17-654: Analysis of Software Artifacts

Tool Evaluation:
EclipsePro Audit by Instantiations

Teams Diversity + Team13

Members:
Adlan Israilov
Brian Krausz
Majid Alfifi
Mohit Bhonde
Raul Vejar

Index

Index ... 2
About the Tool ... 3
About the Experiments ... 4
1) “Hnefatafl”: Personal size, stand alone application ... 4
Rationale ... 4
Initial setup ... 4
Customization ... 4
2) “Ævol”: Medium size project, plug in application ... 4
Rationale ... 4
Initial setup ... 5
Customization ... 5
3) “SeisPM”: Large size project, enterprise application ... 5
Rationale ... 5
Setup ... 5
Customization ... 6
About the Results ... 8
1) “Hnefatafl”: Personal size, stand alone application ... 8
Metrics obtained ... 8
False positives ... 9
True irrelevant positives ... 9
2) “Ævol”: Medium size project, plug in application ... 9
Metrics ... 9
False positives ... 10
True positives ... 11
True irrelevant positives ... 13
3) “SeisPM”: Large size project, enterprise application ... 13
Metrics ... 13
False positives ... 16
True positives ... 16
True Irrelevant Positives ... 16
Conclusions ... 17
Advantages ... 17
Disadvantages ... 17
Final Conclusions ... 18
Assignment Efforts ... 19

About the Tool

EclipsePro is a set of tools developed by Instantiations (www.instantiations.com) for Eclipse environments that contain some of the characteristics of the CodePro suite of the company which is a comprehensive toolset for java quality and testing in enterprise environments.
EclipsePro is composed of two products:

1. *EclipsePro Audit*: a code analysis tool designed to detect defects, implement basic repairs and provide reports on the findings.
2. *EclipsePro Test*: a jUnit test generator and editor which also provides code coverage functions and report generation.

Both tools cost 519 US$ each, which includes only 90 days of maintenance and updates where additional support is sold separately.
This report will be focused on EclipsePro Audit, which has recently seen (on January 25th) its 6th release and is available for trial or purchase at the company's website.
EclipsePro Audit analyzes the static code structure looking for compliance with sets of pre-loaded audit rules. The standard package of rules includes things like:

- Coding Style
- Comments
- Dead Code
- JavaDoc conventions
- Internationalization
- J2EE
- jUnit and Logging
- Naming conventions
- Performance
- Portability
- Program Complexity
- Possible Errors
- Security
- Spell Checking
- Threading

Of course, the rules can be extended with custom built ones through the rule editor.
The tool also includes a comprehensive set of over 350 quick fixes for some of the rules, which allows developers to quickly fix some of the most common mistakes.
The Eclipse integrated interface, metric generation (such as LOC, comment lines, number of functions, etc) and comprehensive reporting capabilities of the tool are designed to make code inspections and audit an easier task.

About the Experiments

Given that the main purpose of this exercise is to decide on the usefulness of the tool, and that this can change depending on the domain on which it is applied, we decided to carry out testing of the tool in different environments:

1) "Hnefatafl": Personal size, stand alone application

Rationale

We wanted to evaluate EclipsePro from a smaller perspective on a smaller codebase. For this reason we chose the Hnefatafl code the Diversity team used in the earlier assignments. We knew that this code was written to be functional rather than well-made, with reusability, security and performance relatively low on the list of priorities.

Initial setup

- Eclipse 3.4
- EclipsePro Audit 6.0
- Team Diversity Hnefatafl code

Customization

We opted to skip customizing the software and relying on the default options (under the premise of being a single coder with limited time trying to quickly use the tool).

2) "Ævol": Medium size project, plug in application

Rationale

The Ævol Studio project is a follow-up project of the previous year MSE team studio project (Architecture Evolution). That team had several requirements, one of them was to make it maintainable, so that follow-up teams would be able to easily understand and modify it (modifiability, learnability).
Also there is a scalability requirement, which just recently emerged during the Quality Attribute Workshop. It states that the tool should be able to create and process architecture diagrams of a large size, up to 100 nodes or architecture instances. Although, there was no direct call for performance (latency), scalability implies some connection with it, for example response time issue (who wants a tool which will need a several days to process your architecture diagram?)
Also, the team thought that this was a good opportunity to analyze the inherited tool on some of the quality attributes described.

Initial setup

- Eclipse Europa v3.3.2 has been installed (the project is directly dependant on this version of Eclipse).
- Source code of Architecture Evolution Tool v1.2 as checked out from project's repository.
- Downloaded and installed EclipsePro Audit 6.0.

Customization

- Disabled EclipsePro Audit v5.5.0 auto-fix feature to ensure that changes won't cause program failure.
- Created our own preferred set of rules, that is a new *Audit Set of Rules*.
    a. *AETool_audit_set* has been created based on default set of rules provided by EclipsePro Audit.
    b. Many rules has been eliminated considering the nature of AETool project (Rich Client Platform development, plug-ins)
    c. Refined set of rules ready to apply on AETool project.

3) "SeisPM": Large size project, enterprise application

Rationale

The Seismic Project Management System (SeisPM) (Saudi Aramco) is a J2EE web application which provides workflow in addition to interfaces with legacy seismic processing systems. The objective of this project is to use the tool in a real-life environment following objectives typical to the needs of any such project.
The first purpose of the exercise was to use the tool as a metric gathering system. Specifically, we need to know for each system the number of classes, lines of code, average number of methods per class, and average number of lines per method.
The second purpose is to find violations in terms of the following:

- Style: find statistics about the code conformance to java coding standards
- Adherence to Effective Java: find violations of Java best practices
- Security: find segments of code that may lead to security problems
- Performance: find segments of code that may lead to poor performance while there exist better alternatives

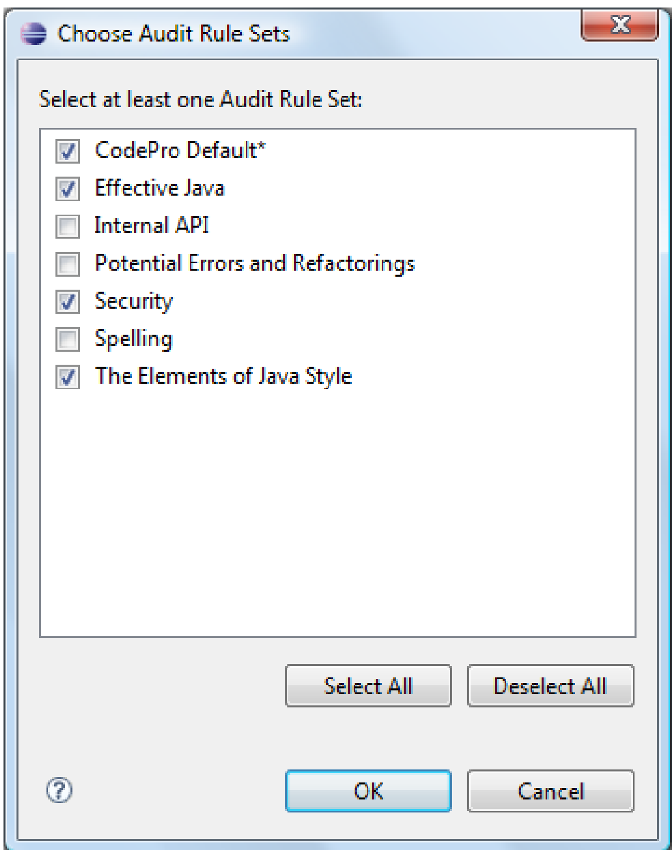
Setup

- Eclipse 3.3
- EclipsePro Audit 6.0

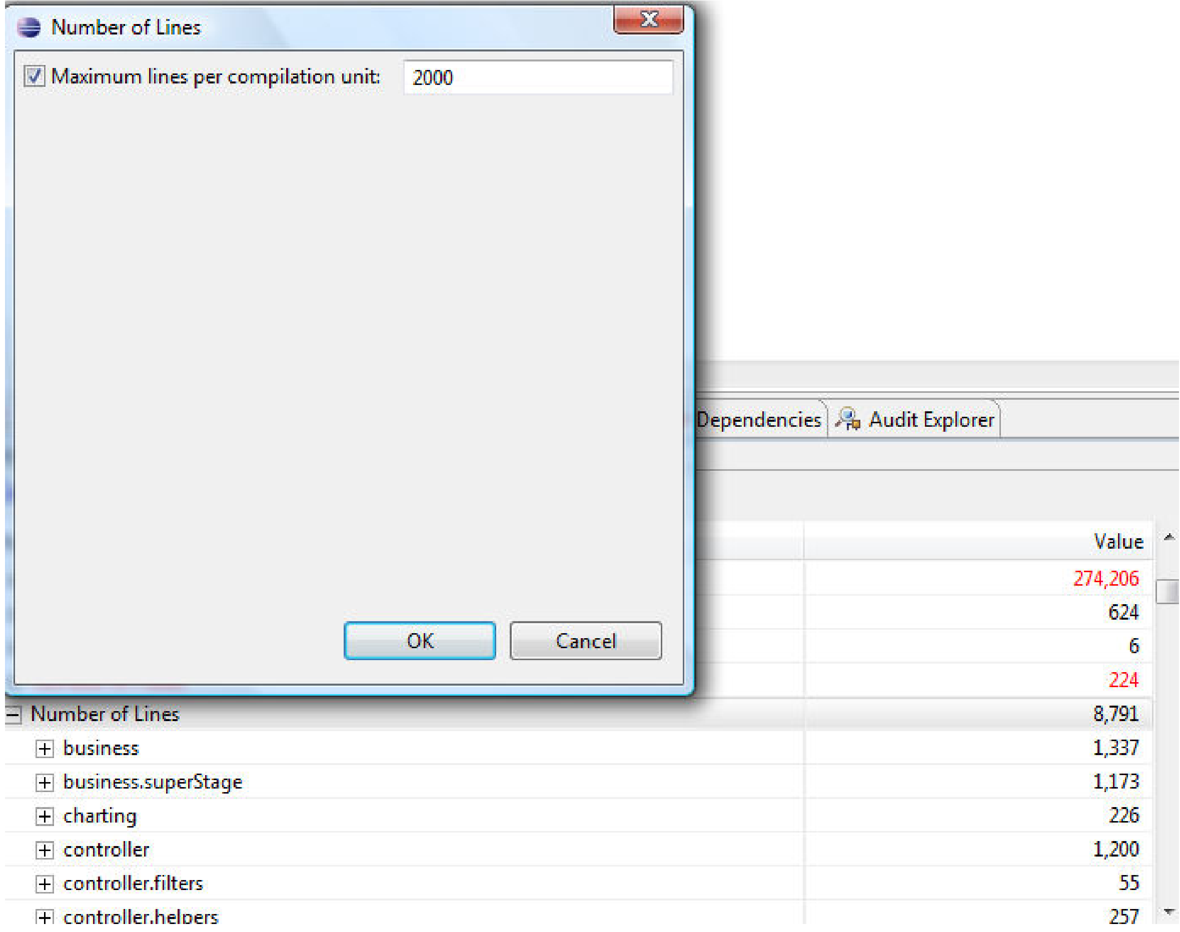
The "src" directory from the application was copied into an eclipse project. The code was compiled and needed libraries added to the path because the analysis will not run without the code being compiled in eclipse (this wasted a lot of time- the tool will just run forever without telling anything). After that, different analyses were run by following the simple menu provided by the tool.

Customization

The tool provides the following way for choosing which classification of violations to look for when running the analysis.



When calculating metrics, it's possible to customize how the tool provides violations warnings. For example the following shows how to set the max average value for "ok" number of lines. If it is not satisfied the tool will report it as violation or warning.


Number of Lines
Maximum lines per compilation unit:
2000
OK
Cancel
Dependencies
Audit Explorer
Value
274,206
624
6
224
Number of Lines
8,791
business
1,337
business.superStage
1,173
charting
226
controller
1,200
controller.filters
55
controller.helpers
257

About the Results

Each project was evaluated individually, the final output HTML reports are attached to this report as evidence (although the ones of the third project where censored to remove references to the real code because of privacy issues of the company).

1) "Hnefatafl": Personal size, stand alone application

Metrics obtained

LOC: 459
Comments: 88
Number of Lines: 806
Methods: 45

Rule	Occurrences
Always Override toString	5
Avoid Subtyping Cloneable	1
Badly Located Array Declarators	1
Clone Method Usage	1
Conditional Operator Use	1
Constants in Comparison	24
Convert Class to Interface	1
Dangling Else	6
Declare Default Constructors	1
Enumeration Constant Naming Convention	5
Exception Declaration	1
Explicit "this" Usage	7
Field Javadoc Conventions	8
File Comment	10
Hiding Inherited Fields	1
Import of Implicit Package	1
Import Order	1
Local Declarations	28
Method Javadoc Conventions	68
Missing Block	21
Multiplication Or Division By Powers of 2	1
Non-private Constructor in Static Type	2
Numeric Literals	31
Obey General Contract of Equals	1
Obsolete Modifier Usage	8
Override both equals() and hashCode()	1
Override Clone Judiciously	4
Package Naming Convention	1
Questionable Name	5
Restricted Superclasses	1
Spell Check Comments	12
Spell Check Identifiers	5
Static Field Naming Convention	1
String Concatenation	1
String Literals	4
Too Many Violations	3
Type Javadoc Conventions	11
Unnecessary Import Declarations	1
Unnecessary Return Statement Parentheses	1
Variable Declared Within a Loop	5
Variable Should Be Final	33
Variable Usage	1

False positives

There was only one false positive, though much of the testing was subjective based on coding style.

False Positive: Obey General Contract of Equals - Missing identity check (RulesMove.java – Line 40)

The explanation states that "the equals method should compare the identity of the receiver and the argument, returning true if they are the same." This is exactly what that block of code does:

```
public boolean equals(Object o) {
        if(o instanceof RulesMove) {
                RulesMove m = (RulesMove)o;
                if(m.getSource().equals(this.getSource()) &&
m.getDestination().equals(this.getDestination()))
                        return true;
        }
        return false;
}
```

True irrelevant positives

All the errors not highlighted as false positives, are considered true positives, although not relevant since they regard style issues and a project this small did not have any coding standard to enforce.

2) "Ævol": Medium size project, plug in application

Architecture Evolution project (AETool) is a set of plug-ins for Eclipse. The following tables represent analysis results for each of the most important plug-ins in AETool.

ID	Plug-in
1	edu.cmu.archevol.edit
2	edu.cmu.archevol.diagram

Metrics

Plug-in #	Execution time (seconds)	LOC	Number of comments	Number of Lines	Number of Methods
1	6	1860	273	3340	140
2	10	12104	1782	18605	907

	Number of violations detected	
Maintainability	*Plug-in #1*	*Plug-in #2*
File comment	1	107
String concatenation	29	52
Multiple return statements	28	216
Use "for" loop instead of "while" loop		6
Variable usage	1	1
Code in comments	34	84
Comment local variables	74	752
Unused field, label, method		3
Missing image file (plug-ins development)		
Undefined property (plug-ins development)		
Exception creation		13
Convert class to Interface		1
Use of instanceof should be minimized	4	175
Add methods to interface		
Close where created		1
Empty catch clauses	4	1
Variable should be final		34

String concatenation in a loop		
Performance		
Variable declared within a loop	7	106
Method invocation in loop condition	3	12
Methods should be static	13	101
Append string		
Define initial capacity	1	52
Define load factor		14
Index Arrays with integers		
Prefers interfaces to reflection		
Correctness		
Enforce Singleton property with private constructor	1	1
Illegal main method		
Float and String comparison		
Invalid loop construction		65
Empty methods, statements, classes		50
Possible null pointer	52	534
Recursively call with no check		
Use == to compare with Null		
Dangling else	1	1
Total true positives:	2588	
Total false positives:	35	

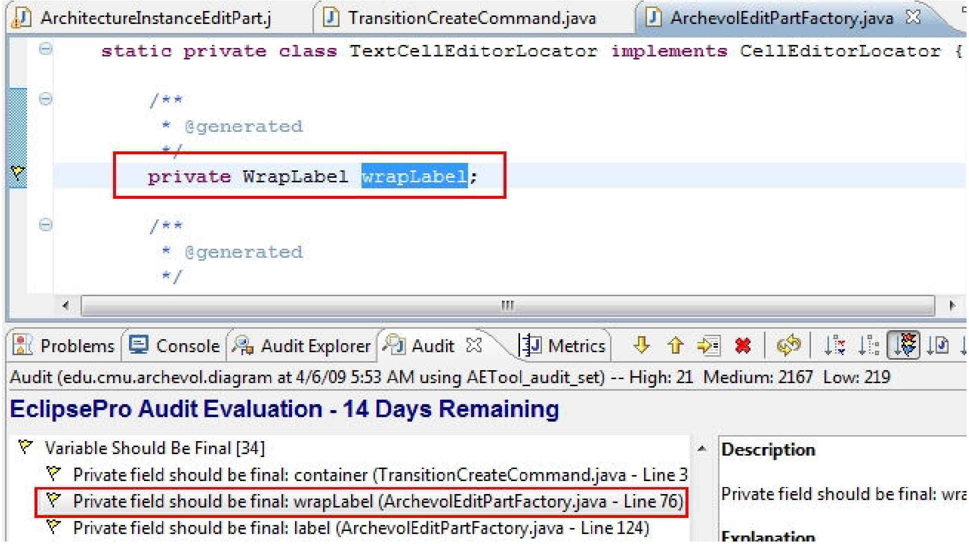
False positives

- Couldn't distinguish code comments from text comments.
  Example: No difference between ***// Text*** and ***// System.out.println();***

- Couldn't recognize variables, which should have been declared as constants (final).
  Example#1: From the table we can see that under Plug-in#1 column 'Variable should be final' category equals 0, which means EclipsePro Audit was not able to recognize the following code example:

```
ArchitectureInstanceItemProvider.java
 * <copyright>
package edu.cmu.archevol.provider;

import edu.cmu.archevol.ArchitectureEvolution;

/**
 * This is the item provider adapter for a {@link edu.cmu.archevol.ArchitectureInstance} object.
 * <!-- begin-user-doc -->
 * <!-- end-user-doc -->
 * @generated
 */
public class ArchitectureInstanceItemProvider extends ItemProviderAdapter
        implements IEditingDomainItemProvider, IStructuredItemContentProvider,
        ITreeItemContentProvider, IItemLabelProvider, IItemPropertySource {

    protected String defaultCategoryName = "<Misc>";
    /*
     * (non-Javadoc)
     *
     * @see org.eclipse.emf.edit.provider.ItemProviderAdapter#createAddCommand(org.eclipse.emf.edi
     *      org.eclipse.emf.ecore.EObject,
```

Example#2:



where

```
/**
 * @generated
 */
public TextCellEditorLocator(WrapLabel wrapLabel) {
    this.wrapLabel = wrapLabel;
}

/**
 * @generated
 */
public WrapLabel getWrapLabel() {
    return wrapLabel;
}
```

True positives

1. First example
    a. **Class**: edu.cmu.archevol.edit.ArchevolEditPlugin
    b. **Line in the code**: 42
    c. **Severity**: Medium
    d. **Warning message**: Singleton class has a constructor that is not private
    e. **Issue**: ArchevolEditPlugin can be instantiated more than once.
    f. **Why it's an issue**: AETool allows editing properties of architecture evolution diagram and ArchevolEditPlugin defines the way how those properties could be modified. Having more than two shared instances of that class does not make sense and might cause an unexpected behavior during runtime.

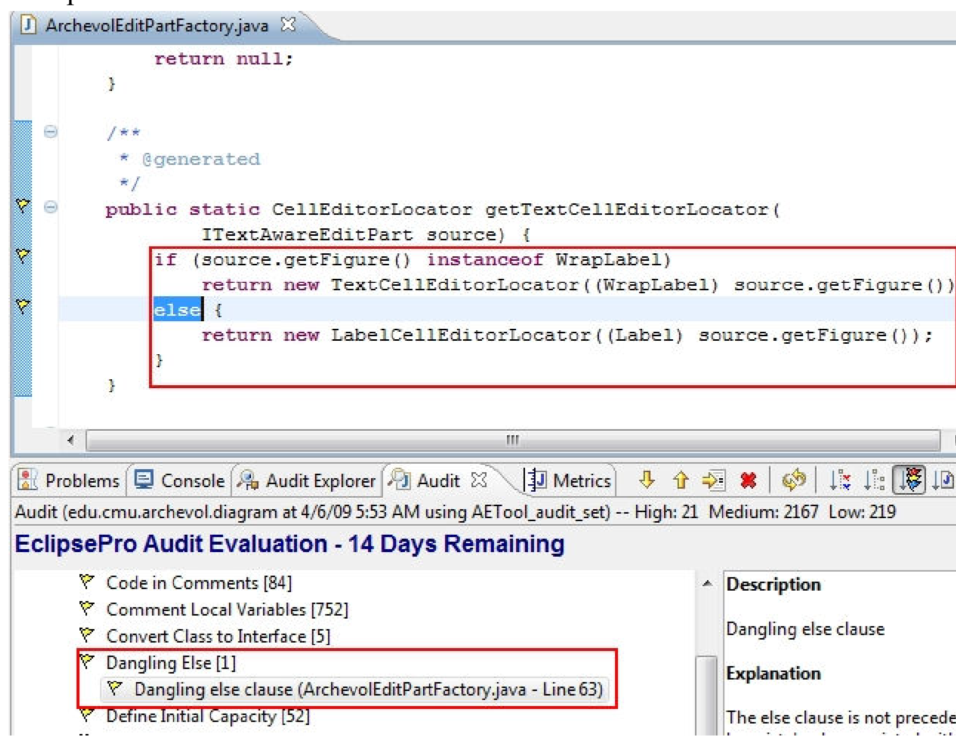
```
* <copyright>
package edu.cmu.archevol.provider;

import org.eclipse.emf.common.EMFPlugin;

 * This is the central singleton for the Archevol edit plugin.
public final class ArchevolEditPlugin extends EMFPlugin {
    /**
     * Keep track of the singleton.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @generated
     */
    public static final ArchevolEditPlugin INSTANCE = new ArchevolEditPlu(

     * Keep track of the singleton.
    private static Implementation plugin;

    /**
     * Create the instance.
     * <!-- begin-user-doc -->
     * <!-- end-user-doc -->
     * @generated
     */
    public ArchevolEditPlugin() {
        super
          (new ResourceLocator [] {
           });
    }
```

2. Second example
    a. **Class**: edu.cmu.archevol.edit.ArchitectureInstanceItemProvider
    b. **Line in the code**: 675
    c. **Severity**: Medium
    d. **Warning message**: Method invoked in a loop condition.
    e. **Issue**: aips.size() invoked in a for loop multiple times.
    f. **Why it's an issue**: Not efficient use of memory. Such type of mistakes with higher impact might affect performance of our project.

```
ArchitectureInstanceItemProvider.java
                .getArchitectureinstancepaths();

        if (styles == null) {
            System.out.println("No style to be added.");
            return;
        }

        for (EvolutionStyle style : styles) {
            String contributorName = style.getContributionname();
            Collection<String> propNames = ModelExtenderImpl.getInstance()
                    .getPathInstancePropertiesNames(contributorName);
            if (propNames != null && propNames.size() > 0) {
                for (int i = 0; i < aips.size(); i++) {
                    Path path = aips.get(i).getPath();
                    String pathName = getName(path);

                    String focusedPath = "P2";
                    if(focusedPath == null)
                        addDynamicAipPropertyDescriptor(ai,style,pathName,i); /
                    else if(!focusedPath.equals(pathName))
                        addDynamicAipPropertyDescriptor(ai, style, pathName, i)

                }
            }
        }

    }
```

True irrelevant positives

- Threw a "Dangling else" warning, when it was not needed. It is correct to say it is dangling because the if part is not "blocked", but it is harmless and not a violation of the coding style.
  Example:



3) "SeisPM": Large size project, enterprise application

Metrics

Metric Name	Value
Abstractness	2.5%
Average Block Depth	1.13
Average Cyclomatic Complexity	1.86
Average Lines Of Code Per Method	9.48
Average Number of Constructors Per Type	0.34
Average Number of Fields Per Type	4.09
Average Number of Methods Per Type	11.71
Average Number of Parameters	1.00
Comments Ratio	19.9%
Efferent Couplings	188
Lines of Code	25,877
Number of Characters	1,410,297
Number of Comments	5,152
Number of Constructors	69
Number of Fields	1,087
Number of Lines	43,283
Number of Methods	2,331
Number of Packages	35

Number of Semicolons	13,859
Number of Types	199
Weighted Methods	5,921

Security Violations

The tool found almost 976 security violations, 70 of them are regarded as "High" and the rest as "Medium"

Violation Counts by Severity

Violation Severity	Violation Count
High	70
Medium	906
Low	0

Violations by Audit Rule

Audit Rule	Count
Avoid Inner Classes	9
Avoid Package Scope	84
Command Injection	2
Cross-Site Scripting	1
Deprecated Method Found	8
Deserializeability Security	186
Do Not Implement Serializable	17
Don't Return Mutable Types	311
Enforce Cloneable Usage	137
Instance Field Security	88
Log Forging	2
Missing Catch of Exception	10
Mutability Of Arrays	2
Path Manipulation	1
Process Control	2
Request Parameters In Session	61
Serializeability Security	32
Static Field Security	21
Too Many Violations	1
Use of Random	1

"Style" Violations

The following shows the results of style analysis.

Violation Counts by Severity

Violation Severity	Violation Count
High	190
Medium	4,822
Low	2,305

Violations by Audit Rule

Audit Rule	Count
Block Depth	46
Brace Position	208
Constant Field Naming Convention	39
Constructors Only Invoke Final Methods	10
Cyclomatic Complexity	46
Dangling Else	42
Empty Catch Clause	40
Explicit "this" Usage	1,165
Field Javadoc Conventions	613
File Comment	189
Instance Field Naming Convention	22
Instance Field Visibility	88
Large Number of Fields	35
Large Number of Methods	29
Large Number of Parameters	50
Line Length	1,138
Local Variable Naming Convention	234
Method Javadoc Conventions	1,507
Method Naming Convention	46
Missing Block	141

Non-terminated Case Clause	3
Numeric Literals	291
Package Javadoc	29
Package Naming Convention	8
Package Prefix Naming Convention	29
Questionable Name	33
Redundant Assignment	1
Source Length	27
Space Around Operators	665
Too Many Violations	156
Type Javadoc Conventions	295
Use equals() Rather Than ==	14
Variable Usage	29
White Space Usage	49

Violations of guidelines from "Effective Java"

The classification "Effective Java" is not exclusive as there are other violations that also come from "Effective Java" such as security and performance violations. However, when "Effective Java" is chosen then all violations from other categories that are based on "Effective Java" book will be reported.

The following are the results:

Violation Counts by Severity

Violation Severity	Violation Count
High	85
Medium	3,980
Low	3

Violations by Audit Rule

Audit Rule	Count
Allow compareTo to Throw Exceptions	1
Always Override toString	173
Avoid null Return Values	2
Boolean Method Naming Convention	66
Constant Field Naming Convention	39
Constructors Only Invoke Final Methods	10
Declare As Interface	328
Empty Catch Clause	35
Favor Static Member Classes over Non-Static	1
Field Javadoc Conventions	613
Instance Field Naming Convention	19
Large Number of Parameters	60
Local Variable Naming Convention	231
Method Javadoc Conventions	1,541
Method Naming Convention	15
Method Parameter Naming Convention	162
Minimize Scope of Local Variables	177
Non-private Constructor in Static Type	4
Obey General Contract of Equals	9
Overloaded Methods	10
Override both equals() and hashCode()	8
Package Naming Convention	8
Package Prefix Naming Convention	29
Prefer Interfaces To Reflection	2
Questionable Name	33
Reusable Immutables	9
Static Field Naming Convention	12
String Concatenation in Loop	16
String Created from Literal	2
Too Many Violations	52
Type Javadoc Conventions	295
Unnecessary Exceptions	105
Use Interfaces Only to Define Types	1

False positives

The tool found the following as violating the style standards when it's not.
Invalid boolean method name: "contains" should be prefixed with 'can', 'equal', 'equals', etc.

	15	public static boolean **contains**(String str1, String str2)
	16	{
	17	return str1.indexOf(str2) != -1;
	18	}

True positives

The tool provided very useful information for parts of the code that may be a security problem. Such as Log Forging where information from HTTP Requests is logged to the log file if there is a problem. This is a security risk because hackers can send any suspicious code that can get to the log and may do harm. There were 2 such cases.

The tool found a very important violation (True Positive) which is the following code:

	74	public void setListOfStages2DRecords(ArrayList listOfStages2DRecords) {
	75	**listOfStages2DRecords = listOfStages2DRecords;**
	76	}

This may have gone untested but the intention is to use this keyword for the left side field. It's not clear why the tool considered this as style rather than something else.

The following are some of the violations with their corresponding item in "Effective Java" book.

Issue reported	Item # in "Effective Java"
compareTo() in Server.java doesn't through ClassCastException	**11**: Consider implementing Comparable
Two cases in GetImageServlet.java where null was returned instead of an empty list.	**27**: Return zero-length arrays, not nulls
in many methods, the following used: ArrayList l = new ArrayList(); rather than List l = new ArrayList();	**34**: Refer to objects by their interfaces
some exceptions handlers just do nothing	**47**:Don't ignore exceptions
INTViewerConstants is an interface that has only constants	**17**: Use interfaces only to define types

True Irrelevant Positives

However there was few true positives but irrelevant. For example, the project team is aware of cross-site scripting and that is how the system deals with the legacy systems that are running on several UNIX machines scattered in the environment.

Violation	Recommendation	Severity	Resource	Line
Cross-Site Scripting	User data should never directly be put onto a web site, the path should be eliminated.	High	ExecRemoteCmd.java	60

Conclusions

Advantages

- Very simple to use "out of the box". Item functions are very obvious and clearly labeled.
- The tool has "Explain" feature which exists for every audit rule and color-coded flags for severities and categories. This provides very useful summary about that audit and what it means so developers can "fix" the code.
- The tool provides several metrics such as comments ratio, average lines of code per method etc which is not available on Eclipse by default (a much-welcomed feature). It would be even more useful if more information is provided about what is the best practice to have for such metrics. This is in one way similar to what the tool already provides as default values for what is considered "high" violation as shown earlier.
- A large amount of customization, while still providing a reasonable default rules set. All of the CodePro's audit rules are defined using the extension points. We can easily add our new rules by adding a plug-in with a new set of audit rules. Java doc is provided.
- Rules sets provide a great flexibility of sharing knowledge from project to project and enforcing standards inside the team (if audition is done constantly)
- Rules sets can be customized during the audit, by switching on or off some rules
- Auto fix function allows user to make automatic fixes for some specific type of code style violations (significantly reduces time and automates process of fixing errors)

Disadvantages

- Reporting is simplistic. For example, CSV reports aren't in a format that allows for calculations to be done on them. It would be good if it's possible to include the description of the summaries with the reports generated.
- Some rules are partially redundant, which can get bothersome. Example: Dangling Else and Missing Block often go hand-in-hand
- Cannot count comments logically. Usually breaks down big comment blocks on multiple small ones and counts them as separate blocks of comments.
- Autofix function is on by default, which might cause unexpected behavior of analyzed project in after the first run of EclipsePro Audit.
- Does not provide data flow analysis
- There is some overlap between violations. Tool do not have option to group output by code, that is show all violations per line of code or block of code.
- Not very often, but it makes a mistakes. Usually in case when you need to predict usage of entity, like "Variable should be declared as final" audit rule.
- It also does not do a very good job in recognizing the context of using classes or their instances. Sometimes it gives some weird recommendations to replace abstract classes with interfaces just because it does not have methods or variables. It does not take in account that some code is generated (by technologies like GMF).

Final Conclusions

While this does seem like a useful tool for encouraging teams to use a uniform coding style, it seems less useful for actually identifying errors, and rarely more-so than other forms of testing. This, along with the price tag, makes its applicability towards smaller projects, especially those with only one coder, questionable at best.

In mid-size projects, the tool was useful to identify some possible errors like "null pointer" or "code style issues", however you must very carefully applying it for a complex projects with comprehensive object-oriented concepts applied since the tool seems to make more mistakes in those settings.

In large-size projects, the tool is extremely useful to enforce good practices across the team and the chances of finding bugs increases with a larger codebase. It is also very good at finding problems with attributes such as performance, security and extendibility, as long as the defects are typical and simple (which in large projects are abundant).

Overall the tool seems to have a nice balance between flexibility and usability. The reporting capabilities still have room for improvement, but as a "developer's tool" it is sufficient. The price tag is a little expensive for individual projects, especially given the offer of free tools available which do not cover as much functionality but provide many of the basic features. The tool also does not cover other aspects of quality and testing such as test managing, pre/post condition assertions, dynamic and performance analysis; although this last one is understandable since other tools inside the CodePro toolset provide support for these.

This tool would make a nice complement to other techniques such as inspections, but it is doubtful it could replace them. In fact, the tool is advertised as something that will make inspections more efficient by providing style compliance and basic analysis so that the inspection can focus on the more important issues.

Assignment Efforts

Test 1: 4 hours
Test 2: 8 hours
Test 3: 8 hours
Report writing/consolidation: 8 hours
Presentation preparation: 5 hours
Total: 33 hours

Additional time was spent by all members in identifying the tool, installing it and becoming familiarized with it. This was not tracked.