interface interactions
- ANTLR - ANother Tool for Language Recognition

1.1.1 ANTLR

The ANTLR language recognition tool allows the Automatic JUnit Creation Tool to interpret Java 1.5 syntax differently. The capability allows hooks to be set that will undertake a particular action when key language structures occur. These ANTLR hooks free the Automatic JUnit Creation tool from the worries involved in syntax validation and symbol tree traversal.

To find more information on ANTLR see: http://www.antlr.org/

1.2 Design

The Automatic JUnit Creation architecture is intended to remain as modular as possible. This will enable future open source feature enhancements and upgrades with a minimal overhead cost.

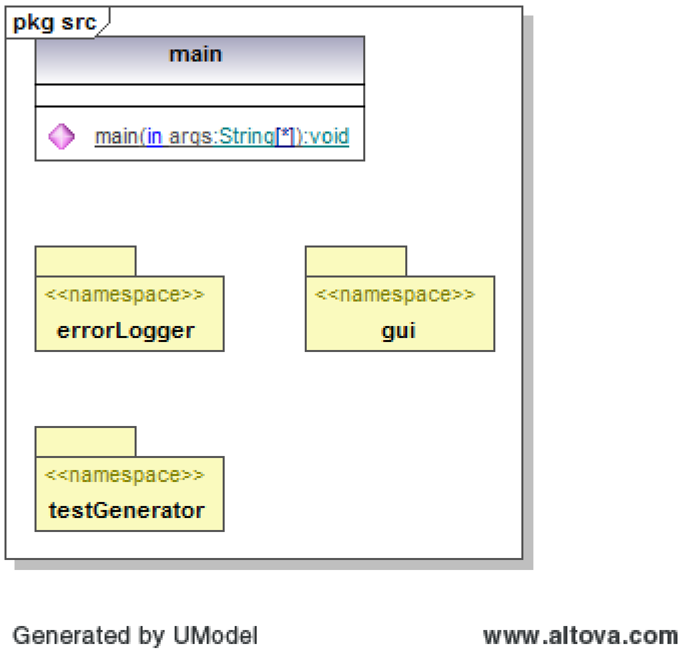
1.2.1 Package Structure

This section details the key packages of the Automatic JUnit Creation tool.

1.2.1.1 Main

The main package is responsible for starting the GUI and initializing the project. It also contains three sub-packages:

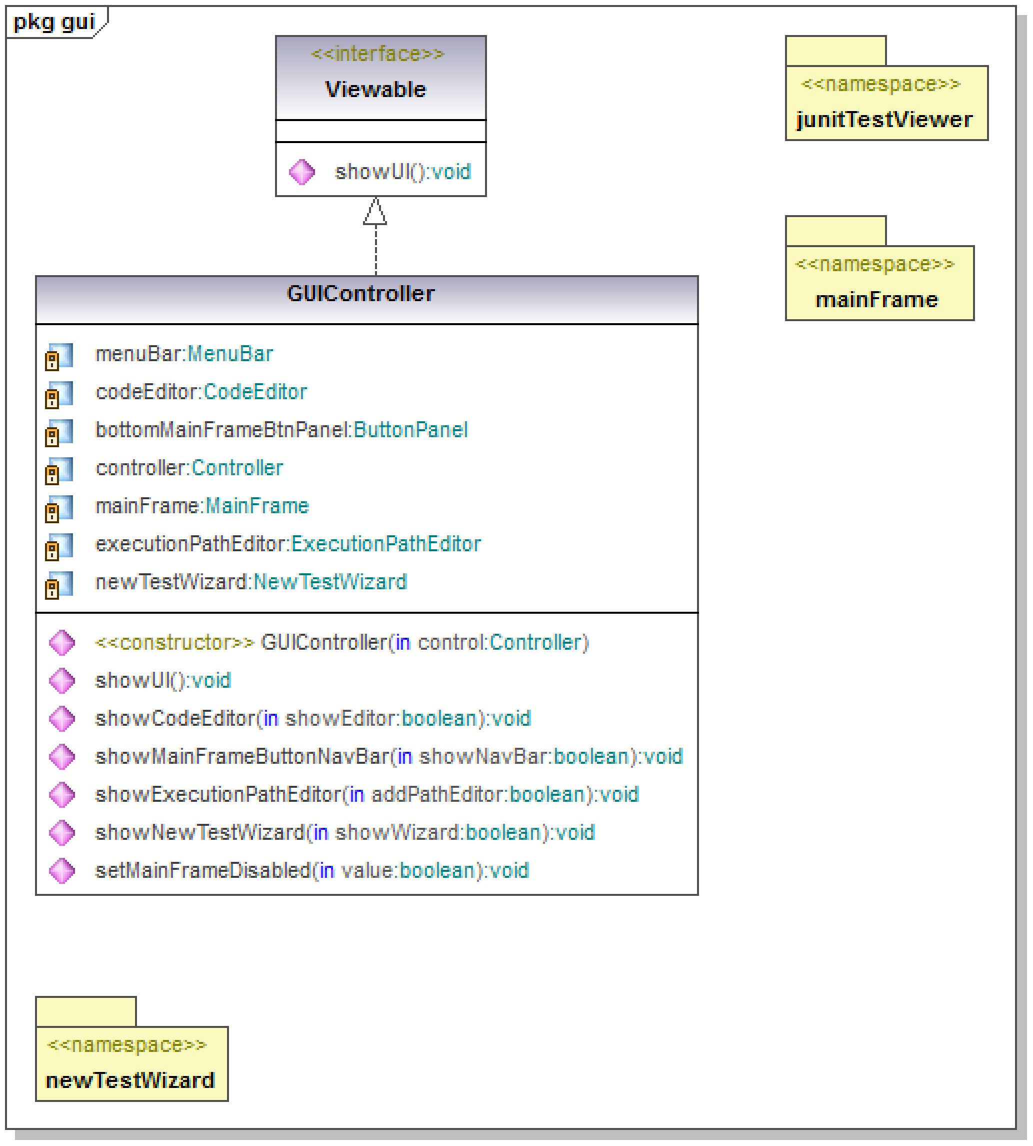
- errorLogger – Responsible for handling application errors
- gui – responsible for containing all user interaction elements
- testGenerator – responsible for interpreting java code and building tests



1.2.1.2 GUI

The gui package is responsible for holding all graphical components of the system. One of the key design elements of this package is the GUIController. This controller allows programmers a clean interface for controlling all graphical components within the project. If a programmer for instance wants to show the code editor which is a different GUI component all they have to do is access the GUIController object and call showCodeEditor(true).

- GUIController – allows all visual components to be controlled from one place
- Viewable – interface for all gui components
- junitTestViewer provides a way to view the generated unit test
- mainFrame – holes the main graphical component that all other components fit into

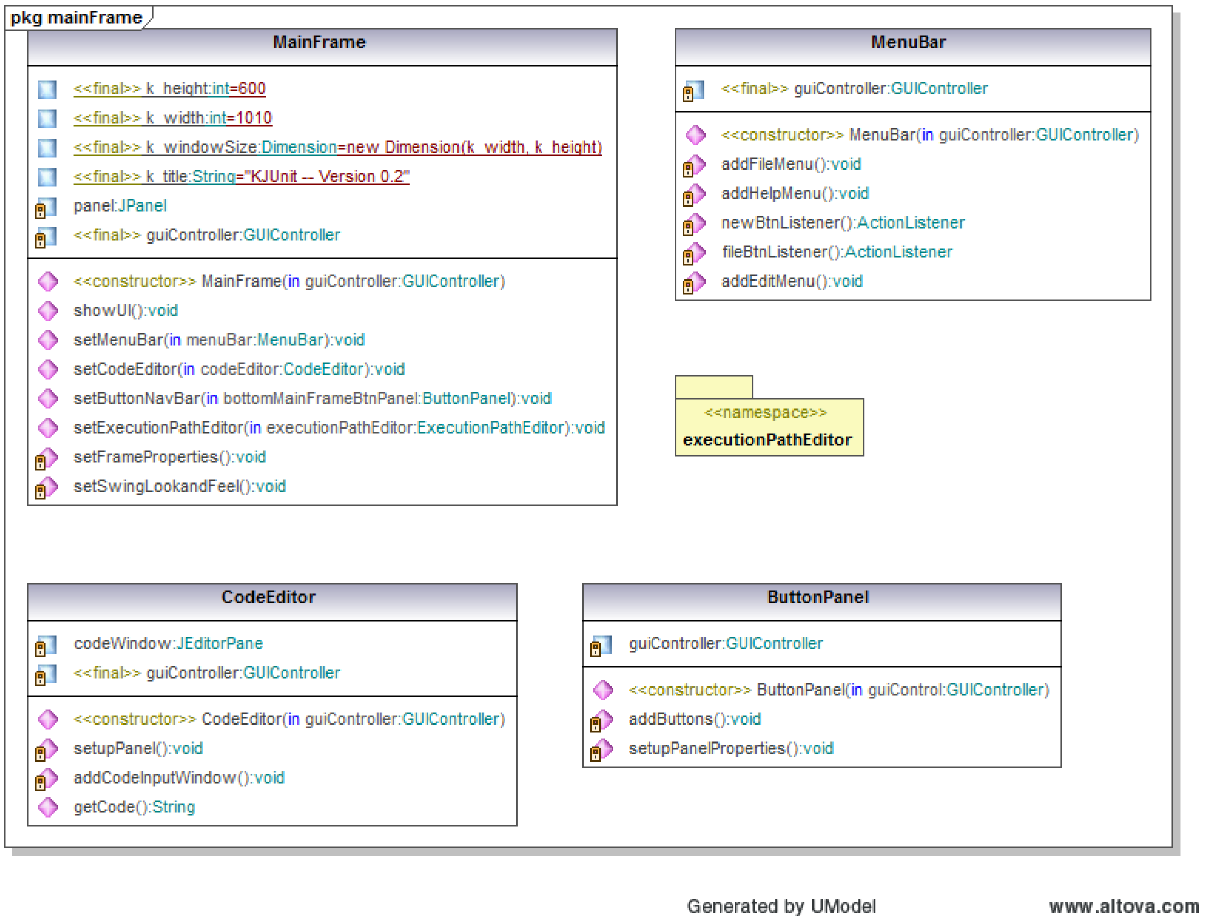

pkg gui
<<interface>>
Viewable
showUI():void
<<namespace>>
junitTestViewer
<<namespace>>
mainFrame
GUIController
menuBar:MenuBar
codeEditor:CodeEditor
bottomMainFrameBtnPanel:ButtonPanel
controller:Controller
mainFrame:MainFrame
executionPathEditor:ExecutionPathEditor
newTestWizard:NewTestWizard
<<constructor>> GUIController(in control:Controller)
showUI():void
showCodeEditor(in showEditor:boolean):void
showMainFrameButtonNavBar(in showNavBar:boolean):void
showExecutionPathEditor(in addPathEditor:boolean):void
showNewTestWizard(in showWizard:boolean):void
setMainFrameDisabled(in value:boolean):void
<<namespace>>
newTestWizard

Generated by UModel www.altova.com

1.2.1.3 Main Frame

The mainFrame package is responsible for providing the main graphical user interface.

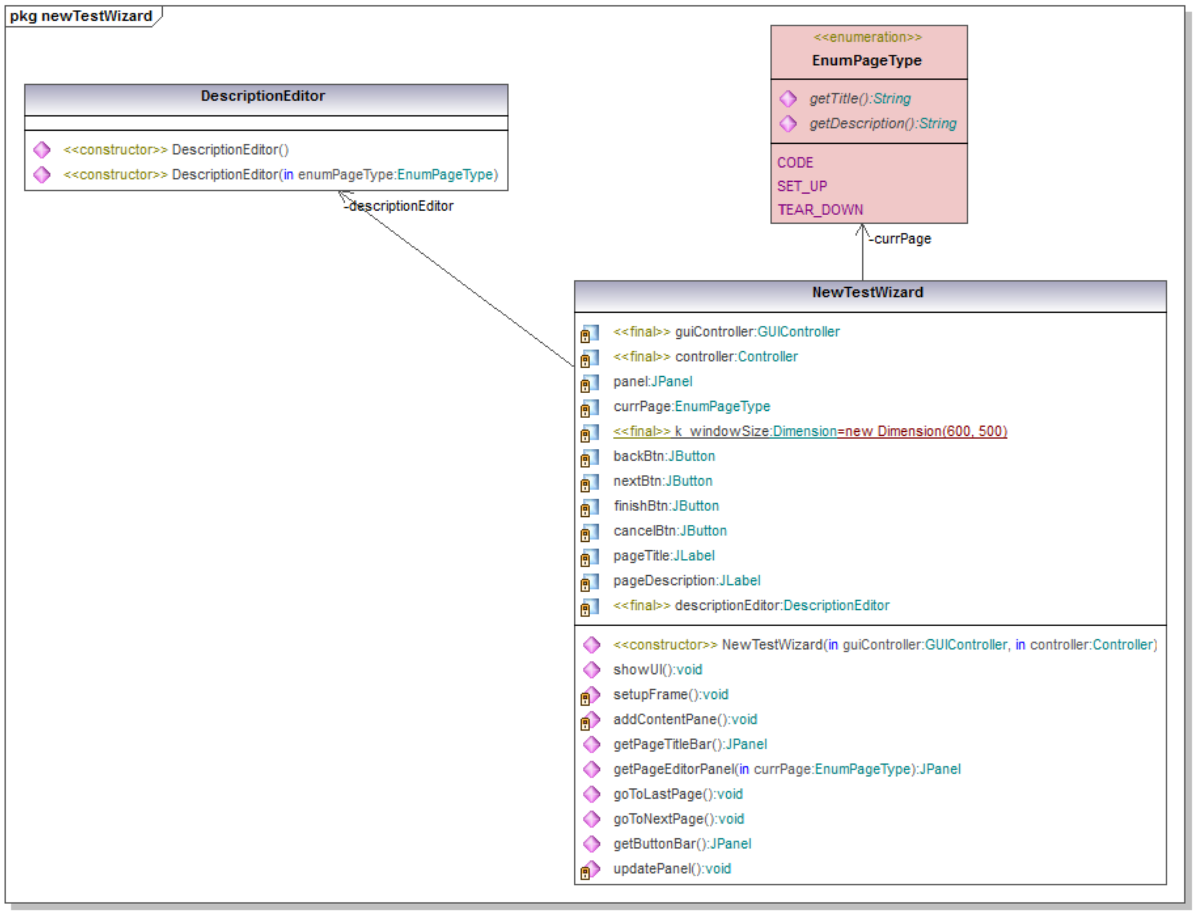
- MainFrame -- class essentially provides a container for all other graphical elements to be attached.
- MenuBar -- provides a swing menu bar for the MainFrame
- CodeEditor -- provides a class to show the user input source code
- ButtonPanel -- provides a component with buttons for changing methods in the main frame


pkg mainFrame
MainFrame
<<final>> k_height:int=600
<<final>> k_width:int=1010
<<final>> k_windowSize:Dimension=new Dimension(k_width, k_height)
<<final>> k_title:String="KJUnit -- Version 0.2"
panel:JPanel
<<final>> guiController:GUIController
<<constructor>> MainFrame(in guiController:GUIController)
showUI():void
setMenuBar(in menuBar:MenuBar):void
setCodeEditor(in codeEditor:CodeEditor):void
setButtonNavBar(in bottomMainFrameBtnPanel:ButtonPanel):void
setExecutionPathEditor(in executionPathEditor:ExecutionPathEditor):void
setFrameProperties():void
setSwingLookandFeel():void
MenuBar
<<final>> guiController:GUIController
<<constructor>> MenuBar(in guiController:GUIController)
addFileMenu():void
addHelpMenu():void
newBtnListener():ActionListener
fileBtnListener():ActionListener
addEditMenu():void
<<namespace>>
executionPathEditor
CodeEditor
codeWindow:JEditorPane
<<final>> guiController:GUIController
<<constructor>> CodeEditor(in guiController:GUIController)
setupPanel():void
addCodeInputWindow():void
getCode():String
ButtonPanel
guiController:GUIController
<<constructor>> ButtonPanel(in guiControl:GUIController)
addButtons():void
setupPanelProperties():void
Generated by UModel
www.altova.com

1.2.1.4 New Test Wizard

The new testWizard package holds the wizard that appears when the user wants to create a new test.

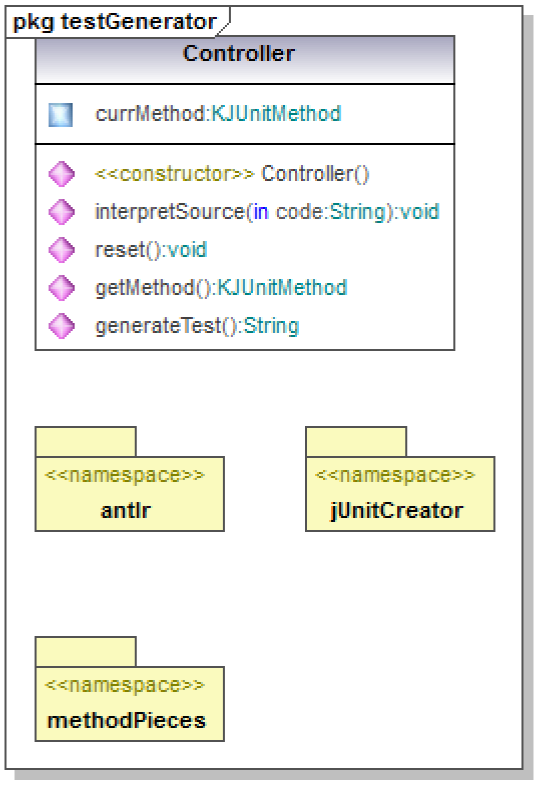
- DescriptionEditor – provides a scrollable box for viewing page directions
- EnumPageType – allows the current page to be tracked
- NewTestWizard – responsible for changing the pages in the wizard.



1.2.1.5 Test Generator

The testGenerator package is responsible for the underlying logic of the application. Underlying logic refers to anything that reads the user specified source file or any user created test values.

- Controller – A class that allows all sub-components for the testGeneator package to be manipulated form a single source.
- Antlr – holds code generated by the antlr framework that parses and evaluates the user source file
- jUnitCreator – make a JUnit test based on user input values
- methodPieces – holds the parts that make a JUnit test

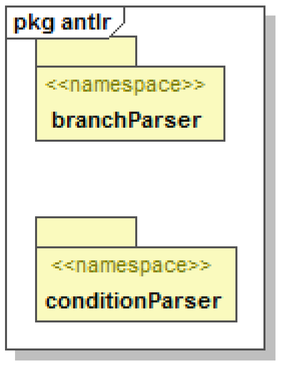


Generated by UModel www.altova.com

1.2.1.6 Antlr

This package is responsible for handling any portions of the project that deal with interpreting or evaluating the user specified source file.

- branchParser – Finds all execution paths in the system
- conditonParser – takes a statement and determines it logical conditions and negates them

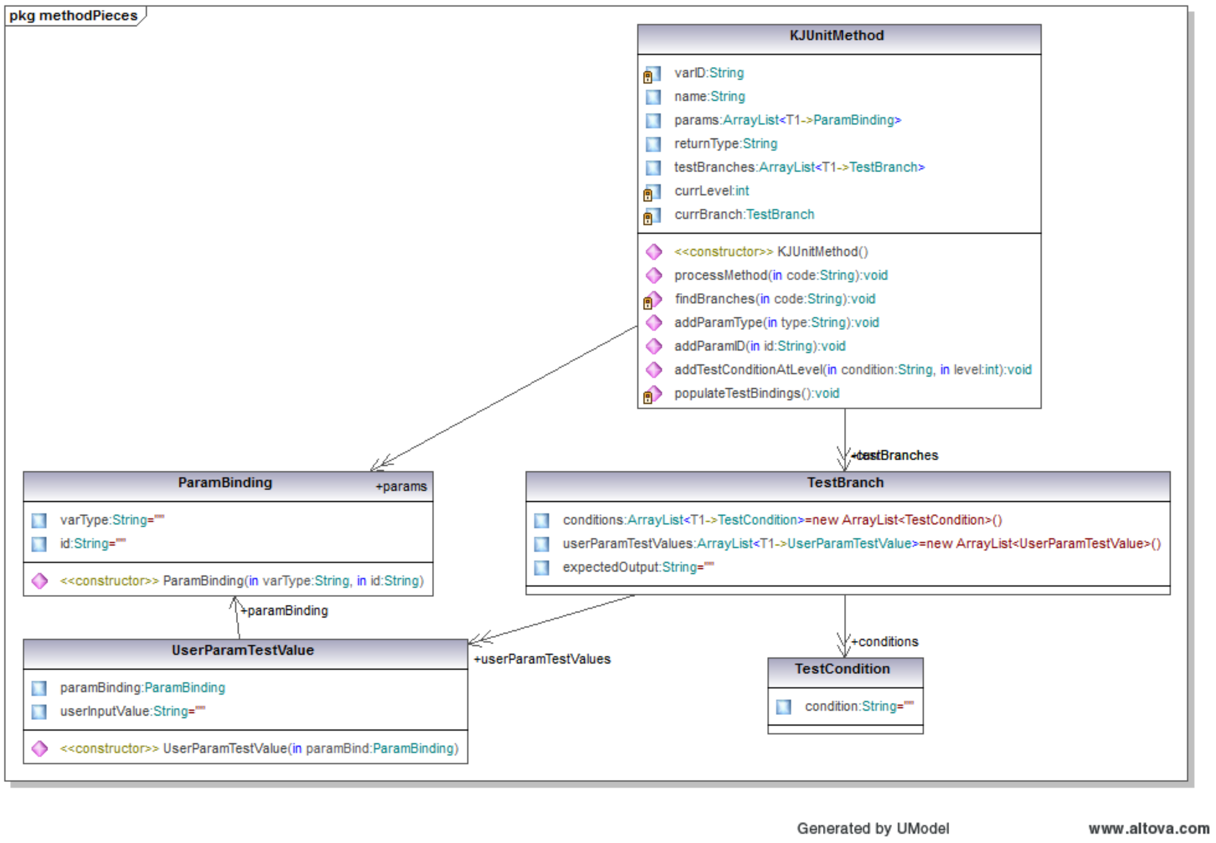


Generated by UModel www.altova.com

1.2.1.7 Method Pieces

The package methodPieces holds all the parts that make up a JUnit test within the system.

- KJunitMethod – models a method within the tool
- TestBranch -- a single execution path through the method
- TestCondition – a portion of the execution path
- ParamBinding – represents a parameter for the method signature
- UserParamTestValue – represents a test value for the given ParamBinding


pkg methodPieces
KJUnitMethod
varID:String
name:String
params:ArrayList<T1->ParamBinding>
returnType:String
testBranches:ArrayList<T1->TestBranch>
currLevel:int
currBranch:TestBranch
<<constructor>> KJUnitMethod()
processMethod(in code:String):void
findBranches(in code:String):void
addParamType(in type:String):void
addParamID(in id:String):void
addTestConditionAtLevel(in condition:String, in level:int):void
populateTestBindings():void
+testBranches
+params
ParamBinding
varType:String=""
id:String=""
<<constructor>> ParamBinding(in varType:String, in id:String)
TestBranch
conditions:ArrayList<T1->TestCondition>=new ArrayList<TestCondition>()
userParamTestValues:ArrayList<T1->UserParamTestValue>=new ArrayList<UserParamTestValue>()
expectedOutput:String=""
+paramBinding
UserParamTestValue
paramBinding:ParamBinding
userInputValue:String=""
<<constructor>> UserParamTestValue(in paramBind:ParamBinding)
+userParamTestValues
+conditions
TestCondition
condition:String=""
Generated by UModel
www.altova.com