17654 – Analysis of Software Artifacts

Tool Evaluation
EclipsePro

Christopher Nelson – crnelson
J. Luis Rios – jriostre
Chung-Hao Shih – chunghas

Apr 27th, 2006

Describe the Tool

EclipsePro is an Eclipse plug-in that provides a set of tools for analyzing source code, generating test cases, and monitoring test coverage.

The analysis tool is based on heuristic rules that the user may configure to

- Identify dead code
- Warn for possible logical or semantic errors that may lead to actual bugs
- Discover bad coding practices inhibiting quality attributes such as performance and security
- Improve the readability and decrease the complexity of the code
- Discover the source of internationalization issues
- Enforce a team to comply with a particular coding style and convention.

Depending on the necessities of the developer, the application of all these rules may have a big impact on quality attributes such as performance, maintainability, testability, security, usability and portability.

The tool also provides the ability to automatically generate unit test cases for the developers using very basic unit test patterns and regression test mechanisms. Furthermore, the tool provides metrics for test coverage in order to give a clearer idea of how much code these generated unit tests are testing.

Describe the Experimental Setup

In order to proceed with the evaluation, the following components were set up (Please refer to the resources section for the URLs where these tools can be downloaded)

- Eclipse 3.1.2
- OSATE 1.2.3 Plug-in for Eclipse
- EclipsePro 4.3.1

Additionally the OSATE code base and a prototype using the infrastructure provided by OSATE were used as inputs to evaluate *EclipsePro*.

Analysis of OSATE code base

The goals of this part of the evaluation were to

- Get a better understanding of the benefits that *EclipsePro* offers

- Detect issues such as bad practices and potential bugs in the OSATE code base
- Identify areas in OSATE that may be improved
- Evaluate how the use of EclipsePro may have an impact in our Studio project

Once the tools were in place, we determined what kind of issues were the most important to us to analyze. Below is a list of these issues grouped by the quality attributes they address that the team must satisfy in our studio project.

Performance	
Append string	The appending of a string literal must be replaced by the appending of a character literal
Concatenation in appending method	A concatenation of strings is incorrectly used in the append method of the StringBuffer class
Debugging code	Calls to System.out are kept in production code
Define initial capacity	The initial capacity of Collections and StringBuffer is not tailored to the specific case
Favor static member classes over non-static	Member classes that does not reference the enclosing class must be static
Method invocation in loop condition	A method call is used in every iteration of the loop to evaluate the exit condition
Variable declared within a loop	A variable is continually declared and initialized within each iteration of the loop
Maintainability	
Block depth	A measure of complexity based on how many levels of nested blocks exist in a particular block of code
Close where created	Streams and sockets must be closed in the same method where they are instantiated
Empty catch clause	Empty catch clauses make harder the debugging of an application
Empty method	Methods with no implementation (non-abstract) distract the developer
File length	Lengthy files makes the code harder to understand

Include implementation version	A deployed system must have a way to differentiate itself from other versions
Log exceptions	A system is more testable and traceable if faulty conditions are recorded
Non-protected constructor in abstract type	It makes no sense to have a non-protected constructor in an abstract class
Protected method in final class	Final classes cannot be derived so it is not reasonable to have a protected method
Source length	Limiting the size of constructors, initializers, and methods in order to improve the readability of the code
String literals	String literals are a source of problems when the application is meant to be used in different locales
String method usage	The equals method of the String class must not be used in applications that need to be localized
Unused method	Unused methods make the code harder to understand
Variable should be final	When the value of a variable does not change it should be declared final to indicate so
Variable usage	A variable is assigned a value but this value is never used
Correctness	
Float comparison	Because of rounding problems float values should not be compared with the equality and inequality operators
Hiding inherited fields	A class defines a field with the same name as a field declared by its parent class making impossible to access the parent field
String comparison	Strings should not be compared with the equality and inequality operators

Table 1 – Analysis description

Unit Tests for Prototype using OSATE code base

The goals of this part of the evaluation were to

- Get a better understanding of EclipsePro's unit test generation capabilities.
- Detect potential bugs in our team's code base.

In our Studio project and during its implementation phase, the team intends to follow a test-driven development process. As part of this process, the generation of unit tests for all the code plays a very important role. Therefore, the functionality provided by EclipsePro for generating unit tests may help us generate additional tests cases that could improve the quality of the development.

The target code for the evaluation of this tool was taken from an OSATE plug-in prototype we had previously written. This prototype is a small program consisting of 942 lines of java code.

The prototype is composed of the following java source files

```
edu.cmu.sei.aadl.plugindemo.ComponentPortGroupCandidateSwitch.java
edu.cmu.sei.aadl.plugindemo.ConnectionPortGroupCandidateSwitch.java
edu.cmu.sei.aadl.plugindemo.PlugindemoPlugin.java
edu.cmu.sei.aadl.plugindemo.PortGroupCandidate.java
edu.cmu.sei.aadl.plugindemo.actions.CheckPortGroupCandidate.java
```

The following test cases were generated by *EclipsePro*

```
edu.cmu.sei.aadl.plugindemo.ComponentPortGroupCandidateSwitchTest.java
edu.cmu.sei.aadl.plugindemo.ConnectionPortGroupCandidateSwitchTest.java
edu.cmu.sei.aadl.plugindemo.PlugindemoPluginTest.java
edu.cmu.sei.aadl.plugindemo.PortGroupCandidateTest.java
edu.cmu.sei.aadl.plugindemo.actions.CheckPortGroupCandidateTest.java
```

Describe Qualitative and Quantitative Data Gathered

The OSATE analysis was executed in

Time	Lines of Code
2 minutes 33 seconds	59,510

Table 2 – Execution time and source code size

The number of issues is summarized in the following table.

Performance	
Rule Name	**Number of Issues**
Append string	9
Concatenation in appending method	2
Debugging code	4
Define initial capacity	22
Favor static member classes over non-static	6

Method invocation in loop condition	10
Variable declared within a loop	339
Total Performance	**395**
Maintainability	
Rule Name	**Number of Issues**
Block depth	38
Close where created	3
Empty catch clause	2
Empty method	53
File length	7
Include implementation version	1
Log exceptions	10
Non-protected constructor in abstract type	6
Protected method in final class	4
Source length	32
String literals	1970
String method usage	50
Unused method	2
Variable should be final	7
Variable usage	8
Total Maintainability	**2158**
Correctness	
Rule Name	**Number of Issues**
Float comparison	3
Hiding inherited fields	193
String comparison	6
Total Correctness	**202**
Total Evaluation	**2755**

Table 3 – Statistics of analyzed code

Analysis of special conditions such as null dereferencing and aliasing were tested. Unfortunately, *EclipsePro* does not perform data flow analysis and was not able to report issues in these two categories.

The following table presents the coverage of the generated unit tests. This is represented in terms of how many methods, lines, blocks and instructions were covered by the generated tests.

Generated code	Method	Lines	Blocks	Instructions
ComponentPortGroupCandidateSwitchTest	0/2	0/6	0/2	0/19
ConnectionPortGroupCandidateSwitchTest	0/6	0/99	0/48	0/482
PlugindemoPluginTest	5/6	8/16	8/12	21/41
PortGroupCandidateTest	5/6	29/150	15/84	88/705
CheckPortGroupCandidateTest	3/4	3/15	4/8	9/44
Average	54%	13%	24%	9%

Table 4 – Coverage of generated tests

From the coverage range of the unit tests, we can tell that if we blindly trust the quality of the automatically generated unit tests that would mean that we would be covering only 9% of all the instructions of the total code. This is extremely low. There are execution paths that are not covered and therefore human rechecks are necessary to improve the coverage of the generated unit tests.

There are several reasons for the low code coverage. For example, code referencing interfaces could not be analyzed because this tool does not take into account run-time behavior and statically there is no way to infer what actual code will be executed.

Benefits of the Tool

Configurable rules

Rules for analyzing source code are configurable including severity and parameters for their tailoring

The processing time is good

The time EclipsePro took to analyze the source code and generate the unit tests is very reasonable compared to the benefits one can get.

Integrate many analysis techniques in one tool

Analysis of source code, unit test generation, code coverage analysis, and metrics of the source code can all be achieved with *EclipsePro*.

Automatically generates the framework of the unit test

It generates the framework methods of the unit test. For example tearDown( ), setUp( ), etc. Therefore, the developer does not need to write all the basic components of the unit tests.

Automatically generate the basic test classes

EclipsePro will automatically generate the test methods by analyzing the methods within the target source class. For example, for the method abc(), it will automatically generate testAbc().

Provide a mechanism to ensure the verification of unit tests before the test is executed

Having *fail("unverified")* in the end of each automatic generated unit test method is a check to help ensure that each test method is verified before it can pass.

```
{
    MarkerReporter er = null;
    ComponentPortGroupCandidateSwitch result = new ComponentPortGroupCandidateSwitch(er);
    // add test code here
    assertNotNull(result);
    fail("unverified");
}
```

Figure 1 - Verification

Check invalid and valid parameters for each method call

For each automatically generated method, *EclipsePro* will try to give null values and other reasonable values as parameters to verify the result of each method.

```
public void testCompareTerminal_fixture_1()
    throws Exception
{
    PortGroupCandidate fixture = getFixture();
    String src = null;
    String dst = null;
    boolean result = fixture.compareTerminal(src, dst);
    // add test code here
    assertEquals(false, result);
}
```

Figure 2 - Parameters

Provide a mechanism ease regression testing

EclipsePro provides the TestAll.java component for testing every unit test at once or for testing every unit test in each of the packages.

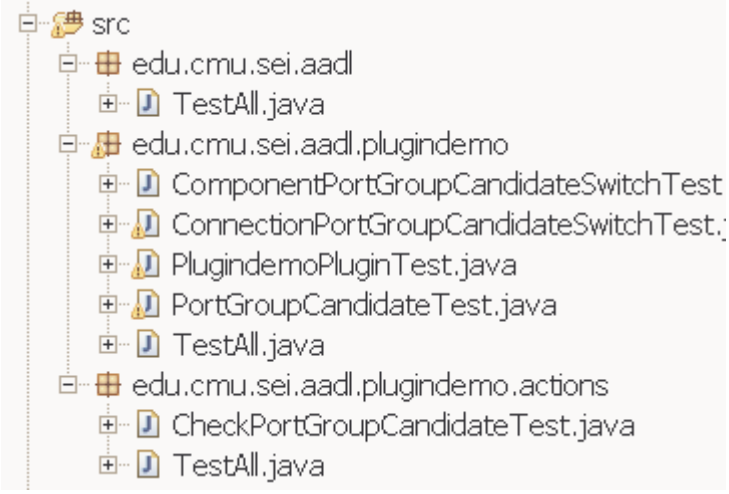

Figure 3 – Regression testing

Automatically generates comments for the unit tests

EclipsePro generates comments for each unit test method it generates. These comments include tags for @author, @see, and @return. This automatic generation of comments (and any other artifact) can boost team's productivity.

```
/**
 * The class <code>PortGroupCandidateTest</code> contains tests for the class {@link <code>PortGroupCandidate</code>}.
 *
 * @generatedBy CodePro at 4/23/06 3:30 PM
 * @author okbong
 * @version $Revision: 1.0 $
 */
public class PortGroupCandidateTest extends TestCase
{
```

Figure 4 - Comments

Test coverage

EclipsePro provides test coverage analysis. Therefore, it is helpful for detecting blocks that have not been covered by the current unit tests.

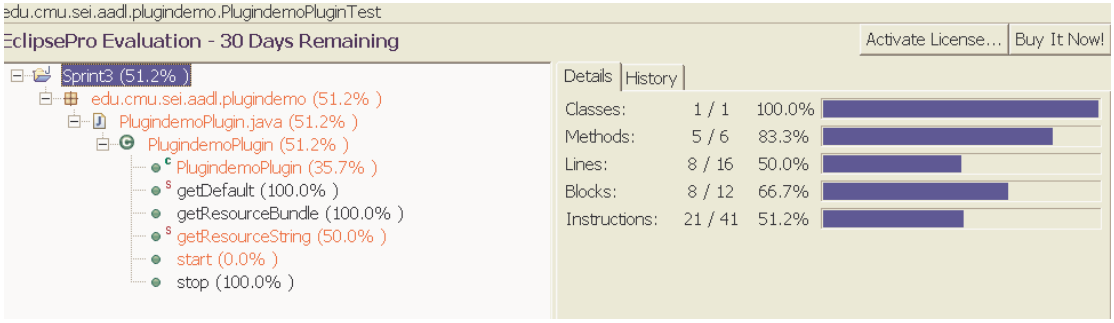

Figure 5 – Test coverage

Drawbacks of the Tool

Could not generate test cases for interfaces

If a method references an interface, EclipsePro cannot distinguish it from an actual instance of an object. The consequence of this is that it will throw a NullPointerException when generating the unit test cases. However, this actually means that the content of the method in the interface could not be analyzed. The actual implementation of the method implementation will be dynamically linked during runtime execution and that cannot be tested by the static analysis used by EclipsePro to generate the unit tests.

Missing library

The tool does not automatically include the library of the original project, which is used to generate the unit test cases. Therefore, it is necessary to manually add one by one all the required libraries.

Missing a great deal of basic unit test scenarios

The documentation that EclipsePro provides is not specific in terms of how many unit test scenarios it can generate. Many scenarios for the unit tests are missing. For example, checking other types of input values rather than only null.

False positives

- The tool may return some false positives
  - Constructors must invoke only final methods but inherited methods are not allowed
  - Constant conditional expressions such as while(true) are reported and there is not way to get rid of them
  - Hiding inherited fields does not allow to ignore certain fields such as copyright notices
  - False positives for unused fields because they are indeed being used in the body of the enclosing class
  - No able to detect that some variables must be constants since they do not change their value

Scope of the Applicability of the Tool

EclipsePro is not a substitute for tools such as Fugue, Metal, or Prefix. However, it outputs a different type of analysis that may be also important, this is, it may be used as a complement to these other tools. In terms of unit test generation the tool is useful for producing skeletons that require manual modification and in that regard it may improve the productivity of the development team.

Conclusions

Although the analysis of the source code and the test cases that resulted in the statistics shown in the tables are very useful for learning about certain types of errors, *EclipsePro* lacks some of the features found in some of the tools used in class. For example

- The type of analysis that *EclipsePro* does cannot be compared with Blast or Fugue where it is possible to define the protocol defining the contract for how a class may be correctly used. *EclipsePro* does not support the definition of protocols.
- *EclipsePro* does not provide a data flow analysis. As opposed to Metal, Prefix, and Fugue where the tool can detect issues such as locking misuses and dereference of null variables, *EclipsePro* focuses on detecting errors that do not involve run-time behavior.
- As part of this evaluation we informally compared Eclat, which is a research tool developed at MIT for the generation of test cases, with *EclipsePro*. Eclat relies on Daikon for discovering the invariants of every method. According to the limited testing we executed, Eclat generated more diverse test cases than *EclipsePro*.

On the other hand, EclipsePro was very useful for finding areas in the OSATE code base that need certain level of rework. Examples of these areas are the indiscriminate usage of string literals, poor documentation conforming to the javadoc standard, no logging in order to improve traceability, and the lack of fine-tuning such as the definition of the initial capacity of collections used throughout the application in order to improve its performance.

The test-case component of the tool has some limitations. For example

- For large and complex projects the use of EclipsePro may not be of much help because of its limitations in covering language constructs such as code referencing interfaces.
- The functionality for unit test generation is not good enough since it requires human involvement. If more static analysis features were introduced into the tool, such as in Eclat, EclipsePro would be much more powerful.

However, EclipsePro can still be used as a starting point for use case generation. Although according to our evaluation, the generated unit tests are not as reliable in terms of coverage as they must be, these initial unit tests can be used as a skeleton to create further test cases.

Furthermore, the test coverage offered by the tool is a good very useful for seeing how well written the test cases are. It gives the developer a clear overview of what the test cases have covered and what is actually missing.

In summary, we will likely use the tool a couple times during our summer semester to determine if there are sections of code that need particular attention for code reviews. Selection of these sections of code for review would be based on the number of warnings/errors *EclipsePro* generates on a given code section. We do not plan on incorporating the tool into our daily development processes.

Resources

Where to get the tools used in this evaluation

- Eclipse 3.1.2 [http://www.eclipse.org/downloads/]
- OSATE 1.2.3 Plug-in for Eclipse [http://www.aadl.info/]
- EclipsePro 4.3.1 [http://www.instantiations.com/eclipsepro/]