Leading the Evolution™

DevPartner Java Edition Code Review

User’s Guide

Release 4.5

20090928132917

Table of Contents

Preface

This manual provides a printable version of the information in the online help It is intended for new Code Review users and for users of previous versions who want an overview of the product's features and functions.

Users of previous versions of Code Review should read the DevPartner Java Edition Release Notes to learn how this version differs from previous versions.

This manual assumes that you are familiar with the Windows or UNIX operating environments and with Java software development concepts.

What This Manual Covers

This manual contains the following chapters and appendixes:

- Chapter 1, "Using Code Review Java Edition" describes how to use Code Review to analyze program design and code.
- Chapter 2, "User Interface" describes the windows and dialog boxes in Code Review.
- Chapter 3, "Tutorials" demonstrates how to build a code model, use the Design Validation feature, and use the Code Validation feature.
- The Glossary defines technical terms used in the Code Review interface and documentation.

Conventions Used In This Manual

This book uses the following conventions to present information.

- Interactive features of Code Review appear in **bold typeface**. For example:

  To update the information displaying in the **Application Testing** tab of the Start page, click **Refresh**.

- Computer commands appear in `monospace typeface`. For example:

  Execute the `nmjava` command.

- File names and paths appear in boldfaced monospace typeface. For example:

  The session file is saved in the **`\var\sessionfiles`** folder of your product folder.

- Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in ***`italic monospace type`***. For example:

Enter **`http://`*`servername`*`/cgi-win/itemview.dll`**, where ***`servername`*** is the designation of your server.

Getting Help

If ever you have any problems or you would like additional technical information or advice, there are several sources. In some countries, product support from Micro Focus may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described below. If you obtained it from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us as described below.

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Product Support can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name, release (version), and build number of the product.
- Installation information, including installed options, whether the product uses local or network databases, whether it is installed in the default folders, whether it is a standalone or network installation, and whether it is a client or server installation.
- Environment information, such as the operating system and release on which the product is installed, memory, hardware/network specifications, and the names and releases of other applications that were running.
- The location of the problem in the product software, and the actions taken before the problem occurred.
- The exact product error message, if any.
- The exact application, licensing, or operating system error messages, if any.
- Your Micro Focus client, office, or site number, if available.

Contact

Our Web site gives up-to-date details of contact numbers and addresses. The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus Product Support customer, please see your Product Support Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

Additional Help

For more information about Code Review, visit the Micro Focus DevPartner Java Edition product page at http://www.microfocus.com/products/DevPartner/JavaEdition.asp and click the Code Validation link

Chapter 1

Using Code Review Java Edition

You can use Code Review to:

- Create a code model — Create a new code model based on your Java source code or byte-code.
- View rule violations and view design violations — Once a model is available, it is automatically analyzed for compliance with design principles and coding rules. You can view and edit it using a number of views.
- Synchronize source code with the code model — Keep the source code and code model synchronized by updating the model from code edits and refactoring the source code with model edits.
- Create a design reference — Create a package structure design as a design reference.
- Compare a code model with a design reference — Verify code models against the design reference, allowing implementations to be checked against designs.
- Add UML diagrams to Javadoc — This function is available from the command line only; it is demonstrated by the tutorial “Enhancing Javadoc with UML Diagrams” on page 87.
- Publish Maintainability and Compliance metrics to Optimal Delivery Manager. For more information, see “Metrics Publishing Utility” on page 41. Also refer to the separate documentation for PubMetrics.

Code Review functions are available through the interface window and through the command line. For information about the interface, see the topics in the User Interface section of the table of contents. For information about using Code Review from the command line, see “Command Line Interface” on page 35.

Using this Guide

This guide describes:

- The features that analyze and enhance your source code
- The user interface
- Code validation rules
- Design validation principles and concepts

The tutorials demonstrate how to build a code model, use UML diagrams to improve a program's package structure, apply coding rules to correct errors and improve performance, and enhance Javadoc by including UML diagrams.

Creating a Code Model

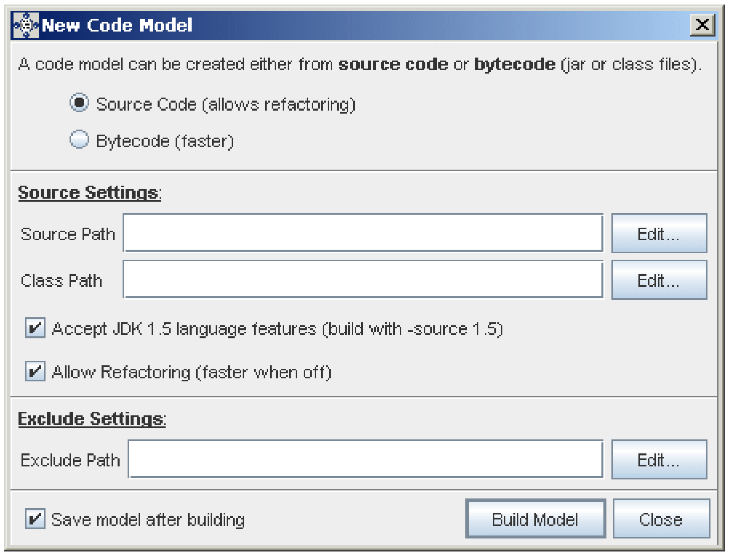
A *code model* is used to analyze the rules and package structure of Java code. Code Review creates a code model by parsing and analyzing Java code from either source code or bytecode. After the code model is built, you can view the results of the rules and structure analysis in the Code Validation Summary page and the Design Validation Summary page. You can save the code model to file; code model files have the extension `.psm`.

Code Review incorporates features that allow the code model to be modified. Changes to a code model created from Java source code can be applied to the code (for details, see "Synchronizing the Source Code with the Code Model" on page 32). Although a code model built from bytecode can be used solely for analysis, it allows assessment of third-party libraries. Because it takes less time to build than a source code model, a model created from bytecode is also convenient for analyzing large applications.

Building a code model may take considerable time depending on the size of your application. An alternative to reduce the build time for either a source code or a bytecode model is to set the **Exclude Path**. This allows the analysis to focus on specific components rather than an entire application. Setting the exclude paths can also be used to resolve issues arising from duplicate class files.

To create a new code model:

1 Choose **File>New Code Model**. The **New Code Model** dialog box appears.


New Code Model
A code model can be created either from **source code** or **bytecode** (jar or class files).
[● Source Code (allows refactoring)] ○ Bytecode (faster)
Source Settings:
Source Path
Edit...
Class Path
Edit...
[✓] Accept JDK 1.5 language features (build with -source 1.5)
[✓] Allow Refactoring (faster when off)
Exclude Settings:
Exclude Path
Edit...
[✓] Save model after building
Build Model
Close

2 Set the input source type. Select one of the following options:

◇ **Source Code (allows refactoring)** — Allows refactoring.

◇ **Bytecode (faster)** — Builds more quickly.

The rest of the settings vary depending on the input.

1 To set the paths, type the path in the field; or browse and select the location:

a Click **Edit** to display the **Edit Source Path** dialog box.

b Click **Add** and browse to the appropriate location.

2 If you select **Source Code (allows refactoring)**, set the **Source Settings** as required:

◇ **Source Path** — List of directories and files where the source files can be found. If all the sources are in one source tree, enter a path consisting of the root of this tree.

◇ **Class Path** — Java classpath used to build the model. This should be the same as the classpath that is used when the sources are compiled.

Note: It is often possible to load sources without a complete classpath, but the model is then incomplete because not all references can be resolved. In some cases, the parser may not be able to build the model if the classpath is not defined.

◇ (JDK 1.4 only) **Enable 'assert' keyword (build with -source 1.4)** — Enables the new syntax introduced in JDK 1.4 (most notably the `assert` keyword).

◇ (JDK 1.5 only) **Accept JDK 1.5 language features (build with -source 1.5)** — Enables the new syntax introduced in JDK 1.5.

◇ **Allow Refactoring (faster when off)** — Enables refactoring of the source files from the code model. This option requires additional work when analyzing, so it is much faster (typically twice as fast) to build a code model with this option cleared. However, you **must** select this option if you want to synchronize the source code with code model edits.

3 If you select **Bytecode (faster)**, set the **Bytecode Settings:** as required:

◇ **Bytecode Path** — List of directories and files where the compiled bytecode files can be found. If all the class files are in one source tree, give a path consisting of the root of this tree.

◇ **Source Path** — Optional list of directories and files where the source files can be found. If all the sources are in one source tree, give a path consisting of the root of this tree.

Set this field if you want to view the source code in either a **Source Fragment** pane or an external editor.

4 To define the **Exclude Settings**, type the path directly in the **Exclude Path** field. Alternatively, you can browse and select the location:

◇ Click **Edit** to display the **Edit Exclude Path** dialog box.

◇ Click **Add** and browse to the appropriate location.

Note: The **Exclude Settings** option does not apply to PMD/custom rules.

5 Select the **Save** option if you want to save the code model automatically to the working folder upon completion of the build.

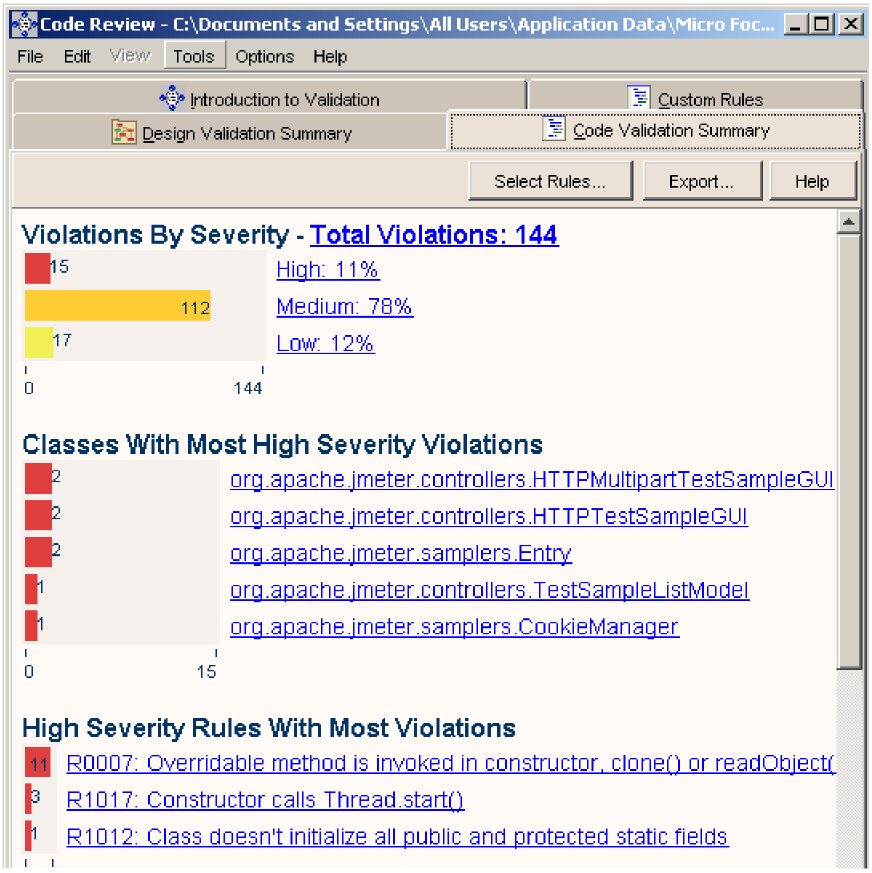
Note: The file name is generated automatically and will overwrite an existing file of the same name without warning. The file name is derived from the highest level package with at least four child packages. If this criterion is not satisfied, the file name will default to **`model.psm`**.

6 Click **Build Model** to create the code model. A **Build Status** section is added to the dialog box. If compile errors are encountered, they are displayed in a table and the dialog box remains open. To open the source in the external editor at the line of the error, double-click a table row.

Code Validation

When you create a code model based on Java source code or bytecode, Code Review analyzes it automatically, using Code Review, PMD, and custom coding rules, and displays a summary of the results in the Code Validation Summary page.

Figure 1-1. Code Validation Summary Page


Code Review - C:\Documents and Settings\All Users\Application Data\Micro Foc...
File
Edit
View
Tools
Options
Help
Introduction to Validation
Custom Rules
Design Validation Summary
Code Validation Summary
Select Rules...
Export...
Help

Violations By Severity - Total Violations: 144

15 High: 11%
112 Medium: 78%
17 Low: 12%
0 144

Classes With Most High Severity Violations

2 org.apache.jmeter.controllers.HTTPMultipartTestSampleGUI
2 org.apache.jmeter.controllers.HTTPTestSampleGUI
2 org.apache.jmeter.samplers.Entry
1 org.apache.jmeter.controllers.TestSampleListModel
1 org.apache.jmeter.samplers.CookieManager
0 15

High Severity Rules With Most Violations

11 R0007: Overridable method is invoked in constructor, clone() or readObject(
3 R1017: Constructor calls Thread.start()
1 R1012: Class doesn't initialize all public and protected static fields

You can view details of the code violations by clicking on the pertinent link to display the Code Validation Details page. The information in that page can help you correct the source code. To be able to decide which changes you should make in the source code, you need to understand the rules and sets used in the analysis.

Coding Rules

The rules that Code Review uses to analyze your program reflect best Java coding practices.

When you build a code model, the profile includes all built-in coding rules and the rules selected in the Custom Rules page. You can filter the results list by using the **Select Rules** option in the Code Validation Summary page. Information about each violated rule is displayed when you select the rule in the **Rule Filter** dialog box or in the Code Validation Details page.

Note: To specify which PMD rulesets or custom rules to include in the profile, change the selection in the Custom Rules page and rebuild the code model. By default, the Basic and Design rulesets are selected.

Severity Level

Rule violations are prioritized by severity:

- High — The violation is likely to cause a run-time error, performance problems, or design issue.
- Medium — The violation may cause a problem, but there are legitimate circumstances when the rule can be ignored.
- Low — The violation may affect the maintainability of the application, reflect a common practice or be the result of a rule that can cause many false positive violations.

Rule Categories

The coding rules are grouped by categories or rulesets that identify the types of errors the rules detect. For example:

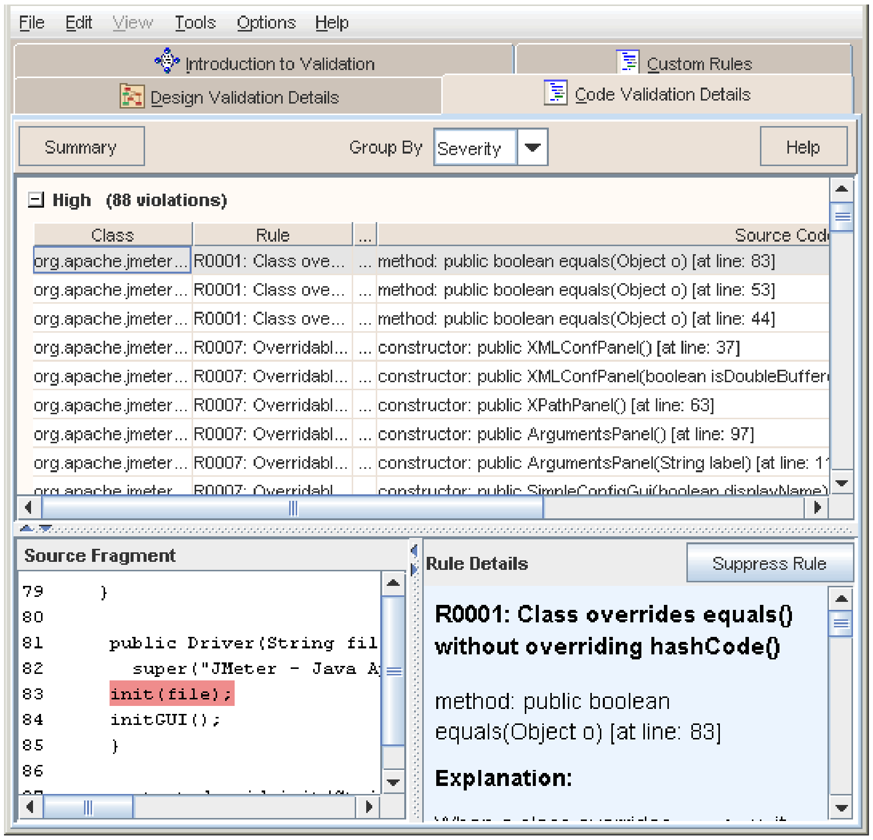
- Correctness — The rule violation is likely to cause a bug. The code probably does not do what you expect. You should always examine code that violates these rules, whether the code is new or legacy.
- Design — The rule helps improve maintainability and object oriented programming (OOP) design. Code that violates these rules could cause bugs in the future when it is modified or classes are extended. Design rules are more appropriate for new code than for existing legacy code.
- Performance — The rule violation is likely to cause problems with speed, memory or resources.

A complete list of the coding rules built into Code Review, with descriptions, is available in JRulesReference.html, in theDevPartner Java Edition product installation folder. For explanations of the rulesets listed in the Custom Rules page, see the PMD Web site, http://pmd.sourceforge.net/.

Viewing Rule Violations

The Code Validation Details page provides the means to view and edit the source code. This page displays individual code violations and explanations. To display the page, click any link in the Code Validation Summary page.

Figure 1-2. Code Validation Details Page



High (88 violations)

Class	Rule	...	Source Code
org.apache.jmeter...	R0001: Class ove...	...	method: public boolean equals(Object o) [at line: 83]
org.apache.jmeter...	R0001: Class ove...	...	method: public boolean equals(Object o) [at line: 53]
org.apache.jmeter...	R0001: Class ove...	...	method: public boolean equals(Object o) [at line: 44]
org.apache.jmeter...	R0007: Overridabl...	...	constructor: public XMLConfPanel() [at line: 37]
org.apache.jmeter...	R0007: Overridabl...	...	constructor: public XMLConfPanel(boolean isDoubleBuffer
org.apache.jmeter...	R0007: Overridabl...	...	constructor: public XPathPanel() [at line: 63]
org.apache.jmeter...	R0007: Overridabl...	...	constructor: public ArgumentsPanel() [at line: 97]
org.apache.jmeter...	R0007: Overridabl...	...	constructor: public ArgumentsPanel(String label) [at line: 1
org.apache.jmeter...	R0007: Overridabl...	...	constructor: public SimpleConfigGui(boolean displayName)

```
79      }
80
81        public Driver(String fil
82          super("JMeter - Java A
83          init(file);
84          initGUI();
85        }
86
```

R0001: Class overrides equals() without overriding hashCode()

method: public boolean equals(Object o) [at line: 83]

Explanation:

To correct code violations, it is first advisable to examine the summary to determine what remedial action is required and then edit the code accordingly:

1 Examine the Code Validation Summary page to determine the worst cases of code violations from the three categories.

2 If necessary, click **Select Rules** to open the **Rule Filter** dialog box, where the rules displayed in the results can be selected. Once defined, this set of rules will persist through sessions until changed. When the settings are satisfactory, click **OK** to return to the Code Validation Summary page.

3 To view details of specific violation groupings, click the link of interest to open the Code Validation Details page.

4 In the Code Validation Details page, select one of the following options from the **Group By** list to change the grouping method and display a list of the items in the selected group.

- ◇ **Class** — Java class or interface containing the rule violation
- ◇ **Rule** — Rule that is being violated
- ◇ **Category** — Rule category (also called ruleset)

◇ **Severity** — Severity of rule violation

5 To view the table of violations for an item, click the item.

6 To view the source that causes a specific violation, select it in the table. The **Source Fragment** pane highlights the violating code. The **Rule Details** pane provides more information about the rule being violated.

7 To edit the source code to correct the violation, either double-click the item in the table, click the **Source Fragment** link or double-click inside the **Source Fragment** pane to open the external editor.

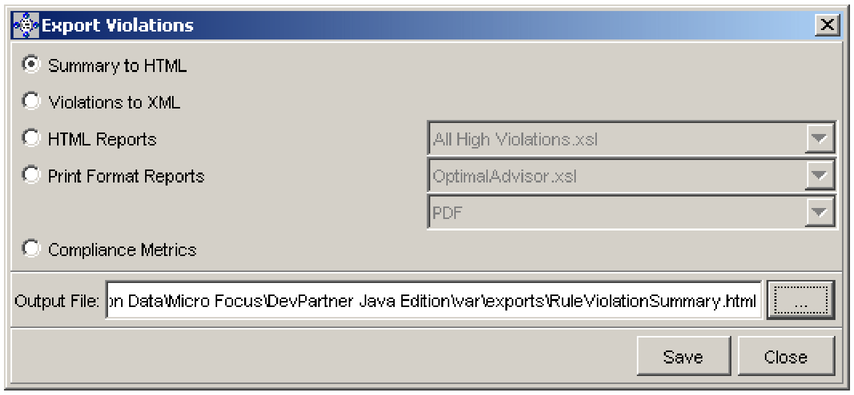
8 To exclude a rule from the results list, click **Suppress Rule** in the **Rule Details** pane.

9 To return to the Code Validation Summary page, click **Summary**.

Exporting Rule Violation Reports

You can export Rule Violation reports in HTML, XML, or print format:

1 In the Code Validation Summary page, click **Export** to display the **Export Violations** dialog box.



Output File: n Data\Micro Focus\DevPartner Java Edition\var\exports\RuleViolationSummary.html

2 Select what you want to export. For details on the export options, see “Export Violations” on page 59 and “Customized Reports” on page 46.

3 Specify the file name and location where you want to save the exported violations report or accept the default.

4 Click **Save** to export the violations.

5 When the export is complete, close the dialog box.

The default location of the exported file depends on your operating system:

- Windows XP or 2003 — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\exports`**

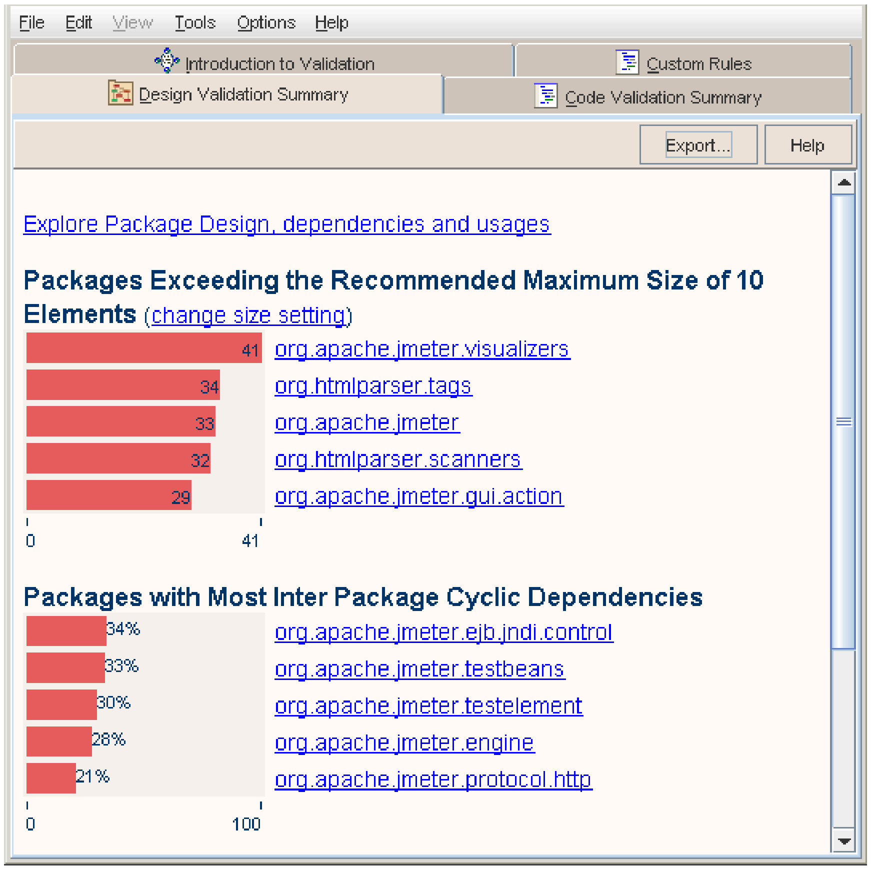
Note: By default, the **`Application Data`** folder is hidden. To display the **`exports`** folder and its contents, type the path in the **Address** field of Windows Explorer and press Enter.

- Windows 2008 or Vista — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\exports`**

- UNIX — ***`DPJ_dir/var/exports`***
  where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder.

Design Validation

When you create a code model based on Java source code or bytecode, Code Review analyzes it and displays a summary of the results in the Design Validation Summary page.



Explore Package Design, dependencies and usages

Packages Exceeding the Recommended Maximum Size of 10 Elements (change size setting)

41	org.apache.jmeter.visualizers
34	org.htmlparser.tags
33	org.apache.jmeter
32	org.htmlparser.scanners
29	org.apache.jmeter.gui.action

0 41

Packages with Most Inter Package Cyclic Dependencies

34%	org.apache.jmeter.ejb.jndi.control
33%	org.apache.jmeter.testbeans
30%	org.apache.jmeter.testelement
28%	org.apache.jmeter.engine
21%	org.apache.jmeter.protocol.http

0 100

Figure 1-3. Design Validation Summary Page

Based on the information and suggestions provided by design validation, you can improve the package design of your source code, whether you are working with legacy source code or developing new code. Code Review can only analyze and make suggestions. As a developer, you need to make intelligent decisions about the changes you actually want to make. To do so, you need to understand the principles of good package design.

Package Design

A good modular structure is essential when building and maintaining applications. The benefits of modular design have long been recognized and include product flexibility, comprehensibility, and reduced development time.[1]

1. D. Parnas, Carnegie-Mellon University, “On the criteria to be used in decomposing systems into modules”. *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053–1058. Online HTML copy from the ACM site at http://www.acm.org/classics/may96/.

In Java code, the modular structure is defined by the package hierarchy. Every Java class is part of a package, and packages are hierarchically structured in a package tree.

Code Review lets you work with design and improve package hierarchies in the following ways:

- Analyze and improve the package hierarchies of existing source code.
- Develop new code that reflects good package design principles.
- Design package hierarchies outside the source code.
- Verify source code against a package design.

Package Definitions

In Java code, the modular structure is defined by the package hierarchy, not the class hierarchy[1]. Every Java class is part of a package and packages are structured hierarchically in a package tree.

Unlike classes or methods, packages in Java are implicitly defined. A class (or more precisely a compilation unit) declares itself to be in a certain package. This declaration is used to determine the packages in an application.

Interfaces and dependencies are two important concepts in modular design. For Java packages, these properties of packages are also defined implicitly:

- The *dependencies* between two packages A and B are defined as the set of all dependencies from types that are part of A to all the types that are part of B. Dependencies between packages are therefore implicitly defined in the type definitions via references to other types.
- The *interface* of a package is defined by the interfaces of the set of public types (classes or interfaces) in the package. Types are part of a package if the compilation units they are in have a package statement referring to the package.

A dependency on a nested package is also a dependency on the parent package. For example, a class that depends on **`java.util`** also depends on **`java`**.

Package Design Principles

Packages are not just namespaces. A good package structure is based on solid design principles.

The goal of package design is a package structure that it is easy to understand, test, maintain and extend. Robert C. Martin distinguishes the following design principles[2]:

- Acyclic Dependency Principle (ADP) — *Allow no cycles in the package-dependency graph.*

1. J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification*, Chapter 7. Online version available at http://java.sun.com/docs/books/jls/second_edition/html/packages.doc.html#60384.

2. Martin, R.C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall 2002.

Packages that adhere to the acyclic dependency principle are typically easier to unit test, maintain and understand. Cyclic dependencies make it more difficult to predict what the effect of changes in a package are to the rest of the system. The ADP is related to the concept of layering: a package that conforms to the ADP has a layered architecture.

- Stable Dependencies Principle (SDP) — *Depend on the direction of stability.*

  Stable packages are packages that are difficult to change. The SDP principle states that packages should depend on packages that are more stable than themselves. Unstable packages that are used a lot by other packages are potential problem areas in a design.

- Reuse-release Equivalence Principle (REP) — *The granule of reuse is the granule of release.*

  This principle is based on the idea that packages, rather than individual classes, are the units of reuse. If a package is to be reused, all classes in that package should be designed for reuse.

- Common Reuse Principle (CRP) — *The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.*

  This principle is also based on the idea that packages, rather than individual classes, are the units of reuse. Classes that are intended to be reused together, should be put in the same package.

- Common Closure Principle (CCP) — *Classes within a released component should share common closure. That is, if one needs to be changed, they probably all need to be changed. What affects one, affects all.*

  Classes that are likely to change together should be put in the same package. Unrelated classes should not be put in the same package.

Designing Package Structures

In the traditional approach to application development, designing the package structure is the first step in the design phase. From the UML point of view, the package design is the highest level of design and the design approach is top-down. This is known as the *waterfall* approach.

Most organizations, however, prefer to use more flexible, iterative approaches. Agile development methods, such as Extreme Programming (XP), advocate spending less time on up-front design. Instead, design improvements are realized through refactoring as you develop the software. Using agile methods, the package design changes along with the class design.

Code Review both supports approaches to software development and enables you to design packages when working with code.

Working with Existing Code

When working with existing code written by somebody else, it can be an enormous task to re-engineer an intended architecture. Software documentation is often not up-to-date or sometimes even nonexistent and key developers may have left the project. The most accurate and complete source of information is the source code. There may, however, be hundreds of files and it is difficult to know where to start if you do not have a clue about the intended architecture.

Code Review gives you immediate insight into how different parts of the program depend on each other. It can reconstruct an intended layering, even if there are many cycles in the dependency structure. You also have a powerful set of metrics that give direct insight into the nature of the software with which you are dealing. The results are often the starting point for a refactoring.

Developing New Code

In agile development methods, designs are largely made while coding. However, when working on a detailed level, it can be very difficult to keep the general architecture in mind. Nevertheless, any small change in a method can break an architecture by introducing an undesirable dependency.

Code Review can help you make design decisions while coding. It makes you more aware of the consequences and the quick feedback on your decisions can be enlightening. Furthermore, it assists in improving the design through refactoring.

Refactoring Existing Code

Refactoring is improving the design of existing software code.

Code Review lets you quickly assess the consequences of moving classes and packages to another location in the package structure. In many cases, an analysis with Code Review leads to the conclusion that a class is in the wrong package. However, moving classes in the code can be a lot of work because all the references to the class need to be modified.

Code Review provides automated refactoring support for:

- Moving and renaming packages and classes.
- Removing unused import statements.

Verifying Designs

In some projects, it may be beneficial to define a package structure outside of the code. Code Review lets you define such a package structure and then verify that an implementation conforms to this design. This is particularly useful when working in large teams.

Recommended Reading

One of the first people to publish on the topic of package design was John Lakos in his book on *Large Scale C++ Software Design*. (7) In C++, classes are usually organized in source directories: the language does not directly support packages. Nevertheless, many of the principles described in this book, including the ADP, are also relevant for Java development. R. Braithwaite-Lee has written an article, “Dependencies and Levelization”, based on the work of John Lakos. (1)

Robert Martin also introduces a set of metrics for assessing object oriented designs. The package design principles in this document have been directly taken from his book that discusses both agile development methods and design patterns and principles. (8) Section 4 in the book is devoted to package design.

Kirk Knoerschild has written *Java Design, Objects, UML and Process*. About the ADP he writes, "When developing Java applications, we should rarely find ourselves in a situation where we have violated the ADP. The consequences of doing so are dire, and we should avoid it at all costs."

Refactoring by Martin Fowler (3) is one of the most important books on object oriented development of recent years. It introduces and describes the process of refactoring in great detail. Fowler has also written an interesting article that deals with package design, layering and reducing coupling. (2)

Many books about Java design and style do not cover package design. A positive exception is *The Elements of Java Style*, which includes the ADP and other package conventions. (11)

An earlier version of Code Review was presented at the IASTED SEA conference. (5)

Three other documents that provide insight into different aspects of design validation include an article on UML by Hubert Matthews and Mark Collins-Cope in *ObjectiveView* (9); the chapter on Packages in *The Java Language Specification* (4); and an article by D.Parnas from *Communications of the ACM* (10).

Bibliography

1 Braithwaite-Lee, R. "Dependencies and Levelization".

2 Fowler, M. "Reducing Coupling". *IEEE Software* July-August 2001, pp. 102-105. Online PDF copy available from the Web site of Martin Fowler at http://martinfowler.com/ieeeSoftware/coupling.pdf.

3 Fowler, M. *Refactoring*, Addison-Wesley, 1999.

4 Gosling, J., B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*, Chapter 7: "Packages". Online version available at http://java.sun.com/docs/books/jls/second_edition/html/packages.doc.html#60384.

5 Hautus, E. "Improving Java Software through Package Structure Analysis". Sixth IASTED International Conference Software Engineering and Applications, 2002. Online PDF copy available at http://www.xs4all.nl/%7Eehautus/papers/PASTA.pdf.

6 Knoernschild, K. *Java Design, Objects, UML and Process*, Addison-Wesley, 2002.

7 Lakos, J. *Large Scale C++ Software Design*, Addison-Wesley, 1996.

8 Martin, R. C. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

9 Matthews, H. and M. Collins-Cope, "The Topsy Turvy World of UML". ObjectiveView # 4. Online PDF copy available at http://www.ratio.co.uk/objectiveview.html#issue4.

10 Parnas, D. "On the criteria to be used in decomposing systems into modules". *Communications of the ACM*, Vol. 15, No. 12, 1972, pp. 1053–1058. Online HTML copy from the ACM Web site at http://www.acm.org/classics/may96/.

11 Vermeulen, Ambler, Bumgardner, Metz, Misfeldt, Shur, & Thompson. *The Elements of Java Style*, Cambridge University Press, 2000.

Layering

A common system architecture approach is to define a layered architecture. A layered architecture is easier to understand and maintain than one where all packages depend on each other.

Strict Versus Non-strict Layering

Szyperski makes a distinction between strict and non-strict layering.[1] In a strict layering, elements depend only on elements from the layer directly below it. The lower layers are hidden. In a non-strict layering, layers can depend on any lower layers. In Java programming, strict layering is not common.

Reconstructing a Layering

In a package structure that conforms to the ADP, a non-strict layering can be reconstructed using the following definition:

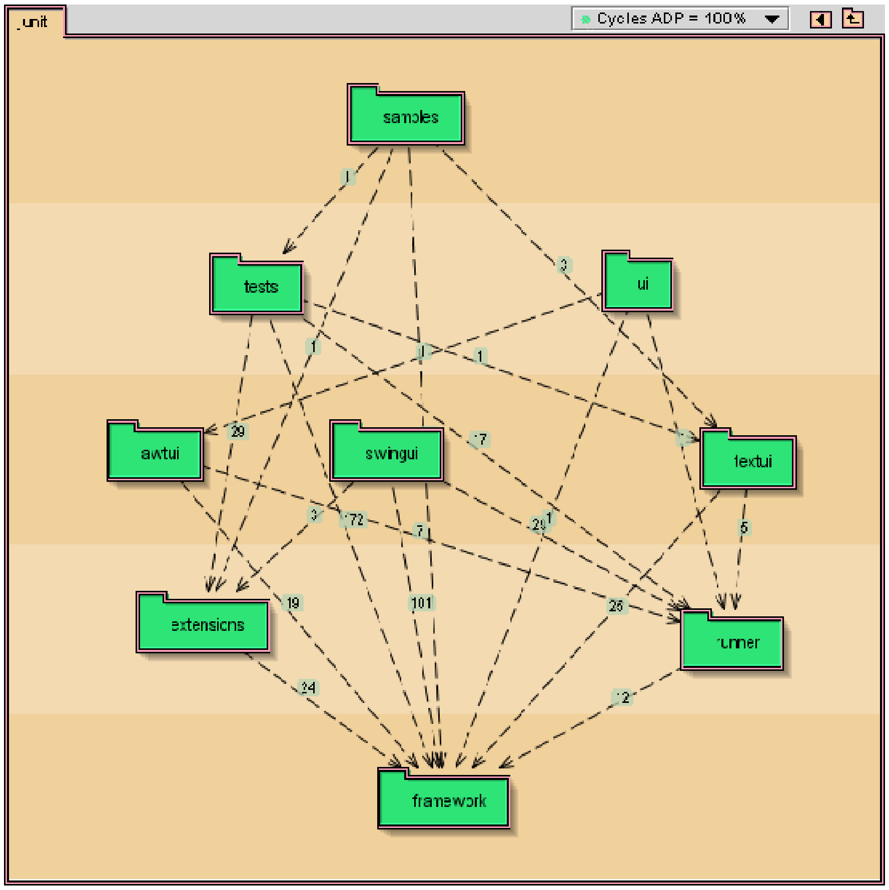
> The package *layer* is the maximum length of a dependency path to a package with no dependencies.

Note that the reconstructed layering has the added property that packages on one layer do not depend on each other.

1. Szyperski, C. *Component Software*, Addison-Wesley, 1998.

The following graphic shows a layout generated with this layering definition.

Figure 1-4. Non-Strict Layer Reconstruction

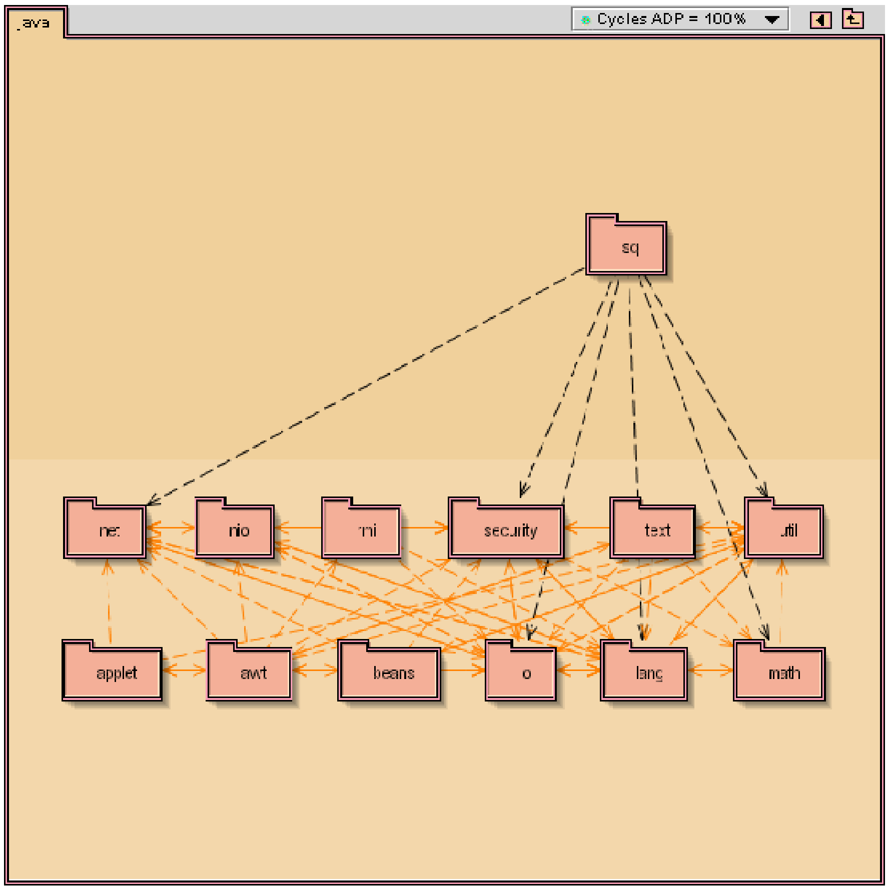


Dealing with Cycles

Code Review allows package structures to be visualized as UML class diagrams. Unlike the JUnit example in the preceding figure, most Java programs do not conform to the ADP. In such cases, it can be difficult to comprehend the intended package structure.

A simple way to deal with cyclic dependencies is to simply ignore dependencies that are part of a cycle. The result of this approach is shown in the following graphic, using the Java package as an example. Dependencies that are part of a cycle are shown in orange.

Figure 1-5. UML Class Diagram — Package Structure



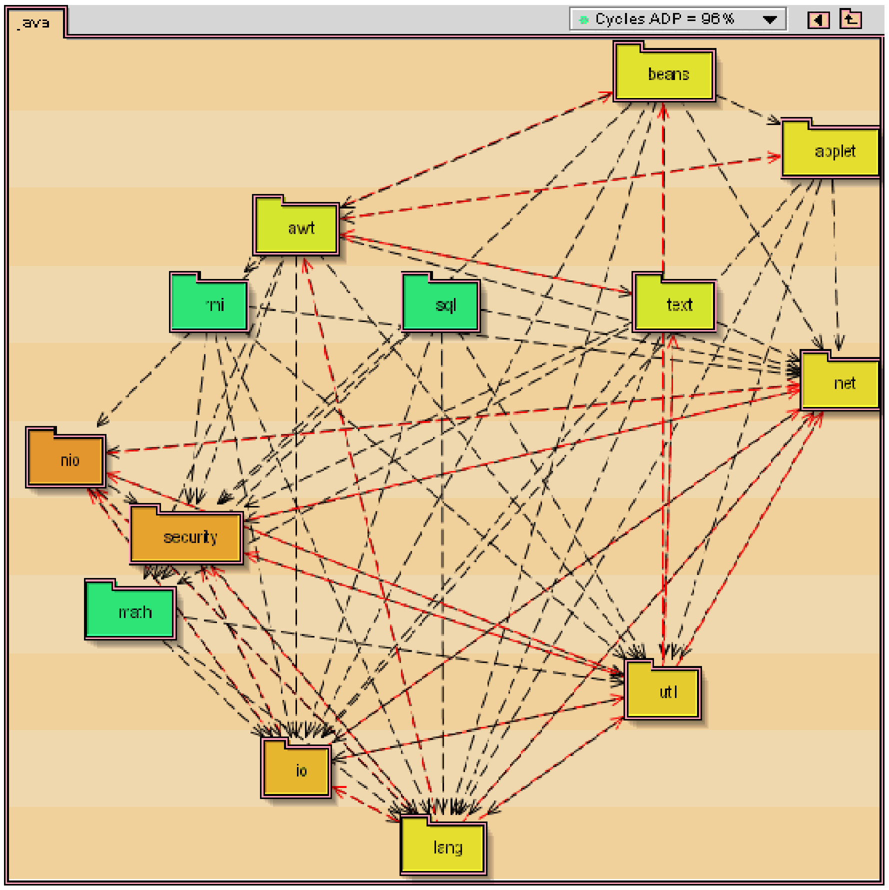
Since the approach in the preceding graphic does not help to understand the intended layering, Code Review introduces an algorithm that reconstructs a layering in a more advanced way.

The algorithm finds sets of dependencies such that, if they were to be removed, the resulting package structure would be acyclic. For all possible sets, a set is chosen that has a minimal total weight.

The *weight* of a dependency is the number of references from one package to another. The weight is an indicator for the amount of work that is necessary to remove this dependency.

The smart layering algorithm finds an estimate for the amount of work that would be necessary to make a package structure acyclic. The dependencies that have been selected for removal are called the *undesirable dependencies*.

Figure 1-6. Smart Layering Algorithm Showing Undesirable Dependencies



Layers at the Class Level

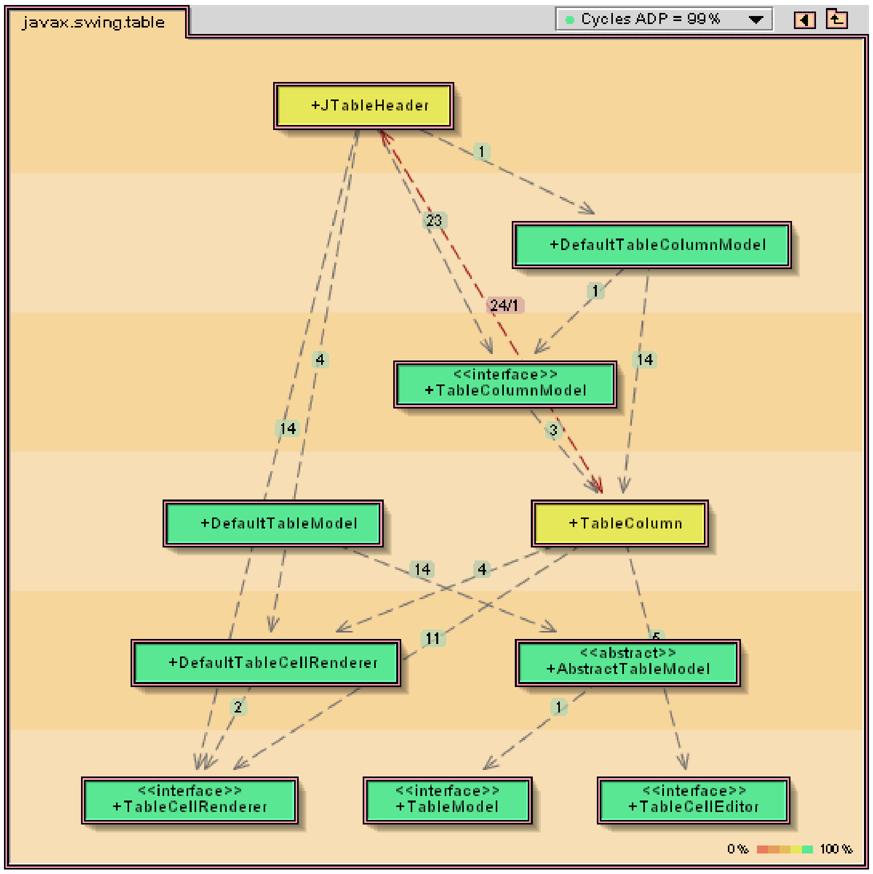
These algorithms can also be used to analyze dependencies between classes. Cycles are more acceptable between classes, because classes from the same package are usually meant to be reused together anyway (see the CRP). Nevertheless, removing cycles between classes can make a package design more easy to understand and test.

The following graphic shows the package **`javax.swing.table`**.

The algorithm suggests to remove the dependency between **`TableColumn`** and **`JTable-Header`**.

This dependency not only ties together **`TableColumn`** and **`JTableHeader`**, but all classes in the top three layers. Removing the dependency (for example with an inner interface in **`TableColumn`**), would make the design acyclic. **`TableColumn`** could then be tested or reused independently of **`JTableHeader`**, **`DefaultTableColumn`**, and **`TableColumn-Model`**.

Figure 1-7. javax.swing.table Package


javax.swing.table
Cycles ADP = 99 %
+JTableHeader
+DefaultTableColumnModel
<<interface>> +TableColumnModel
+DefaultTableModel
+TableColumn
+DefaultTableCellRenderer
<<abstract>> +AbstractTableModel
<<interface>> +TableCellRenderer
<<interface>> +TableModel
<<interface>> +TableCellEditor
1
23
24/1
1
4
14
14
3
14
4
11
2
1
0 %
100 %

Is UML Upside Down?

One thing to note about Code Review UML diagrams is that they are upside down with respect to the convention that parent classes should be shown higher than subclasses. They do, however, follow the convention that higher layers are shown on top off lower layers. H. Matthews and M. Collins-Cope have written an article about this mismatch of conventions[1].

Design Metrics

Metrics can help you create software that is based on package design principles and layering by indicating where design rules are possibly violated. Code Review provides several kinds of metrics:

- Basic metrics — Numbers of specific language elements that occur in a program
- Quality metrics — Measurements based on design principles and general software engineering principles
- Stability metrics — Measurements based on package stability

The Metrics window displays the statistics for each metric. Each tab in the window is associated with a diagram view in the Design Validation Details page; when you select a tab, the diagram automatically changes to the associated view.

1. Matthews, H., and M. Collins-Cope. “The Topsy Turvy World of UML”. *ObjectiveView* #4. PDF version available online at http://www.ratio.co.uk/ov4.pdf.

Basic Metrics

The basic metrics measure the number of specific language elements that occur in a program.

- Nde — Number of direct elements in a package (subpackages + classes)
- Nt — Number of types (classes + interfaces)
- Nc — Number of classes
- Na — Number of abstract types (interfaces + abstract classes)
- Ni — Number of interfaces
- Nf — Number of compilation units (files)

The basic metrics include subpackages when present.

Quality Metrics

The quality metrics are based on the design principles (see "Package Design Principles" on page 15) and on general software engineering principles.

Metrics should always be seen as indicators, not as absolute truth. It is possible to score well on all metrics, but still have an unsatisfactory design.

Acyclic Dependency Principle Metric (ADP/ADPR): The ADP states that there should be no cycles in the package-dependency graph. The ADP metric gives the percentage of dependencies that do not need to be removed in order to get cycles out of the dependency graph. If the metric is 100%, it means there are no cycles in the package structure. If the metric is 90%, it means 10% of the dependencies need to be fixed minimally to get the package structure to conform to the ADP.

The ADP analyzes the dependency graph on a single level, in other words, it does not consider how subpackages are structured. The ADP recursive (ADPR) is a variation that analyzes dependencies in the complete package tree. These metrics are introduced by Code Review. They have proven to be very valuable in assessing the software quality of large projects.

Dependency Inversion Principle Metric (DIP): The Dependency Inversion Principle is another principle from ***Agile Software Development: Principles, Patterns, and Practices (2002)***. It is a class-level design principle, and does not apply to package design. It states that classes should depend upon abstract entities, not concrete ones.

The DIP metric is defined as the percentage of dependencies in a package or class that has an interface or an abstract class as a target. A DIP of 100% indicates that all dependencies in a package diagram are based on interfaces or abstract classes.

Stable Abstractions Principle Metric (SAP): The SAP states that a package should be as abstract as it is stable. The SAP metric in Code Review is the same as Robert Martin's D stability metric (*Agile Software Development: Principles, Patterns, and Practices*, 2002), expressed as a percentage. Abstract packages that are used a lot, as well as concrete packages that themselves are not used, score the highest.

Encapsulation Principle Metric (EP): In a good modular design, each of the modules encapsulates implementation details that are not visible to the user of the module.

The Encapsulation Principle is, therefore: *A substantial part of a package should not be used outside of the package.*

Limited Size Principle Metric (LSP): To avoid solutions where all elements are placed in a single package, we introduce the Limited Size Principle: *All packages should have a limited amount of direct children.*

The LSP metric defines the limit (rather arbitrarily) at 10. Packages with more than 10 subpackages or top-level classes should be split up in smaller packages.

Stability Metrics

The stability metrics have been defined by Robert C. Martin in *Agile Software Development: Principles, Patterns, and Practices, 2002*. They can be seen as intermediate results for the SAP metric.

- Ca — Afferent couplings: the number of incoming classes
- Ce — Afferent couplings: the number of outgoing classes
- I — Instability: **`I = Ce/(Ca+Ce)`**
- Nc — Number of classes in the package
- Na — Number of abstract types (interfaces + abstract classes) in the package
- A — Abstractness: **`A = Na/Nc`**
- D' — Normalized distance: **`D' = |A+I-1|`**

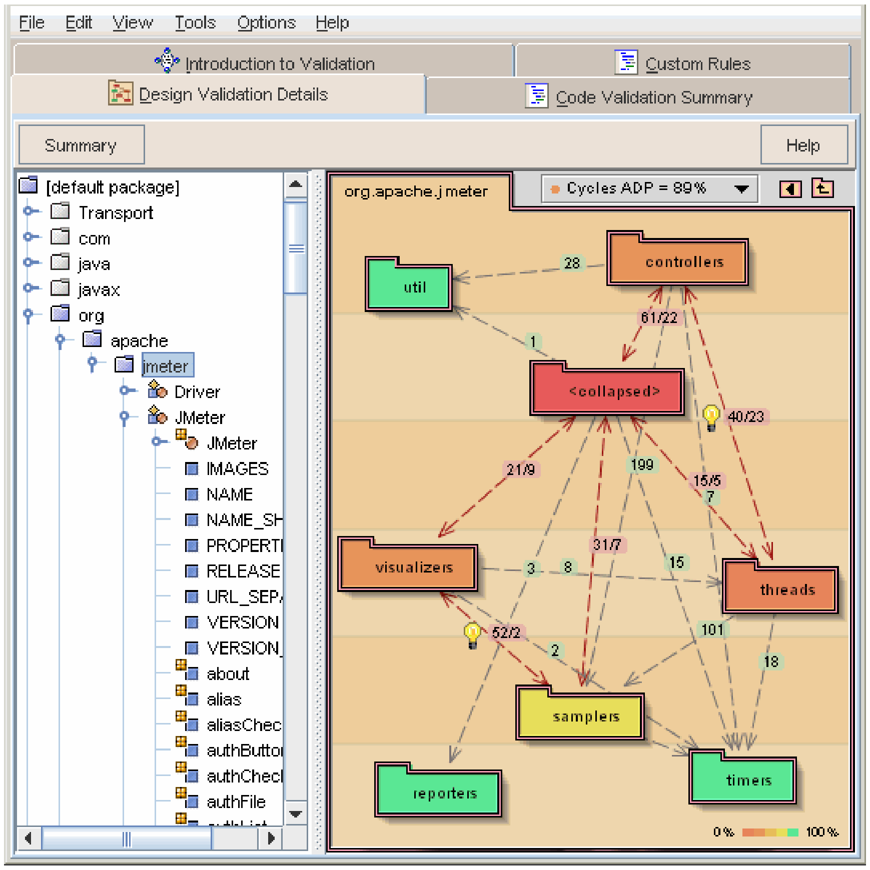
Code Review offers the possibility to use either the original definitions for the couplings Ca and Ce or weighted definitions. In the latter case, the original definitions are modified; the couplings Ca and Ce are not calculated according to the number of incoming and outgoing classes but rather on the weight of the dependencies. Using the dependency weight takes into account that some classes are more tightly coupled than others. The meaning of the metric is not altered.

The stability metrics include subpackages when present.

Viewing Design Violations

While the Design Validation Summary page displays an overview of design violations, the Design Validation Details page provides the means to view and edit the structure of the code model. The Design Validation Details page displays package design details in separate tree and class diagram panes. To display the design validation details, click any link in the Design Validation Summary page.

Figure 1-8. Design Validation Details Page — Code Model Structure



To fix design violations, it is first advisable to examine the summary to determine what remedial action is required, and then refactor the code accordingly:

1. Examine the Design Validation Summary page to establish the worst cases of design violation from the three categories. If necessary, click **change size setting** to open the **Recommended Package Size Setting** dialog box, in which you can specify the maximum number of elements to list.

2. To view details of a specific violation, click the corresponding link to open the Design Violation Details page.

3. In the Design Validation Details page, right-click on objects in the tree or class diagram panes to display a popup menu. The options available depend on the object selected.

4. To display information about dependencies, double-click a dependency to bring up the Dependency View; or double-click a package, class, or interface to display the Usage View.

5. To view the analysis of the model using another quality metric, click the quality metric to display a menu, and select another metric.

6. To view metrics, choose **View>Metrics View**.

7. To improve the layout of a class diagram, drag and drop elements in the diagram. Changes to the layout can be saved as part of the model but this does not affect the source code. If the contents of a package change, a new layout is computed.

8 To change the font size, right-click the background, select **Font Size** and select the font size you want to use. This may change the layout of the packages.

Note: **Note:** Reducing the font size is useful if there are so many elements on the class diagram pane that package and class elements are overlapping.

9 To open the class diagram for a package, double-click the package.

10 To navigate to the parent package of the current class diagram, double-click the **Show Parent** icon.

11 To move an element, drag and drop it to a new location in the tree pane. You can also drop it on the class diagram, as long as the target element is represented in the class diagram pane.

12 To edit a class or interface, double-click it to open the external editor for that class. For more information, see "Configuring an External Editor" on page 34.

13 To undo or redo edits, choose **Edit>Undo** or **Edit>Redo**.

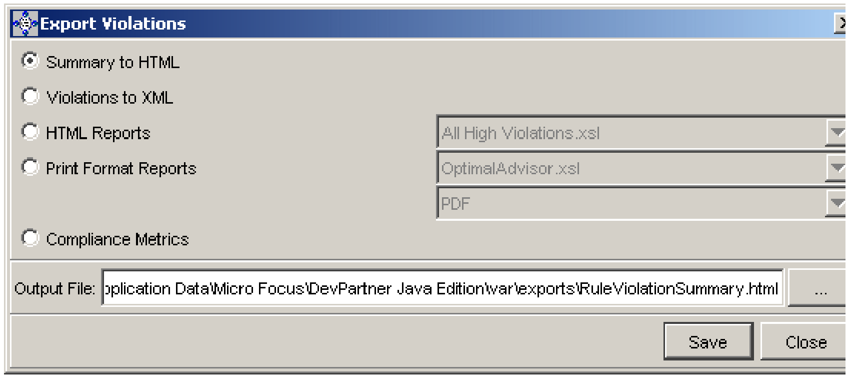
14 To return to the Design Validation Summary page, click **Summary**.

Note: When you edit a code model, the change is stored in the model and can be applied to the code by choosing **Tools>Refactor Code With Model Edits**.

Exporting Design Reports

You can export design metrics in a variety of formats:

1 In the Design Validation Summary page, click **Export** to display the **Export Metrics** dialog box.


Export Violations

(●) Summary to HTML
( ) Violations to XML
( ) HTML Reports — All High Violations.xsl
( ) Print Format Reports — OptimalAdvisor.xsl, PDF
( ) Compliance Metrics

Output File: pplication Data\Micro Focus\DevPartner Java Edition\var\exports\RuleViolationSummary.html

2 Select the desired report format. The formats are described in "Export Options" on page 28.

3 If you selected **Print Format Reports,** select the report format (PDF, postscript, or text) from the third list; PDF is selected by default.

4 A default location and filename are entered automatically in the **Output File** field. To change the location or filename, type over the default or use the browse button to navigate to the desired location.

5 Click **Save** to export the violations summary.

Note: Saving the report may take several seconds.

6 When the export is complete, the dialog box closes.

Default File Location

The default location of the exported file depends on your operating system:

- Windows XP or 2003 — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\exports`**

Note: By default, the **`Application Data`** folder is hidden. To display the **`exports`** folder and its contents, type the path in the **Address** field of Windows Explorer and press Enter.

- Windows 2008 or Vista — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\exports`**
- UNIX — ***`DPJ_dir`*****`/var/exports`**
  where ***`DPJ_dir`*** is the path of the DevPartner Java Edition product folder

Export Options

The **Summary to HTML** option creates a copy of the Design Validation Summary page, without the hyperlinks.

The **Changepoint Metrics** option creates an XML file containing the value of the Maintainability metric. The value of this metric is displayed on the **Cycles** tab of the Metrics window, and is defined in the tab's inline help. For more information, see "Viewing Metrics" on page 28. The Maintainability metric can be published to Optimal Delivery Manager. For more information, see "Metrics Publishing Utility" on page 41. Also refer to the separate documentation for PubMetrics.

The other options create a report, in the specified format, that lists the following metrics for each package:

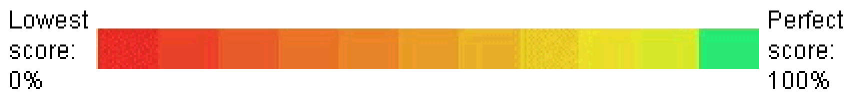
- Members
- ADPR collapsed
- ADPR class level
- Signature Interface Use
- Normalized Distance
- Number of EJBs
- Number of Servlets
- Number of Struts actions
- Number of Struts form beans
- Lines of code
- Lines of manual code
- Lines of generated code

Viewing Metrics

Code Review reports a number of metrics that measure various aspects of the code model design.

The quality metrics are used to color-code elements in the class diagram pane to indicate the degree of conformance with a design principle.

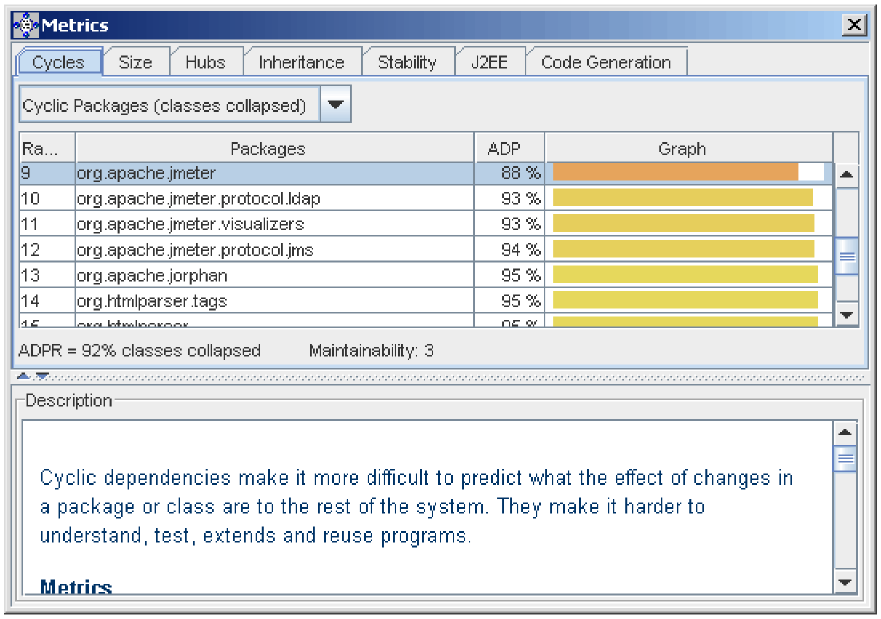
Figure 1-9. Quality Metrics — Conformance



By default, the diagram is color-coded according to the Acyclic Dependencies Principle.

To view the metric values for a code model, choose **View>Metrics**. Choose the appropriate tab in the Metrics window to see the values of interest.

Figure 1-10. Metric Values For Code Model



Ra...	Packages	ADP	Graph
9	org.apache.jmeter	88 %	
10	org.apache.jmeter.protocol.ldap	93 %	
11	org.apache.jmeter.visualizers	93 %	
12	org.apache.jmeter.protocol.jms	94 %	
13	org.apache.jorphan	95 %	
14	org.htmlparser.tags	95 %	

ADPR = 92% classes collapsed   Maintainability: 3

Cyclic dependencies make it more difficult to predict what the effect of changes in a package or class are to the rest of the system. They make it harder to understand, test, extends and reuse programs.

Metrics

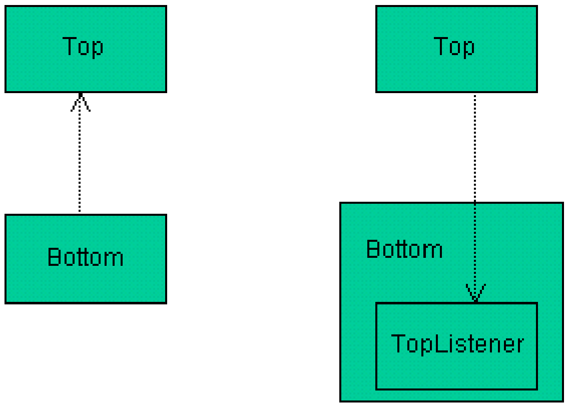
When you open the Metrics window, the tab associated with the current view in the Design Validation Details page is displayed automatically. If you select a different tab, the design view changes to the same metric.

Inline help text in the Metrics window provides definitions for the metrics in each tab. Click the small triangles in the lower right corner to show or hide the help text.

Dependency Inversion Refactoring

Code Review provides support for inversion refactoring between classes, a standard way of removing some cyclic dependencies by inverting the dependencies between two classes. Support is also provided for dependency inversion refactoring between packages when **all** class-to-class dependencies between the two packages can be inverted.

As an example, assume two classes — **`Bottom`** and **`Top`** — in which **`Bottom`** depends on **`Top`**.



To invert the dependencies:

1 Create an inner interface in **`Bottom`**, for example, **`TopListener`**.

2 To the **`TopListener`** interface, add every method from **`Top`** that is called from **`Bottom`**.

3 Replace all occurrences of **`Top`** in **`Bottom`** with **`TopListener`**.

4 Have **`Top`** implement the **`TopListener`** interface. This is possible because **`TopListener`** consists of the methods that were originally in **`Top`**.

5 Remove imports from **`Bottom`** and add them to **`Top`**, if they are in different packages.

The **`Top`** class now depends on **`Bottom`**.

It is important to understand, however, that Code Review can provide support for dependency inversion refactoring between classes only when the following criteria are satisfied:

- **`Bottom`** does not extend or implement **`Top`**.
- **`Bottom`** does not create **`Top`**.
- **`Bottom`** does not use a **`Top`** field.
- **`Bottom`** does not use **`Top`** in its interface.

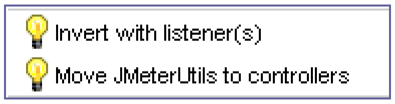
Note: In Code Review, for dependency inversion refactoring between classes and packages, by default the name of the listener is set to **`NameOfClassListener`** and **`NameOfPackageListener`**, respectively.

Fixing Dependencies

The Design Validation Details page shows the dependencies between packages. On the class diagram pane, cyclic dependencies are shown in red if the layering algorithm indicates that a fix is appropriate (for further details, see "Layering" on page 19). Where cyclic dependencies have a light bulb icon beside them, this indicates that Code Review can refactor the code to remove a cycle either by moving a class, removing an unused import or by inverting the dependency. For more information, see "Dependency Inversion Refactoring" on page 29.

To refactor the code:

1 Where a light bulb is shown, right-click it to display a menu of possible actions and choose the action you want to implement. The refactoring actions available in the menu depend upon the number of alternative solutions for correcting the cyclic dependency.

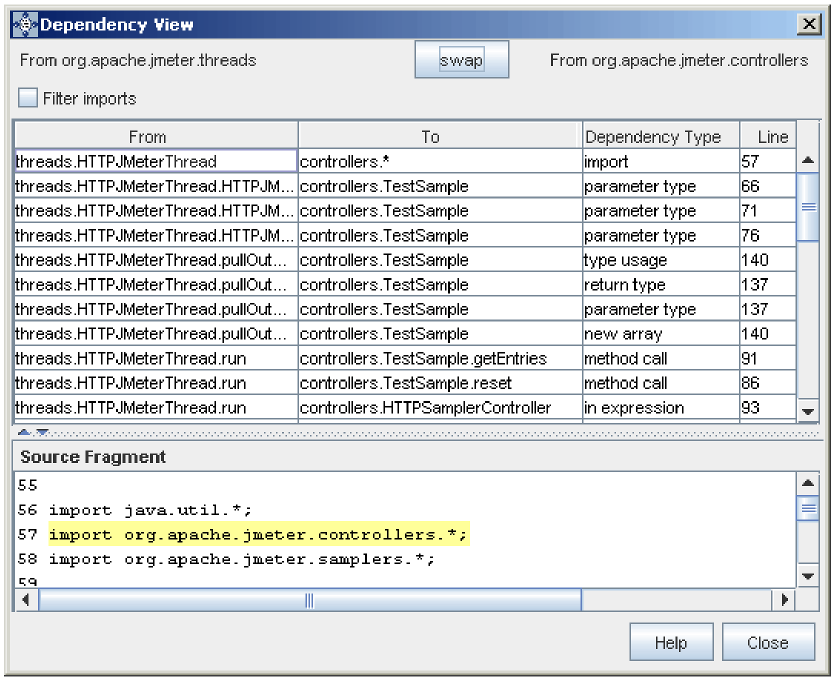


You may need to modify the code manually to eliminate other undesirable dependencies.

2 To display a high-level analysis, showing only cycles between packages (not cycles between individual elements such as classes or methods) choose **View>Classes Collapsed.** This collects all top level classes in a package and represents them by a virtual *collapsed classes* package.

3 If you are satisfied with the changes made in the code model, you apply the changes in the code. For more information, see “Synchronizing the Source Code with the Code Model” on page 32.

4 To display a list of all dependencies, double-click a dependency. Information about the dependencies is displayed in the Dependency View.


Dependency View
From org.apache.jmeter.threads
swap
From org.apache.jmeter.controllers
Filter imports
Source Fragment
Help
Close

From	To	Dependency Type	Line
threads.HTTPJMeterThread	controllers.*	import	57
threads.HTTPJMeterThread.HTTPJM...	controllers.TestSample	parameter type	66
threads.HTTPJMeterThread.HTTPJM...	controllers.TestSample	parameter type	71
threads.HTTPJMeterThread.HTTPJM...	controllers.TestSample	parameter type	76
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	type usage	140
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	return type	137
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	parameter type	137
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	new array	140
threads.HTTPJMeterThread.run	controllers.TestSample.getEntries	method call	91
threads.HTTPJMeterThread.run	controllers.TestSample.reset	method call	86
threads.HTTPJMeterThread.run	controllers.HTTPSamplerController	in expression	93

```
55
56 import java.util.*;
57 import org.apache.jmeter.controllers.*;
58 import org.apache.jmeter.samplers.*;
59
```

You can use this information to determine how to modify the code manually to remove the dependency.

5 To fix a dependency manually, highlight the dependency in the list and click the **Source Fragment** link to open the code in an external editor.

6 After changing the code, choose **Tools>Update Model from Code Edits** to synchronize it with the code model.

Synchronizing the Source Code with the Code Model

Based on Code Review's analysis of your source code, you can edit the model (for example, invert dependencies) or edit the source code itself. In either case, the code model and source code must be synchronized. This topic describes how to synchronize the source code with the code model.

After editing the model, you can update the code from the model by choosing **Tools>Refactor Code With Model Edits.** You can only do this with files that have not been edited outside the control of Code Review since the model was last built.

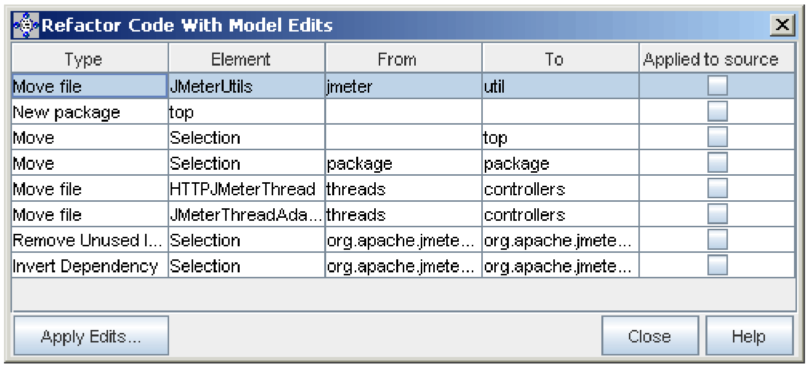
Note: This functionality is not available if you did not select the **Allow refactoring** option when creating the code model. For further details, see "Creating a Code Model" on page 8.

Prior to refactoring the code, it is highly recommended to make a backup of your source files because you cannot choose to undo the changes.

To apply changes in the model to the source code:

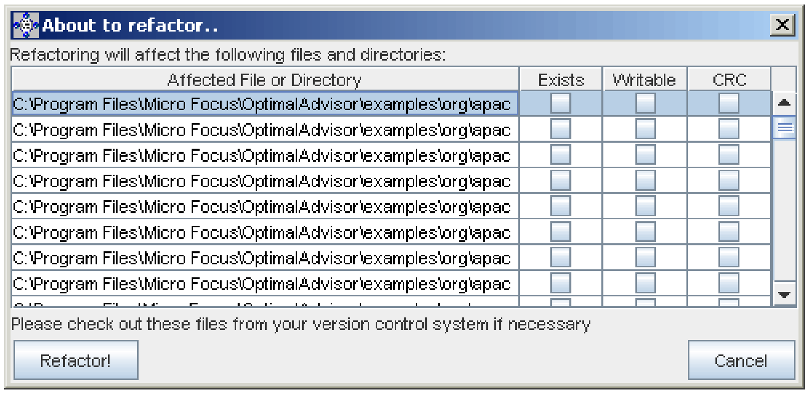
1 From the **File** menu, choose **Save** or **Save As** to save the model.

2 From the **Tools** menu, choose **Refactor Code With Model Edits** to display the dialog box.



Type	Element	From	To	Applied to source
Move file	JMeterUtils	jmeter	util	☐
New package	top			☐
Move	Selection		top	☐
Move	Selection	package	package	☐
Move file	HTTPJMeterThread	threads	controllers	☐
Move file	JMeterThreadAda...	threads	controllers	☐
Remove Unused I...	Selection	org.apache.jmete...	org.apache.jmete...	☐
Invert Dependency	Selection	org.apache.jmete...	org.apache.jmete...	☐

3 Click **Apply Edits** to display the **About to refactor** dialog box. This dialog box shows a list of source code files to be modified. These files need to exist and to be writable. The cyclic redundancy check (CRC) also needs to be correct.



Refactoring will affect the following files and directories:

Affected File or Directory	Exists	Writable	CRC
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐

Please check out these files from your version control system if necessary

4 If a **CRC** option is **not** selected, the associated file has been changed outside the control of Code Review. To continue to refactor the code from model edits, either:

   ◇ Temporarily replace the updated file with the original. After the code is refactored, merge the file that failed the CRC with the refactored file.

   ◇ Choose **Tools>Update Model from Code Edits** to update the model from the source code. Note, however, that all changes made to the model since the model was last built will be lost.

5 If a **Writeable** option is not selected, the read-only file attribute is probably enabled. You need to disable the read-only attribute in order to continue to refactor the code with model edits. For instance, if you are using a source code control system, check out the files in the list.

6 To refactor the code, click **Refactor.** If any of the files do not exist, are not writable, or have failed the CRC, refactoring cannot take place and a warning message is displayed.

Creating a Design Reference

A *design reference* enables you to design a package structure in the unified modelling language (UML) and enforce this design when it is implemented in Java. It contains packages and dependencies but no classes. Unlike a source model, it is not based on source code. Design models have the extension **`.pdm`**. You can share this design with others to ensure that every-one on the project is following the correct design.

To create a new reference package design:

1 Choose **File>New Design Reference.** This creates a default package.

2 Create the required packages in the Tree View. To create new packages, right-click and choose **New package**. To create dependencies, right-click and choose **Dependencies**. Dependencies in the design can only be created between packages with the same parents.

3 Save the design reference.

Comparing a Code Model with a Design Reference

If you are developing source code using a design reference as a guideline to the package struc-ture, you should periodically verify that the source code conforms to the design.

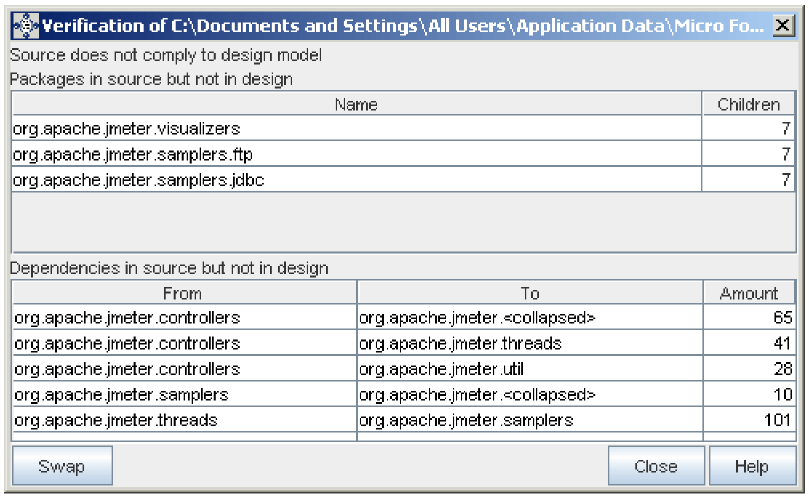
To compare a code model with a design reference:

1 Create or load a code model for the source code. See “Creating a Code Model” on page 8.

1 Choose **Tools>Compare Model with Design.**

2 Select the **`.pdm`** file containing your design reference.

3 Code Review checks the code model on the package level. The results of the comparison are shown in the **Verification** dialog box.


Verification of C:\Documents and Settings\All Users\Application Data\Micro Fo...

Source does not comply to design model

Packages in source but not in design

Name	Children
org.apache.jmeter.visualizers	7
org.apache.jmeter.samplers.ftp	7
org.apache.jmeter.samplers.jdbc	7

Dependencies in source but not in design

From	To	Amount
org.apache.jmeter.controllers	org.apache.jmeter.<collapsed>	65
org.apache.jmeter.controllers	org.apache.jmeter.threads	41
org.apache.jmeter.controllers	org.apache.jmeter.util	28
org.apache.jmeter.samplers	org.apache.jmeter.<collapsed>	10
org.apache.jmeter.threads	org.apache.jmeter.samplers	101

Dependencies between classes are not verified. Code Review combines all classes and interfaces in a collapsed classes package and verifies the dependencies between this temporary package and the other packages. You can model collapsed folders by giving a package the name `<collapsed>` (including angle brackets).

By default, the dialog box shows whether the design matches the source: that is, whether objects that are in the source code are also in the design reference. In effect, it tells you whether the design reference has been updated as development progresses.

You may, however, want to see the opposite perspective: whether the source code complies with the design reference. To list objects that are in the design reference but not in the source code, click **Swap**.

To return to the previous perspective, click **Swap** again.

Configuring an External Editor

You can access an external editor to view or edit source code from several places in Code Review. To do so, you must first configure your preferred editor to open files. No internal text editor is included.

To set the external editor:

1 Choose **Options>External Editor.**

2 Enter the command to start up your editor of choice. Depending upon the editor, you can use the following options to open the specified source file at the appropriate line:

- ◇ `<file>` — Specifies the file to open.
- ◇ `<startline>` — Specifies the line number in the code.

For example, the following command configures TextPad as the external editor:

```
textpad.exe -u <file>(<startline>
```

Note: Some text editors do not support opening a file at a specific line number. For example, the command for Notepad would have to be entered as

```
notepad.exe <file>
```

Consult the documentation shipped with your preferred editor for details of command line options and parameters.

Command Line Interface

Note: For information about the Metrics publishing command-line utility for publishing Code Review metrics to Optimal Delivery Manager, refer to the PDF file *Metrics Publishing Utility User Guide* (**`PubMetrics.pdf`**) in the DevPartner Java Edition product folder.

The majority of Code Review features for analysis, refactoring, and reporting can be used from the command line. The command line interface is useful in a variety of circumstances. For example, an automated analysis could be incorporated into the build process of your product and the results included with the build report.

Using the appropriate `setoption` parameters described below, you can view the generated Code Review report in text or XML format. The report can be saved as a file or printed to the console.

Two commands for reporting are unique to the command line interface:

- `creatediagrams` — Save a tree of package diagrams with an HTML page in the specified folder (`dir`).
- `uml2javadoc` — Add UML diagrams to an existing javadoc tree based on the current model. Optionally, set width and height.

Note: The use of `uml2javadoc` to enhance Javadoc is demonstrated in the tutorial "Enhancing Javadoc with UML Diagrams" on page 87.

Command Syntax

While the command line interface is compatible with ANT, using the command line interface independently requires the use of a script file.

Note: Only the `help` command is designed to function outside a script.

Each line of the script file must be one of the commands or options shown in the tables (see "Command-Line Options" on page 36 and "Commands" on page 38). Lines that start with # are ignored and can be used as comments.

The option(s) must precede the command(s) in the script. If you are creating a new code model, the command `newjava` must precede all other commands. If you are creating a new design model, the command `newdesign` must precede all other commands.

The environment variables `JAVA_HOME` and `ADVISOR_HOME` must be set to the home folder of your JDK installation and the home folder of your DevPartner Java Edition installation, respectively.

Use the following command to invoke a script file:

```
"%JAVA_HOME%/bin/java" -Xmx300M -enableassertions
-DADVISOR_HOME="%ADVISOR_HOME%" -Djava.library.
path="%PATH%" -cp "%JAVA_HOME%\lib\tools.
jar";"%ADVISOR_HOME%\lib\DLM40JNI.
jar";"%ADVISOR_HOME%\bin\Code Review.jar" com.compuware.
advisor.application.commandLine.CommandLine script
<filename.txt>
```

In this script file, `<filename.txt>` is the fully qualified path to the script file.

If the path to the script file contains any spaces, enclose the argument between quotation marks (`" "`). This requirement applies whenever a path is used as an argument, whether at the command line or within a script.

Command-Line Options

When you include `setoption` in your Code Review script, it sets the value for the specified option in the **`advisor.cfg`** file. You must include a separate `setoption` command, one per line, for each option you want to define for your code model or new design.

The following table describes the options. The syntax is as follows.

```
setoption <option> [argument]
```

Table 1-1. Command-Line Options

Option	Valid Values	Description
autosave *value*	TRUE or FALSE	Whether to save the model automatically after building it.
basedir *value*	The path.	The base folder. Set in ANT, also usable as a standalone script option.
bcExcludePath *value*	The path(s) of the files/ folders.	A semicolon-separated path of files/ folders that need to be excluded from bytecode analysis.
buildforrefactoring *value*	TRUE or FALSE	Enable refactoring to be switched off to increase performance when refactoring is not needed.
bytecodepath *value*	—	The bytecode path to use when building the code model.
bytecodesourcepath *value*	—	The source path of bytecode.
classpath *value*	—	The classpath to use when building.
collapsed *value*	TRUE or FALSE	Collapse classes (true or false).

Table 1-1. Command-Line Options (Continued)

Option	Valid Values	Description
dependencyweights *value*	TRUE or FALSE	Show weights in diagrams.
inputPath *value*	—	Any additional bytecode path(s) added by the user.
nodesize *value*	7 through 12	The size of the font used for nodes in diagrams.
packagesizemax *value*	—	Recommended maximum package size.
precision *value*	0 (zero) through 10	The number of digits behind the `.` (period) when outputting values.
quiet *value*	—	Suppress output to standard out.
reportexecution *value*	TRUE or FALSE	Show timing and memory usage for executed commands
reportFile *value*		The output file name. If the filepath is not provided, then the output file is created in the launch folder.
reportFormat *value*	text, Text, TEXT, xml, XML	The format type of the report (text or XML).
ruleset *value*	—	The list of active rules, separated by a backslash (\ ).
scExcludePath *value*	—	A semicolon-separated path of files/ folders that need to be excluded from source code analysis.
showdependencies *value*	TRUE or FALSE	Show dependencies in the ADP report.
source14 *value*	—	Use the `-source14` option when building.
sourcepath *value*	—	The sourcepath to use when build-ing.
usesourceparser *value*	TRUE or FALSE	Parse source (TRUE) or parse bytecode (FALSE).

Commands

The following table describes the commands that can be included in a script file to execute Code Review functionality from the command line.

Table 1-2. Command

Command	Arguments	Description
help	—	Give the list of Code Review scripting commands.
newdesign	—	Create a new empty design.
newjava	sourcePath	Parse Java source to a new model. Argument is a list of directories/files separated by semicolons.
adddependency	from to	Add a dependency between the `from` and `to` element.
adpreport	[prediction] (see Notes)	Generate a list in HTML of undesirable dependencies for the current model, based on the ADP.
adpreporthtml	outputfile	Generate a list in HTML format of undesirable dependencies for the current model.
applytosource	—	Apply the accumulated refactoring steps to the source.
checkrules	packagename or classname	Check the element for any rule violations and record violations in an XML file.
cleanimports	[prediction]	Remove unused imports.
codereport	file	Generate an HTML file with a summary report on code validation.
creatediagram	file package [w [h]]	Save the diagram of a package or class in the file in PNG format.
creatediagrams	dir [w [h]]	Save a tree of package diagrams with an HTML page in the specified folder (`dir`).
delete	name	Delete the named element.
deletedependency	from to	Delete the dependency between the given elements.
designreport	file	Generate an HTML file with a summary report on design validation.
flatten	packageName [false]	Set the flatten value for a given package name; add `false` to switch off.
flattenall	[false]	Set the flatten value for all packages; add `false` to switch off.

Table 1-2. Command (Continued)

Command	Arguments	Description
metric	packagename metricname [prediction]	Give the quality metric value for the given package. The following metrics are supported: NONE, ADP, DIP, SAP, EP, LSP, ADPR, Ca, Ce, I, Nc, Na, A, D', Nde.
move	from to	Move an element.
newpackage	name	Create a new package.
open	file	Open the model in the given file.
publishcodeviola-tions	filepath/file.xml	Export code violations to the speci-fied file.
publishdesign	filepath/file.xml	Export the design report to the speci-fied file.
rename	name newName	Rename an element.
save	file	Save the model to the given file.
script	file [quiet]	Open the file and interpret its con-tents as commands.
uml2javadoc	javadocroot [w [h]]	Add UML diagrams to an existing java doc tree based on the current model. Optionally, set width and height.
verifydesign	file	Verify the current model against the design in the file.
xmlmetrics	file	Export, in XML format, all the metrics of the model built using the `new-java` command.

Notes:

- If the argument `[prediction]` is set, the result of the command is compared to the pre-diction. If the prediction is different from the result, an error message is generated. When used from ANT, a BuildException is raised.
- Where the value of `param?` is a list, unless otherwise stated a string of items separate by semicolons is required. Where a value is required but not specified in the description, the value is Boolean. Where `[w[h]]` is required, it refers to the dimensions of the diagram in units of pixels.

Example Script Files

The following example shows the use of options and commands in a command-line script.

```
setoption scExcludePath /usr/local/share/WorkingCode/SnmpWebMonitor40src/IntroGUI.java
setoption scExcludePath /usr/local/share/WorkingCode/SnmpWebMonitor40src/IntroGUI.java
setoption classpath /usr/local/share/WorkingCode/SnmpWebMonitor40src/activation.jar
newjava /usr/local/share/WorkingCode/SnmpWebMonitor40src
designreport /tmp/sampleDesign.html
```

The following script file executes the tutorial "Improving a Package Structure" at the command line.

Note: To create a working code model to which subsequent commands refer, a script should always begin with a `newjava` command.

```
newjava C:/jmeter/source
move org.apache.jmeter.JMeterUtils org.apache.jmeter.util
newpackage org.apache.jmeter.top
move org.apache.jmeter.Driver org.apache.jmeter.top
move org.apache.jmeter.TextDriver org.apache.jmeter.top
move org.apache.jmeter.JMeter org.apache.jmeter.top
move org.apache.jmeter.JMeterEngine org.apache.jmeter.top
move org.apache.jmeter.threads.JMeterThreadAdapter org.apache.jmeter.controllers
move org.apache.jmeter.threads.HTTPJMeterThread org.apache.jmeter.controllers
deletedependency org.apache.jmeter.util.JMeterUtils org.apache.jmeter.visualizers
applytosource
report
```

The following script specifies a format and output file for a report for saving the output of the `checkrules` command.

```
setoption reportFormat xml

setoption reportFile c:\temp\jClassLib-viols.xml

newjava "C:\src\Common\Tools\jclasslib\src"

checkrules org.gjt.jclasslib
```

Type the following script at the command prompt to view a list of Code Review commands.

```
| "%JAVA_HOME%/bin/java" -Xmx300M -enableassertions

| -DADVISOR_HOME="%ADVISOR_HOME%" -Djava.library.

| path="%PATH%" -cp "%JAVA_PATH%\lib\tools.

| jar";"%ADVISOR_HOME%\lib\DLM40JNI.

| jar";"%ADVISOR_HOME%\bin\Code Review.jar" com.compuware.

| advisor.application.commandLine.CommandLine help
```

Note: If a Code Review License Error occurs, set `-Djava.library.path` to the Distributed License Manager's installation location. By default, the license manager is installed in **`C:\Program Files\Common Files\Compuware`**.

Metrics Publishing Utility

Code Review calculates two quality metrics, Maintainability and Compliance, that can be published to the Optimal Delivery Manager (ODM) using the Metrics publishing utility. For complete information about these metrics and the utility, see the PDF file *Metrics Publishing Utility User Guide* (**`PubMetrics.pdf`**), which is provided in the DevPartner Java Edition installation folder.

Before publishing the metrics, you must export them from Code Review. The Maintainability metric is exported through the Design Validation Summary page. For more information, see "Exporting Design Reports" on page 27. The Compliance metric is exported through the Code Validation Summary page. For more information, see "Exporting Rule Violation Reports" on page 13.

ANT Integration

Code Review supports use of the command line interface with ANT.

Invoking ANT with the Code Review command line interface enabled requires a batch file with the following settings.

```
| @echo off

| set cp="%ANT_HOME%"\lib\ant.jar
```

```
set cp=%cp%;"%ANT_HOME%"\lib\ant-launcher.jar
set cp=%cp%;"%ADVISOR_HOME%"\lib\DLM40JNI.jar
set cp=%cp%;"%JAVA_HOME%"\lib\tools.jar
set cp=%cp%;"%ADVISOR_HOME%"\bin\Code Review.jar
set PATH=%ADVISOR_HOME%\lib;%PATH%
"%JAVA_HOME%/bin/java" -Xmx300M -Xms100M -enableassertions
-Djava.library.path="%PATH%"
-Dadvisorhome="%ADVISOR_HOME%" -classpath %cp% org.apache.
tools.ant.Main %vb% %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Note: The listing is valid for Microsoft Windows operating systems; the settings can be applied to any operating system that Code Review supports.

To use the command line interface with ANT, use the `AdvisorAnt` class in the task definition.

```
<taskdef name="advisor" classname="com.compuware.advisor.appli-
cation.ant.AdvisorAnt">
<classpath>
<pathelement path="${advisorhome}\bin\Code Review.jar"/>
</classpath>
</taskdef>
```

The standard ANT functionality is extended to include the Code Review command line interface. When constructing ANT files, there are likely to be alternative ways to achieve a particular objective. For instance, Code Review supports the ANT `fileset` element and it is possible to specify a list of files either using `include` elements or a single string of semicolon separated items.

Performing an Analysis with ANT

The following ANT listing demonstrates some of the alternative ways to perform an analysis. It analyzes Code Review itself and begins by defining a new bytecode model reliant upon a single file location. This definition then evolves to create a class diagram and to include a more precise specification of the resources.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project basedir="." default="all" name="antSample">
<property name="output" value="${advisorhome}\bin"/>
<property name="3rdparty"
value="C:\Development\MyProject\3rdparty"/>
<taskdef name="advisor"
```

```
classname="com.compuware.advisor.application.ant.AdvisorAnt">
<classpath>
<pathelement path="${advisorhome}\bin\Code Review.jar"/>
</classpath>
</taskdef>
<!— bytecode parsing of a complete folder using the
'setoption' command to enable bytecode parsing —>
<target name="example1">
<advisor command="setoption" param1="usesourceparser"
param2="false"/>
<advisor command="newjava" param1="${output}"/>
</target>
<!— Same as example1 but now using the 'advisor' element
'usebytecode' attribute to enable bytecode parsing —>
<target name="example2">
<advisor command="newjava" usebytecode="true"
param1="${output}"/>
</target>
<!— Include additional resources and create class diagram —>
<target name="example3">
<advisor command="newjava" usebytecode="true"
param1="${output};${3rdparty}\util.jar"/>
<advisor command="setoption" param1="showdependencies"
param2="true"/>
<advisor command="creatediagram" param1="diagram.png"
param2="com.compuware.advisor" param3="400" param4="400"/>
</target>
<!— Same as example3 but now using the ANT 'fileset' element to
specify the resources —>
<target name="example4">
<advisor command="newjava" usebytecode="true"
```

```
param1="${output}">
<fileset dir="${output}">
<include name="${3rdparty}\util.jar"/>
</fileset>
</advisor>
<advisor command="setoption" param1="showdependencies"
param2="true"/>
<advisor command="creatediagram" param1="diagram.png"
param2="com.compuware.advisor" param3="400" param4="400"/>
</target>
<!— Exclude specific classes from analysis —>
<target name="example5">
<advisor command="newjava" usebytecode="true"
param1="${output}">
<fileset dir="${output}">
<include name="${3rdparty}\util.jar"/>
<exclude name="**\ClassToExclude.class"/>
</fileset>
</advisor>
<advisor command="setoption" param1="showdependencies"
param2="true"/>
<advisor command="creatediagram" param1="diagram.png"
param2="com.compuware.advisor" param3="400" param4="400"/>
</target>
<target name="all"
depends="example1,example2,example3,example4,example5"/>
</project>
```

Sample ANT File

The following listing is a complete ANT file that executes the tutorial Improving a Package Structure.

Note: The particular command that is specified defines the number of associated attributes.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project basedir="." default="all" name="antSample">
<property name="advisorhome" value="C:\Program
Files\Micro Focus\DevPartner Java Edition"/>
<property name="jmeter"
value="${advisorhome}\samples\org\apache\jmeter"/>
<taskdef name="advisor"
classname="com.compuware.advisor.application.ant.AdvisorAnt">
<classpath>
<pathelement path="${advisorhome}\bin\Code Review.jar" />
</classpath>
</taskdef>
<!— Code Review Improving a Package Structure tutorial —>
<target name="advisor">
<advisor command="newjava" param1="${jmeter}"/>
<advisor command="move"
param1="org.apache.jmeter.JMeterUtils"
param2="org.apache.jmeter.util"/>
<advisor command="newpackage"
param1="org.apache.jmeter.top"/>
<advisor command="move" param1="org.apache.jmeter.Driver"
param2="org .apache.jmeter.top"/>
<advisor command="move" param1="org.apache.jmeter.TextDriver"
param2="org.apache.jmeter.top"/>
<advisor command="move" param1="org.apache.jmeter.JMeter"
param2="org.apache.jmeter.top"/>
<advisor command="move"
param1="org.apache.jmeter.JMeterEngine"
param2="org.apache.jmeter.top"/>
<advisor command="move" param1="org.apache.jmeter.threads.
JMeterThreadAdapter" param2="org.apache.jmeter.controllers"/>
```

```
<advisor command="move" param1="org.apache.jmeter.threads.
HTTPJMeterThread" param2="org.apache.jmeter.controllers"/>
<advisor command="deletedependency"
param1="org.apache.jmeter.util.JMeterUtils"
param2="org.apache.jmeter.visualizers"/>
</target>
<target name="all" depends="advisor"/>
</project>
```

Customized Reports

Reports can be exported for code rule violations, from the Code Validation Summary page; and for metrics, from the Design Validation Summary page.

Code Rule Violations

Use the **Export Violations** dialog box to export code rule violations to a file. The dialog box includes the options **HTML Reports** and **Print Format Reports**. Both options rely on XSL stylesheets to format the reports.

The stylesheets format an XML stream with the same content as the file created using the **Violations to XML** option. The following fragment is taken from a Code Review XML report. It includes the elements and attributes available to the stylesheet.

```
<?xml version="1.0" ?>
<data>
<date>Fri Oct 24 08:16:35 CEST 2005</date>
<summary>
<graphBackgroundColor>F5F0EB</graphBackgroundColor>
<totalViolations nbr='89'>89<high nbr='0' percent='0'
/> <medium nbr='12' percent='13' /><low nbr='77' percent='86'
/></totalViolations>
<classesWithMostHighViolations>
</classesWithMostHighViolations>
</summary>
<details>
<violation class='org.apache.jmeter.util.JMeterUtils'
```

```
| file='C:\Program Files\Micro Focus\Code
Review\samples\org\apache\jmeter\

| util\JMeterUtils

| .java' line='693' ruleid='3' description='R0003: Public

| class exposes a public field' notes='' category='Design'

| severity='2' severitytext='Medium' />

11| ...

12| <violation class='org.apache.jmeter.visualizers

| .WindowedVisualizer' file='C:\Program Files\Micro Focus\DevPartner
Java Edition\examples

| \org\apache\jmeter\visualizers\WindowedVisualizer

| .java' line='186' ruleid='1011' description='R1011:

| Constructor doesn t initialize all package and private

| fields' notes='' category='Design' severity='3'

| severitytext='Low' />

13| </details>

14| </data>
```

The **HTML Reports** and **Print Format Reports** lists are generated from a listing of the available stylesheets. The stylesheets are located in the ***DPJ_dir*\stylesheets\HTMLReports** folder and ***DPJ_dir*\stylesheets\PrintReports** folder, respectively, where ***DPJ_dir*** is the DevPartner Java Edition product folder.

The stylesheets can be edited to produce customized reports. Adding a new stylesheet to either of the aforementioned folders will create a new item on either the **HTML Reports** or the **Print Format Reports** list.

Metrics

Use the **Export Metrics** dialog box to export design violations and metrics to a file. The dialog box includes the options **HTML Reports** and **Print Format Reports**. Both options rely on XSL stylesheets to format the reports. For more information, see "Export Metrics" on page 61.

Chapter 2

User Interface

The User Interface topics describe views, dialog boxes, and menus. They are displayed when you click the **Help** option in a dialog box or menu.

Views and Dialog Boxes

The Code Review interface includes these views and dialog boxes.

General

- New Code Model
- Verification of Design Reference

Code Validation

- Code Validation Summary
- Code Validation Details
- Custom Rules
- Rule Filter
- Export Violations
- Refactor Code With Model Edits

Design Validation

- Design Validation Summary
- Export Metrics
- Design Validation Details
- Class Diagram Pane
- Tree Pane
- Dependency View
- Usage View
- Dependency Table
- Metrics Window

New Code Model

Use this dialog box to specify the parameters for creating a new code model from source code or bytecode.

Options

- **Source Code (allows refactoring)** — Allows refactoring.
- **Bytecode (faster)** — Builds more quickly.
- **Exclude Path** — List of directories and files where the files to be excluded can be found. If all the files are in one source tree, give a path consisting of the root of this tree.
- **Save model after building** — Saves the code model automatically to the working folder on completion of the build.

The rest of the options vary depending on the selected input:

- **Source Settings**
  - **Source Path** — List of directories and files where the source files can be found. If all the sources are in one source tree, enter a path consisting of the root of this tree.
  - **Class Path** — Java classpath used to build the model. This should be the same as the classpath used to compile the sources.
  - *If JDK 1.4 is installed:* **Enable 'assert' keyword (build with -source 1.4)** — Enables the new syntax introduced in JDK 1.4 (most notably the `assert` keyword).*If JDK 1.5 (5.0) is installed:* **Accept JDK 1.5 language features (build with -source 1.5)** — Enables the new syntax introduced in JDK 1.5 (also called JDK 5.0).
  - **Allow Refactoring (faster when off)** — Enables refactoring of the source files from the code model. This option requires additional metadata, so it is much faster (typically twice as fast) to build a code model with this option cleared. However, you *must* select this option to synchronize the source files with edits in the code model.
- Bytecode Settings
  - **Bytecode Path** — List of directories and files where the compiled bytecode files can be found. If all the class files are in one source tree, give a path consisting of the root of this tree.
  - **Source Path** — Optional list of directories and files where the source files can be found. If all the sources are in one source tree, give a path consisting of the root of this tree. Set this field to view the source code in an external editor.

Use

- See “Creating a Code Model” on page 8.

Verification of Design Reference

This dialog box displays the results of comparing a code model with a design reference.

Fields

The dialog box states whether the source complies with the design; that is, it indicates whether any dependencies in the source are not in the design.

Packages in source but not in design:

- **Name** — Name of noncompliant package.
- **Children** — Number of child packages.
- **Amount** — Dependencies between packages. Dependencies between classes are not verified.

Dependencies in source but not in design:

- **From** — Package containing noncompliant dependency.
- **To** — Package depended upon.
- **Amount** — Dependency weight in the source. Dependency weight is an estimate of the number of places in the source code where one element depends on another.

Actions

You can edit the model while displaying this dialog box. The design verification results are updated on the fly.

Code Validation Summary

The Code Validation Summary page shows a summary of the code rule violations in the code model. The violations are grouped into three categories:

- Violations By Severity
- Classes With Most High Severity Violations
- High Severity Rules With Most Violations

Clicking any of the hyperlinks under the three category headings will open the Code Validation Details page.

Note: The categories **Classes With Most High Severity Violations** and the **High Severity Rules With Most Violations** list only the top five violations, even if there are more than five violations in that category. To see the complete list of violations, click any link to display the Code Violation Details page.

The Compliance index uses a weighted formula based on dividing the number of violations by the lines of source code. It can only be calculated for a Source Code model; it is not available for Byte Code models. A lower score indicates fewer code violations.

Options

- **Select Rules** — Select the rules to use in the analysis.
- **Export** — Print a report, or export an HTML or XML report of the rule violations.

Use

- To view the Code Validation Details for a category, click the appropriate link. For more information, see “Code Validation Details” on page 52.
- See “Viewing Rule Violations” on page 12.
- See “Exporting Rule Violation Reports” on page 13.

Code Validation Details

The Code Validation Details page displays information about specific rule violations. Rule violations are listed in a violations table. Detailed information for a selected violation is displayed in the **Source Fragment** and **Rule Details** panes.

Note: If you go to the Custom Rules page from the Code Validation Details page, change the rules selection, and rebuild the code model by clicking **Start**, the Code Validation Details page is not refreshed automatically. To update the rules violation list, click the **Summary** button on the Code Validation Details page to go to the Code Validation Summary page, then click any link on the summary page to return to the details page.

Violations Table

The table groups violations according to the selection in the **Group By** field. When you click a link in the Code Validation Summary page, the appropriate selection is made automatically; you can change the display by selecting another item from the list.

The violations table contains the following columns, depending on the **Group By** selection:

- **Class** — The class in which the rule violation occurs
- **Rule** — The number and brief description of the rule
- **Severity** — Severity of the rule violation (High, Medium, or Low)
- **Category** — The rule category (ruleset).
- **Source Code/Line** — The location of the rule violation in the source code

Text immediately below the table refers to collections of rule violations within the main group; for example, **Severity** consists of three collections arranged according to **High, Medium,** and **Low** severity. However the violations are grouped, clicking the text will redisplay the table for that collection.

Note: Click the column heading to sort the items by that heading.

Panes

- **Source Fragment** — Displays the source code with the violating code highlighted. Clicking the link at the top of the pane opens the source code into the editor.
- **Rule Details** — Displays the rule being violated, notes, a detailed explanation, and possible remedies.

Note: References to *Effective Java, 2001 ed.* in the rule details refer to Joshua Bloch's book *Effective Java Programming Language Guide*, Addison-Wesley, 2001.

Options

Violations Table

- **Summary** — Return to the Code Validation Summary page.
- **Group by** — Display the details organized according to the selected group.
- **Class** lists the classes that contain code identified as violating a rule, with the violated rule(s) listed under the class.
- **Rule** lists the rules that are violated; for each rule, the table lists the class and specific code line containing the violation.
- **Severity** groups the violations by High, Medium, and Low severity.
- **Category** specifies the category (ruleset) of the violated rule.

Rule Details

- **Suppress Rule** — Filters the rule out of the current code analysis.

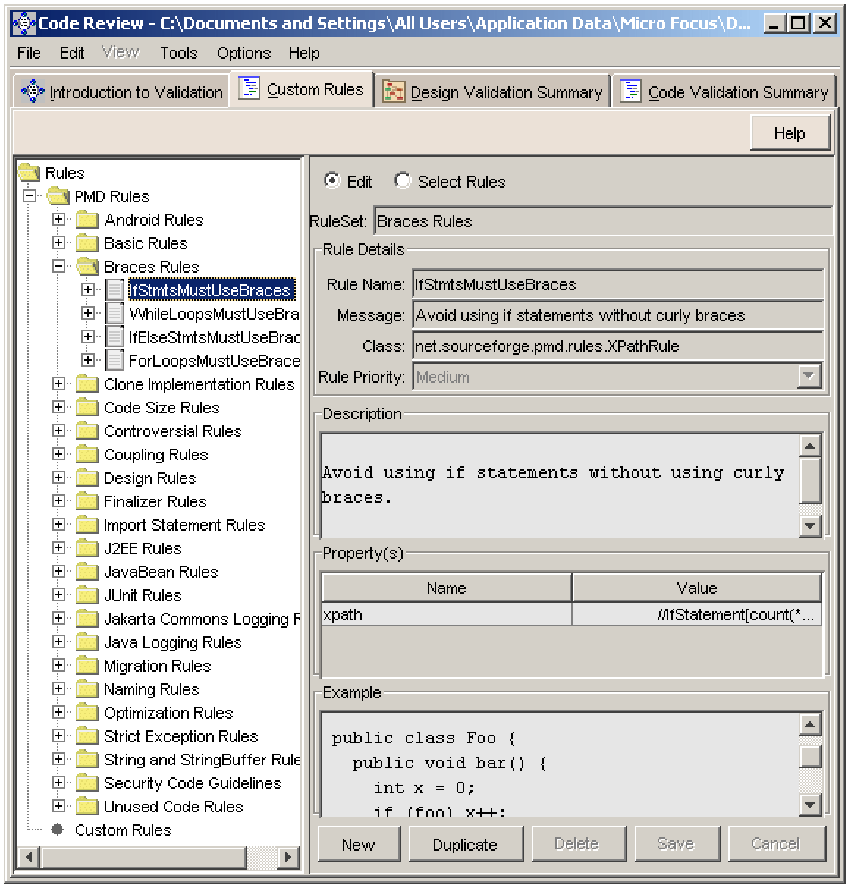
Custom Rules

Code Review incorporates the PMD source code analyzer.

Note: PMD rules are not available for bytecode models. The Custom Rules page is available, however, when you open a bytecode model. You can create new rules while a bytecode model is open, but the PMD and custom rules will only be applied if you rebuild the model with **Source Code** selected.

The Custom Rules page lists the PMD and custom rules grouped into *rulesets*, which are the same as the built-in rule *categories*. When you select a rule, its definitiondisplays in the right pane. The name of the ruleset containing the rule displays at the top of the right pane.

Figure 2-1. Custom Rules Page



(● Edit) ○ Select Rules

RuleSet: Braces Rules

Rule Details

Rule Name: IfStmtsMustUseBraces

Message: Avoid using if statements without curly braces

Class: net.sourceforge.pmd.rules.XPathRule

Rule Priority: Medium

Description

```
Avoid using if statements without using curly
braces.
```

Property(s)

Name	Value
xpath	//IfStatement[count(*...

Example

```
public class Foo {
  public void bar() {
    int x = 0;
    if (foo) x++;
```

Java class rules and XPath rules are the two PMD rule types used in custom rulesets. Use the provided PMD rulesets with both rule types as provided (factory default) to create custom rules. You can duplicate both Java class rules and XPath rules for your own custom rulesets. You can also modify rules duplicated into your custom rulesets, as well as add your own new XPath rules, to provide maximum flexibility and customization to your rulesets. Both Java class and XPath custom rules can be deleted.

For detailed information on how to write coding rules, see "How to write a PMD rule" in the PMD web site http://pmd.sourceforge.net/howtowritearule.html.

You can only change the priority, and rule message fields for custom Java class rules.

Custom rules data are stored in XML files in //Documents and Settings/All Users/Application Data/Micro Focus/DevPartner Java Edition/var/rulesets.

In the Custom Rules page, you can:

- Create custom rules.
- Edit custom rules.
- Delete custom rules.
- Select PMD rulesets to include in a code model.

Creating a Custom Rule and Ruleset

An existing rule in any PMD ruleset must be selected to create a custom rule. Custom rules are stored in the Custom Rules node.

1 Select an existing rule in a ruleset. If planning to store the rule in a custom ruleset that is based on an existing ruleset, select a rule in the existing PMD ruleset.

2 Do one of the following:

- ◇ To create any type of custom rule by duplicating an existing rule, click the **Duplicate** button on the right pane. The text of the selected rule remains in the form; all fields except **Class** and **Property(s)** become active for editing.

  Note that you can duplicate both PMD rules and custom rules. The rule is duplicated into a custom ruleset under the Custom Rules nodes in the tree.

- ◇ To create an new XPath rule not based on an existing rule, click the **New** button on the right pane. The rule is created in a new custom ruleset under the Custom Rules nodes in the tree.

3 As needed, create or edit field contents.

- ◇ **Rule Set** — The name of the custom ruleset for the custom rule. This can be the name of an existing custom ruleset or a new custom ruleset.

  When clicking the **New** or **Duplicate** button to create a new rule, a default ruleset name appears in the RuleSet field on the right pane. The default name is based on the ruleset of the selected rule, with the prefix `My` added to the ruleset name. For example, if you create new rule with a rule from the Basic Rules ruleset selected, the default ruleset name for the new rule is `MyBasic Rules`.

  Once saved, the custom ruleset name cannot be changed.

- ◇ **Rule Name** — Use a brief, descriptive name. You can include spaces and special characters if desired. The default name is `My` followed by the originating rule name. New rules must have unique names.
- ◇ **Message** — Summarize the purpose of the rule.
- ◇ **Class** — Displays the rule class name. This field cannot be changed.
- ◇ **Rule Priority** — Select **High**, **Medium**, or **Low** from the list.
- ◇ **Description** — Provide a brief description of the rule or ruleset. You can only create or edit a description for XPath type rules.
- ◇ **Property** — For XPath rules, this displays the XPath statement of the rule. The XPath refers to the path in the Abstract Syntax Tree (AST). For details, see the XPath Rule tutorial in the PMD Web site, http://pmd.sourceforge.net/xpathruletutorial.html. Any additional XPath properties are copied from the originating rule and can not be edited.
- ◇ **Example** — For new rules or XPath type duplicated rules, provide an example that would be identified as "bad code" by the rule. Duplicated rules that are not XPath type rules display the example of the originating rule, which cannot be edited.

4 Click **Save** to create the custom rule. A message appears indicating that the custom rule has been created.

5 Click **OK**. The rule is added to the Custom Rules node within the new custom ruleset. If a new custom ruleset name is specified, Code Review creates the custom ruleset to store the new rule.

You must rebuild the code model for any newly created custom rule to apply. See "Selecting Rulesets and Rebuilding the Code Model" on page 56 for more information.

Deleting a Custom Rule

Only custom rules can be deleted. PMD rules cannot be deleted.

To delete a custom rule:

1 Select the custom rule and click **Delete**. A message appears to confirm the deletion.

2 Click **Yes** to delete the rule, or **No** to cancel the deletion and return to the rule. If you click **Yes**, the rule is deleted. If the deleted rule is the only rule in its ruleset, the ruleset is also deleted. When a rule is deleted the preceding rule displays in the right pane.

Updating a Custom Rule

Certain fields in a custom rule can be updated. Fields that can be updated depend on the custom rule's rule type.

1 Select the custom rule to be edited.

2 Make the desired changes. Fields that cannot be edited are disabled.

For XPath rules, you can update the **Message**, **Rule Priority**, and **Example** fields.

For Java class rules, you can update the **Message** and **Rule Priority** fields.

3 Click **Save** to update the custom rule.

Editing a Custom Ruleset Name

When creating a new rule or duplicating an existing PMD rule, the rule is placed in a custom ruleset. To change the name of the custom ruleset:

1 Click New or Duplicate to create a new rule. The ruleset name can only be edited at this time.

2 Type a new name for the ruleset in the **Ruleset Name** field. The name must be unique.

3 Click **Save**.

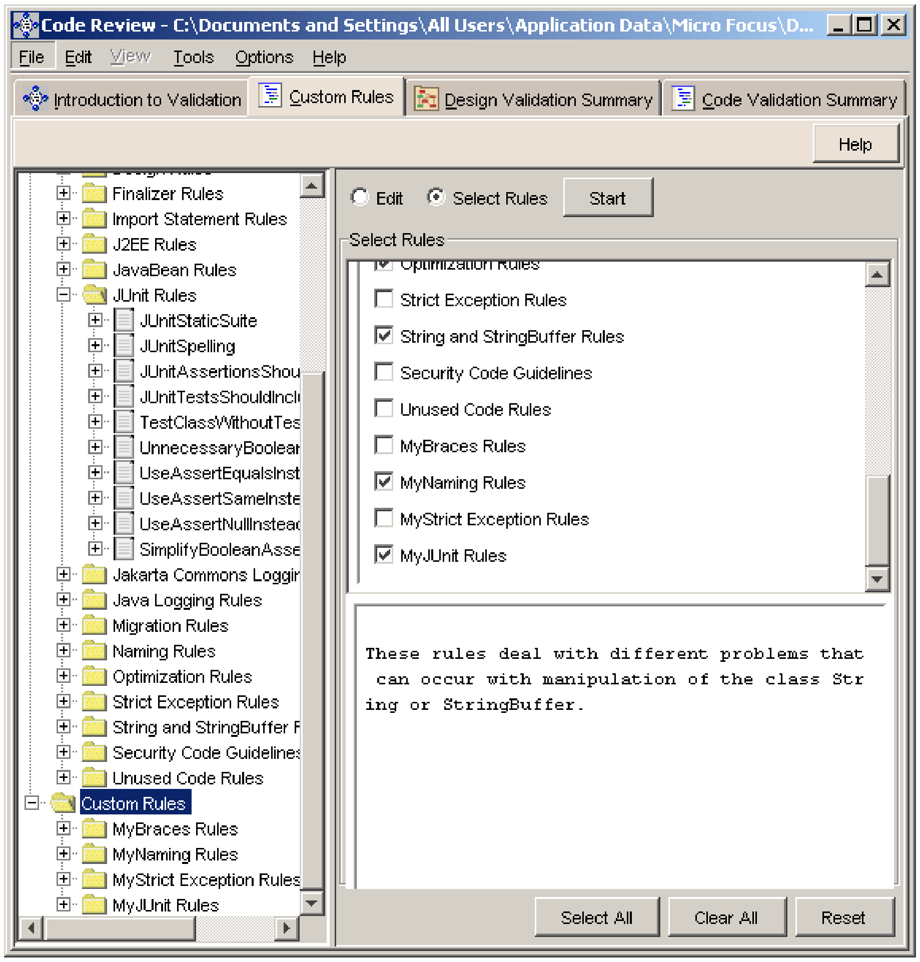
Selecting Rulesets and Rebuilding the Code Model

By default, only the PMD Basic and Design rulesets are used with Code Review's built-in rules when you create a source code model. Through the Custom Rules page, you can select other rulesets for rebuilding the model.

To select PMD rulesets (including any custom rules you created in the rulesets):

1 Select the Custom Rules page. By default, the Edit view displays.

2 Click **Select Rules.** A list of the rulesets appears in the right pane.



( ) Edit [✓ Select Rules]

Select Rules

- [✓] Optimization Rules
- [ ] Strict Exception Rules
- [✓] String and StringBuffer Rules
- [ ] Security Code Guidelines
- [ ] Unused Code Rules
- [ ] MyBraces Rules
- [✓] MyNaming Rules
- [ ] MyStrict Exception Rules
- [✓] MyJUnit Rules

```
These rules deal with different problems that
 can occur with manipulation of the class Str
ing or StringBuffer.
```

3 In the Select Rules list, select the check box or name of each ruleset you want to include (or click a selected ruleset to exclude it). When you select a ruleset, a brief description displays in the lower right pane.

Use the **Select All** or **Clear All** button at the bottom of the window to include or exclude all rulesets. To restore the default setting (Basic and Design only), click **Reset**. Selections are saved.

When you select rulesets, they are not automatically applied to the code model. You must rebuild the code model to include the selected ruleset.

4 Click **Start** to rebuild the code model using the selected rulesets. Once the code model is rebuilt, review the code rule violations in the code model. See "Code Validation Summary" on page 51 more information.

When you select rulesets, they are not automatically applied to the code model. You must rebuild the code model to include the selected rulesets. To rebuild the code model, click **Start.**

Note: If you go from the Code Validation Details page to the Custom Rules page, change the rules selection, rebuild the code model by clicking **Start**, and return to the Code Validations page, the page is not refreshed automatically.

To update the rules violation list, click the **Summary** button on the Code Validation Details page to go to the Code Validation Summary page, then click any link on the summary page to return to the details page.

Once a ruleset is selected, it is included in every code model.

Rule Filter

When you build a code model, all Code Review rules and all selected PMD and custom rules are used in analyzing the code. You can, however, filter the results to focus to display only specific types of rule violations.

Use the **Rule Filter** dialog box to choose the results to be displayed in the Code Validation Summary page and Code Validation Details page.

To open the **Rule Filter** dialog box, click **Select Rules** in the Code Validation Summary page.

The table listing the code rules used in the analysis includes the following columns:

- **Active** — Whether or not the rule is active.
- **Rule** — The rule name. Only the rules included in the analysis results are listed.
- **Severity** — Rule severity. Possible values are High, Medium, or Low.
- **Category** — The Code Review category or PMD ruleset, as applicable.

Click the column heading to sort the items by that heading.

When you select a rule in the table, the **Rule Details** pane displays an explanation of the rule, proposed remedies for violations, an example of code that violates the rule, and expert references. Depending on the rule, not all these details may be included in the text.

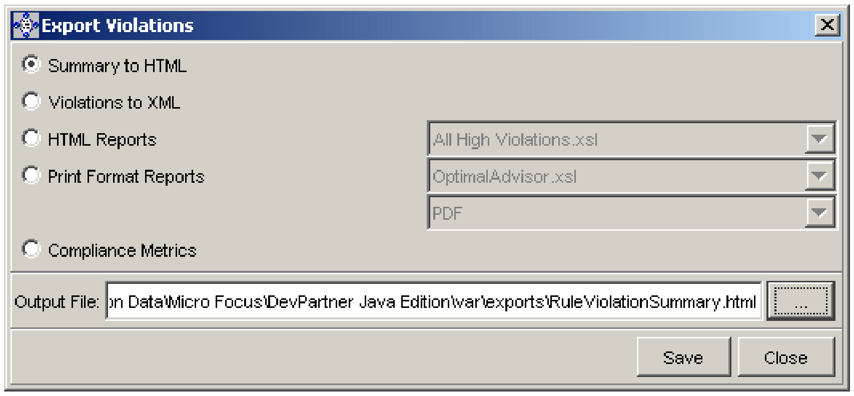
To filter the analysis results, use one of the following options. You can further refine the filter by selecting or clearing individual rules after using the option.

- **Check only** — Select an item from the list to include only the rules matching the selection.
- **Check All** — Select all the rules.
- **Clear All** — Clear all the rules.
- **Reset to Default** — Restores the list to the default selections.

You must rebuild the code model to apply the new filter to the results.

Export Violations

Use the **Export Violations** dialog box to export a report of rule violations in your preferred output format.



[✓ Summary to HTML]
Violations to XML
HTML Reports — All High Violations.xsl
Print Format Reports — OptimalAdvisor.xsl
PDF
Compliance Metrics
Output File: on Data\Micro Focus\DevPartner Java Edition\var\exports\RuleViolationSummary.html

- **Summary to HTML** — Export the summary page as an HTML file.
- **Violations to XML** — Export all violation details as an XML file.
- **HTML Reports** — Export a specific category of rule violation details as an HTML file. When you select this option, the list to the right of the option becomes active so you can select whether to export only High, only Medium, or all violations.
- **Print Format Reports** — When you select this option, the list to the right of it becomes active so you can select the desired format: PDF, PS (postscript), or text.
- **Compliance Metrics** — Export the value of the Compliance metric to an XML file for publishing to Optimal Delivery Manager. For more information, see “Metrics Publishing Utility” on page 41. Also refer to the separate documentation for the Metrics publishing utility.
- **Output file** — Path and file name for the exported report. You can type the path or click the browse button to navigate to the desired folder. The default file name is appended to the path, with the extension appropriate for the selected format.

The default location of the exported file depends on your operating system:

- Windows XP or 2003 — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\exports`**

Note: By default, the **`Application Data`** folder is hidden. To display the **`exports`** folder and its contents, type the path in the **Address** field of Windows Explorer and press Enter.

- Windows 2008 or Vista — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\exports`**
- UNIX — ***`DPJ_dir`*`/var/exports`**
  where ***`DPJ_dir`*** is the path of the Devpartner Java Edition product folder.

Use

- See “Exporting Rule Violation Reports” on page 13.

Refactor Code With Model Edits

Use this dialog box to review code model changes and the apply them to the source code.

Columns

The **Refactor Code With Model Edits** dialog box displays a table listing all changes made to the current model:

- **Type** — Type of edit; one of **move**, **rename**, **delete**, or **new**.
- **Element** — Element that has been edited.
- **From** — Old parent package or package name of the element (if applicable).
- **To** — New parent package or package name of the element (if applicable).
- **Applied to source** — Shows whether the edit has been applied.

Actions

- **Apply Edits** — Start refactoring the source code. A list of files that will be updated is displayed.

Design Validation Summary

The Design Validation Summary page shows a summary of the design violations in the code model. The violations are grouped into three categories:

- Packages Exceeding the Recommended Maximum Size of *n* Elements
- Packages with Most Inter Package Cyclic Dependencies
- Packages with Most Inter and Intra Package Cyclic Dependencies

The first category heading includes a **change size setting** link, which enables you to change the maximum number of elements (the default is 10).

Clicking any of the links under the three category headings opens the Design Validation Details page.

Option

- **Export** — Export reports in HTML, XML, PDF, postscript (PS), or text formats. See “Exporting Design Reports” on page 27.

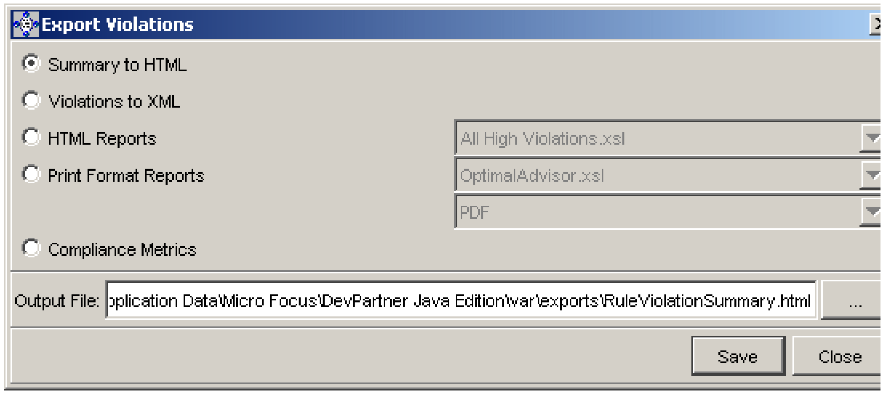
Use

- See “Viewing Design Violations” on page 25.

Export Metrics

Use the **Export Metrics** dialog box to export a report of design violations in your preferred output format.

Figure 2-2. Export Metrics Dialog Box



Summary to HTML [✓]

Output File: pplication Data\Micro Focus\DevPartner Java Edition\var\exports\RuleViolationSummary.html

- **Summary to HTML** — Export the Design Validation Summary page as an HTML file.
- **Metrics to XML** — Export metrics for each package as an XML file.
- **HTML Reports** — Export metrics for each package as an HTML file.
- **Print Format Reports** — Export metrics for each package in a printable format. When you select this option, the list to the right of it becomes active so you can select the desired format: PDF, PS (postscript), or text.
- **Maintainability Metrics** — Export the value of the Maintainability metric to an XML file for publishing to Optimal Delivery Manager. For more information, see “Metrics Publishing Utility” on page 41. Also refer to the separate documentation for the Metrics publishing utility.
- **Output File** — Path and file name for the exported report. You can type the path or click the browse button to navigate to the desired folder. The default file name is appended to the path, with the extension appropriate for the selected format.

The default location of the exported file depends on your operating system:

- Windows XP or 2003 — **`C:\Documents and Settings\All Users\Application Data\Micro Focus\DevPartner Java Edition\var\exports`**

Note: By default, the **`Application Data`** folder is hidden. To display the **`exports`** folder and its contents, type the path in the **Address** field of Windows Explorer and press Enter.

- Windows 2008 or Vista — **`C:\Program Data\Micro Focus\DevPartner Java Edition\var\exports`**
- UNIX — ***`DPJ_dir`*`/var/exports`**
  where ***`DPJ_dir`*** is the path of the Devpartner Java Edition product folder.

Use

- See “Exporting Design Reports” on page 27.

Design Validation Details

The Design Validation Details page consists of a navigation tree that lists the various packages, classes and methods, and a UML class diagram.

Use

You can view, navigate, and edit in the class diagram pane as described in "Viewing and Editing a Class Diagram" on page 63. You can navigate and edit in the tree pane as described in "Viewing and Editing a Package Tree" on page 64.

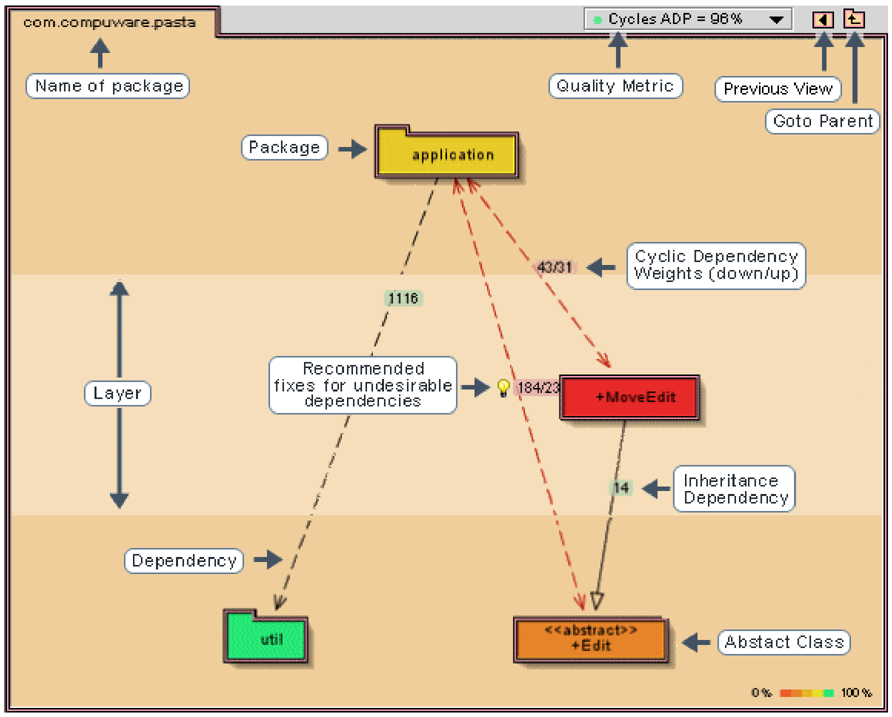
Class Diagram Pane

The class diagram pane shows the elements of a package and the dependencies between these elements.

Class Diagram Notation

The class diagram uses the unified modeling language (UML) notation, with some additions. This following graphic explains the symbols used in the class diagram pane.

Figure 2-3. Class Diagram Symbols



Elements that are referred to from the code, but for which the code itself is not part of the model, are shown in grey.

Package elements are laid out on layers in the diagram background to show the layered architecture of the application. A clear layered architecture requires each layer to depend only on lower layers — the application must not have cycles. If this is the case, Code Review proposes a layering structure, showing cyclic dependencies in red if the selected quality metric is the Acyclic Dependencies Principle. These undesirable dependencies must be removed for the diagram to become acyclic.

Use

You can view, navigate, and edit the diagram as described in "Viewing and Editing a Class Diagram" on page 63.

Viewing and Editing a Class Diagram

You can navigate, view, and edit a code model or design reference class diagram in the class diagram pane:

- To move objects, drag and drop them. You can move objects to improve the layout of a class diagram. When editing a code model, this does not affect the source code. Changes to the layout are saved as part of the model. However, when the contents of a package change, a new layout is computed.
- To focus on a specific object across layers, right-click the object and select **Add to Scratch Area**. The scratch area opens on the left side of the diagram and a copy of the object is moved into it (the object also remains in the diagram). The diagram is automatically rearranged as needed to display the connections between the scratch object and other objects. As you navigate through the layers, the object remains in the scratch area. To remove it, right-click the object and select **Remove from Scratch Area**. You can place more than one object at a time in the scratch area.
- To view detailed information about a dependency, double-click the dashed line extending from the object to open the Dependency View.
- To open the class diagram for a package, double-click the package.
- To navigate to the parent package of the current class diagram, click the **Show Parent** icon.
- To return to the previous view, click the **Previous View** icon.
- To edit a class or interface, right-click it and select **View Source** to open the external editor for that class. For more information, see "Configuring an External Editor" on page 34.
- By default, the class diagram displays ADP cycles. To view the analysis of the model using another quality metric, select the metric from the list above the diagram.
- To open the Metrics window or Usage View window, right-click the object of interest and select the appropriate option from the menu.
- To move an object into another package, right-click the object and select **Move** from the menu.
- Right-click the class diagram background (not on an object) to display a popup menu that enables you to print, save, or customize the view.

Tree Pane

The tree pane on the left side of the Design Validation Details page shows the package structure of the code model or design reference.

Tree elements

The tables list elements shown in the tree.

Table 2-1. Code model icons

Icon	Represents
📁/📁	Package
🗎/🗎	Java file with a public class definition
⊸○/⊸○	Java file with a public interface definition

All elements that are not packages are called *leaf* elements.

In a code model, classes, interfaces, or packages may be referred to from the source, without actually being in the source (because they are available in the classpath or are part of the Java standard library). These external elements are shown in grey.

Table 2-2. Design reference icons

Icon	Represents
📁	Package

Use

You can navigate and edit in the tree pane as described in "Viewing and Editing a Package Tree".

Viewing and Editing a Package Tree

You can create new packages and move elements in the tree pane:

1 To display the class diagram for a package, click a package to give it focus.

2 To move an element in the tree, drag and drop it to a new location in the package structure tree. You can also drop an element on the class diagram, in which case the target is the element represented in the class diagram pane.

Note: When editing a code model, the change is stored in the model and can be applied to the code by choosing **Tools>Refactor Code With Model Edits**.

3 To display a popup menu, right-click a tree element. The menu displayed depends on the element selected. If you have multiple items selected, right-click to display a menu that contains only **Move**. This enables you to move the selected elements to another package.

Dependency View

The **Dependency View** dialog box shows all the dependencies between two elements that have the same parent package.

Panes

- **Dependency table** — The table columns provide information about the dependencies:
  - **From** and **To** — The two packages between which dependencies are examined.
  - **Dependency Type** — A brief description of the dependency relationship.
  - **Line** — The line in the source code.

  For more details, see “Dependency Table” on page 66.
- **Source fragment** — Select an item in the dependency table to display the section of source code containing it, with the dependency highlighted.

Options

- **From** and **To** — Select an item in the list to enter it in the **From** column of the dependency table. The corresponding packages are listed automatically in the **To** list. If the **To** list contains more than one item, you can change the objects in the **To** column by selecting a different item in the list.
- **Swap** — Switch the **From** and **To** elements, if there are dependencies in the opposite direction. This field is only active if it is applicable to the listed elements.
- **Filter imports** — Ignore import dependencies.
- **Source Fragment** — Click the link to open the source code into an external editor.

Note: While the **Dependency View** dialog box is open, you can change the **From** and **To** elements by clicking a dependency in the class diagram pane.

Usage View

The Usage View window enables you to view the dependencies for a selected element, showing the dependencies from and to the element.

Panes

- **Dependency table** — Displays information about the dependencies in two tabs:
  - **Used by** — Shows all dependencies to a chosen model element.
  - **Uses from** — Shows all dependencies from a chosen element.

  Each tab contains four columns:
  - **From** and **To** — The two packages between which dependencies are examined.
  - **Dependency Type** — A brief description of the dependency relationship.
  - **Line** — The line in the source code.

  For more details, see “Dependency Table” on page 66.
- **Source Fragment** — Displays the source code of the selected dependency and includes a link to view or edit the source code in an external editor.

Options

- **Filter dependencies from/to contained elements** — Ignore all dependencies to contained elements of the selected element.
- **Filter imports** — Ignore import dependencies.
- **Used by** — Shows all dependencies *to* a chosen model element.
- **Uses from** — Shows all dependencies *from* a chosen element.
- **Source Fragment** — Displays the source code of the selected dependency and includes a link to view or edit the source code in an external editor.

Dependency Table

The dependency table includes the following columns to provide information about the dependencies listed in the Usage View:

- **From** — Source element.
- **To** — Target element.
- **Dependency Type** — Description of the dependency.
- **Line** — Line number in the source code where the dependency is defined.

Table 2-3. Dependencies in the Usage View

Dependency Type	Description
imports	Uses a type or package in an import statement.
extends	Indicates an inheritance relationship.
implements	Interface implementation relationship.
return type	Uses a type as a method return type.
parameter type	Uses a type in a method parameter.
throws	Declares a method exception.
method call	Calls a method.
field type	Uses a type as a class field.
field access	Uses a field.
type usage	Uses a type as a local variable.
instanceof	Uses a type in an `instanceof` check.
cast	Uses a type in a `cast` operation.
catch	Uses a type as an exception type in a `catch` clause.
new	Consists of a `new type` statement.
new array	Uses a type in an array creation.

Table 2-3. Dependencies in the Usage View (Continued)

Dependency Type	Description
uses in expression	Uses a type in an expression not yet covered by the other flavors.
generic type	Used for the generics introduced in JDK 5.0 (also called JDK 1.5).

Use

- To view or edit the source code of a selected dependency in an external editor, double-click a row or click the **Source Fragment** link.
- To export the table to a `.csv` file, right-click in the table and select **Export data as comma-separated file (CSV)**.

Metrics Window

The Metrics window shows the metrics for all the packages in the package tree.

The window contains tabs corresponding to the perspectives of the Design Validation Details diagram.

- Cycles
- Size
- Hubs
- Inheritance
- Stability
- J2EE
- Code Generation

When you select a Metrics tab, the diagram automatically changes to the corresponding view. Similarly, when you select an item within a tab, the diagram changes to display information for that item.

The Metrics window contains inline help for each tab. To show or hide the help text, click the small triangles in the bottom right corner.

Each tab except the **Stability** tab includes a list that you can use to filter the information displayed. You can sort the tables by clicking the appropriate column heading. A blue triangle indicates the column by which the table is sorted, and whether the sort is ascending or descending order.

Use

- See “Viewing Metrics” on page 28.

Menus

The following context-sensitive menus are available:

- Package Popup Menu
- Leaf Popup Menu
- Dependency Popup Menu
- Class Diagram Popup Menu

Package Popup Menu

The popup menu for a package element has the following items.

Table 2-4. Package Element Menu Items

Menu item	Function	Enabled
Flatten	Turn the **Flatten** option on or off. This option displays all child elements at the same hierarchical level as the parent.	Always
Move	Move to another package (as specified using the fully qualified element identifier). You can also drag and drop the package in the tree pane.	Always
Delete	Delete the package.	Only for empty packages
Rename	Rename the package.	Always
New Package	Create a new child package.	Always
Add To Scratch Area	Place the object in the scratch area.	Only when the element is not in the scratch area
Remove From Scratch Area	Remove the object from the scratch area.	Only when the element is in the scratch area
Dependencies	Add or remove design dependencies to packages on the same level.	Only available for Design Reference models
Metrics View	Open the Metrics window for this package.	Always
Usage View	Open the **Usage View** dialog box for this package.	Always
Help	Display the help topic for the package popup menu.	Always

Leaf Popup Menu

The popup menu for a leaf element has the following items.

Table 2-5. Leaf Menu Items

Menu item	Function	Enabled
View Source	Open the element in the external text editor.	Only for Java source code models
Metrics View	Open the Metrics window for this leaf.	Always
Usage View	Open the **Usage View** dialog box for this leaf.	Always
Help	Display the help topic for the leaf popup menu.	Always

Dependency Popup Menu

The popup menu for a dependency has the following items.

Table 2-6. Dependency Menu Items

Menu item	Function	Enabled
Dependency View	Open the **Dependency View** dialog box for this leaf.	Always for both Java code models and Design Reference models.
Delete	Delete the dependency.	Always for Design models. This item is not available for Java code models.
Remove Unused Import	Remove an unused import.	For Java code models only if the dependency consists of useless imports. This item is not available for Design Reference models.
Invert	Invert the dependency.	For Java code models only if the observer pattern can be applied to invert a dependency. This item is not available for Design Reference models.
Help	Display the help topic for the dependency popup menu.	Always.

Class Diagram Popup Menu

The class diagram popup menu is displayed when you click on the background in the class diagram pane of the Design Validation Details page.

Table 2-7. Class Diagram Menu Items

Item	Function
Classes collapsed	Turns the **Classes collapsed** option on or off. If selected, the classes and interfaces in a package are collapsed into a package. Collapsing classes can be useful to reduce the complexity of a diagram.
Dependency weights	Turns the display of dependencies weights on or off. The dependency weight is the number of references there are from one element to another.
Show Suggested Fixes	Turns the light bulbs for suggested fixes on or off. When shown, clicking a light bulb will generate a list of fixes to correct the undesirable dependency.
Animate Transitions	Turns the animation of diagram elements on or off.
Quality Principle	Changes the quality metric. The selected quality metric is used to color the nodes. Green is the best score, red is the worst, and yellow is in-between. Precise values for metrics are given by the Metrics window.
Font size	Changes the size of the font and the element. The value corresponds to the text font point size. Reducing the font size is useful if the class diagram crowded.
Flatten All Packages	Turns the **Flatten** option on for all packages. Consequently, when browsing the package structure with the **Classes collapsed** option cleared, classes from all subpackages are displayed at the same hierarchical level. With **Classes collapsed** selected, all subpackages are displayed at the same hierarchical level.
Unflatten All Packages	Unflattens all packages.
Print	Prints the diagram.
Save as PNG	Saves the diagram as a PNG image. The size of the image is the same as the window size. The PNG format is supported by most browsers and image tools.
Help	Displays the help topic for the class diagram popup menu.

Chapter 3

Tutorials

The following tutorials demonstrate various features of Code Review and how to work with Code Review to improve your code:

- Improving a Package Structure
- Improving Java Code by Applying Coding Rules
- Enhancing Javadoc with UML Diagrams

Sample files for use with these tutorials are included in the DevPartner Java Edition installation.

Improving a Package Structure

This tutorial demonstrates how you can improve a package structure.

Prerequisites

To execute the tutorial, you need a copy of the JMeter source files. A zipped copy of the JMeter source tree used for this example is included in the **`samples`** folder of theDevPartner Java Edition product folder. Before starting the tutorial, extract the contents of the zip file to a local folder.

This tutorial is based on the 1.4.1-dev version of the JMeter sources. It requires the Sun J2EE SDK version 1.2.1 **`j2ee.jar`** library (available from http://java.sun.com/j2ee/sdk_1.2.1/). To build the JMeter code model without compile errors, the classpath must include a reference to the **`j2ee.jar`** library.

Although working with an incomplete model is often possible, there may be occasions where the classpath must be defined to build the code model. When analyzing your own programs, therefore, it is recommended that you provide a complete classpath.

Duration

This tutorial takes approximately 40 minutes to complete.

Objectives

In this tutorial, you learn how to improve a package structure using Code Review, focusing particularly on the compliance of the package structure with the Acyclic Dependency Principle (ADP). The ADP is a well-known design principle, which states that there should be no cycles between packages. We use the JMeter open source project (project Web site at http://jakarta.apache.org/jmeter/) as a real-life case. In about 40 minutes, you will have made major improvements to the top-level architecture of JMeter.

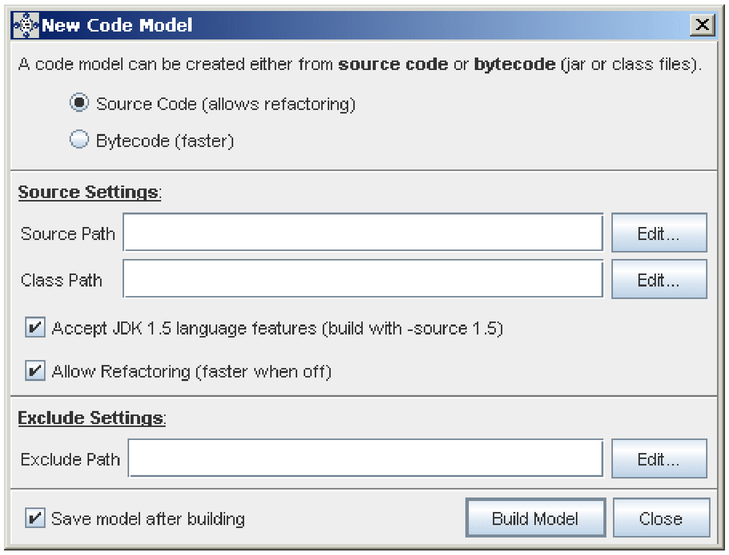
Steps

Note: The screens shown below are examples only. Your actual results may not exactly match the results shown in this tutorial.

Step 1 – Build the source model

In the first step, you make a new Java source model based on the source files of JMeter.

1 From the **File** menu, choose **New Code Model** to display the **New Code Model** dialog box.



New Code Model

A code model can be created either from **source code** or **bytecode** (jar or class files).

[✓ Source Code (allows refactoring)] Bytecode (faster)

Source Settings:

Source Path

Class Path

[✓] Accept JDK 1.5 language features (build with -source 1.5)

[✓] Allow Refactoring (faster when off)

Exclude Settings:

Exclude Path

[✓] Save model after building

2 Select the **Source Code (allows refactoring)** option.

3 In the **Source Path** box, click **Edit**, then navigate to and select the JMeter source root folder. the JMeter source root folder is the location where you extracted the JMeter zip file contents.

4 In the **Class Path** box, click **Edit**, then navigate to and select the `j2ee.jar` library file.

5 Whether you are editing a code model or refactoring source code, saving the code model to file is good practice for the following reasons:

- ◇ Allows a model to be loaded at the start of a session more quickly than rebuilding.
- ◇ Simplifies redistribution of the model amongst team members.
- ◇ Provides a snapshot of the model for later reference.

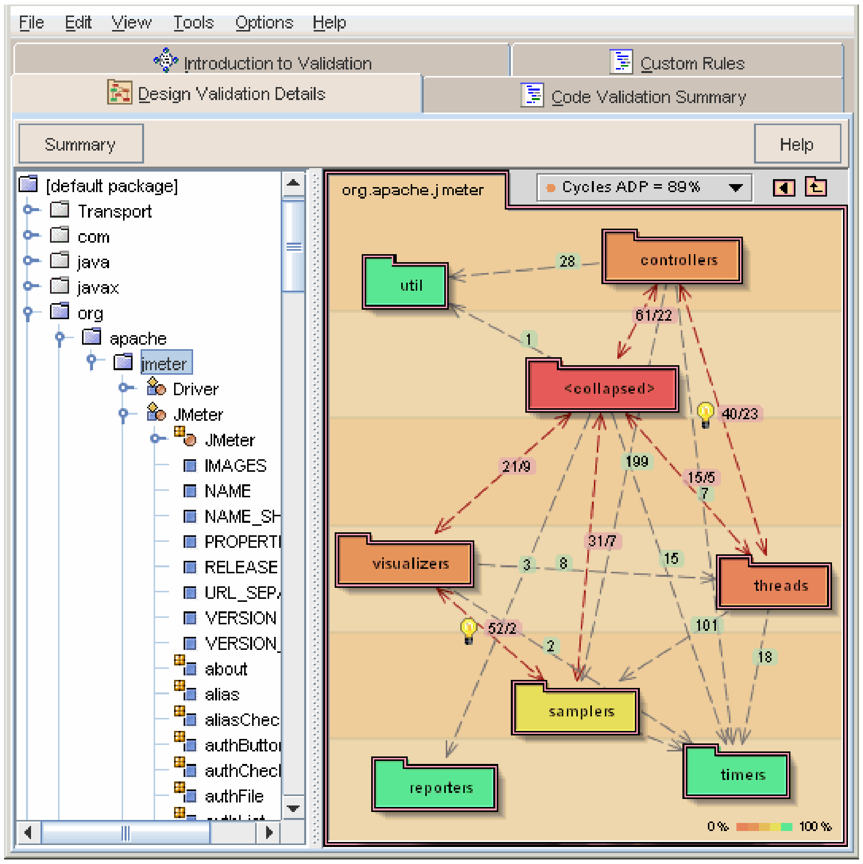
To save the code model automatically, select the **Save model after building** check box. Alternatively, the code model can be saved to file at any stage after the model has been built by choosing **File>Save** or **File>Save As**.

6 Click **Build Model** to build a code model. The progress of the build and error messages are reported in **Build Status**.

7 When the build is complete, if no compile errors were encountered, the dialog box will close automatically. Otherwise, click **Close** to close the New Code Model dialog box.

The results of the rules and design validation analysis are available in the Code Validation Summary and Design Validation Summary pages.

- Click the Design Validation Summary page, which provides an overview of the design violations, grouped into three categories. For more information, see "Design Validation" on page 14.
- To save a report of the records displayed in the Design Validation Summary page, click **Export** to display the **Export Metrics** dialog box.
- **Packages with Most Inter Package Cyclic Dependencies** allows the best initial assessment of the causes of design violations. Click the associated **org.apache.jmeter** link to open the Design Validation Details page, which displays the code model as a hierarchical tree of packages containing classes and a class diagram of the selected package.



Note: To display inter-package cyclic dependencies alone, Code Review "collapses" any top-level classes.

Step 2 – Save and Print the Design Validation Details

While working in the Design Validation Details page, changes can be saved or printed at any stage in a number of different ways:

- From the **File** menu, choose **Save** or **Save As** to save the latest version of your code model.

- To print a copy of the class diagram, right-click the background of the class diagram pane and from the popup menu, choose **Print.**
- Similarly, to save a copy of the class diagram to file, right-click the background of the class diagram pane and from the popup menu, choose **Save as PNG.**

Step 3 – Move top-level classes to subpackages

As shown in the previous figure, JMeter source does not comply with the ADP — it scores 88% on this metric. The red arrows show dependencies that, according to the analysis, should be removed. Code Review provides suggested fixes, indicated by the light bulb icons. When all the red dependencies have been removed, the resulting package structure is acyclic and the score is 100%.

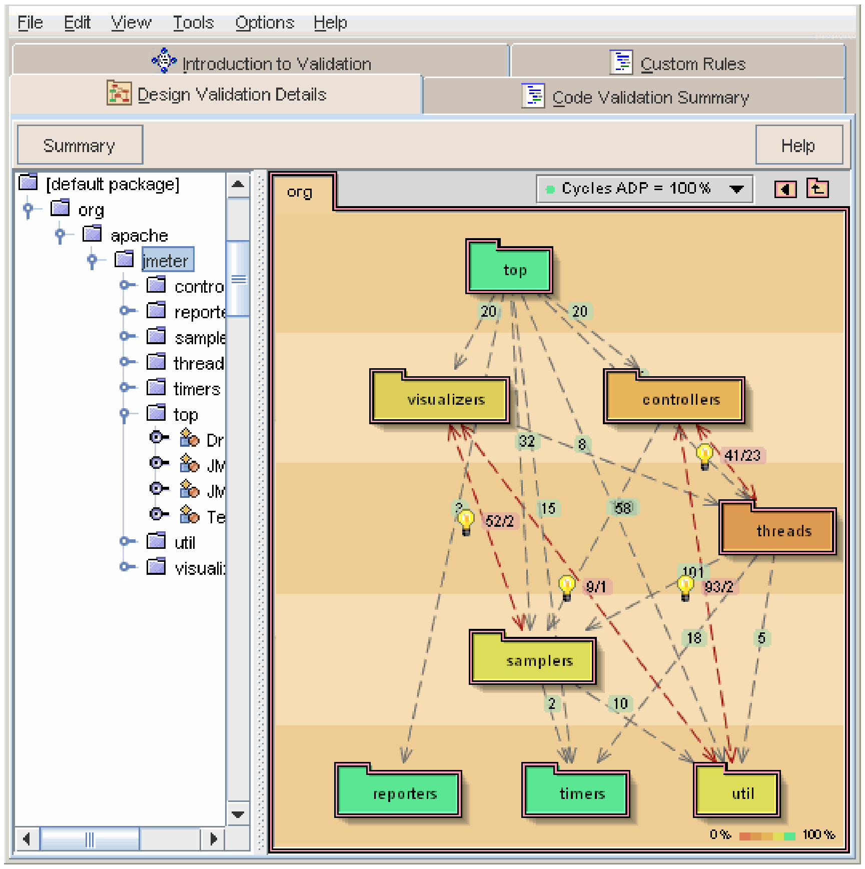
To make the right decisions as a software architect or developer, you should understand the functionality and requirements of the program. Furthermore, you need to have some idea about which parts of the programs are likely to be extended or reused. However, the package structure can be improved even without this knowledge.

One of the first things to look for is whether there are any classes at the top-level. A high occurrence of cyclic dependencies associated with the 'collapsed' top-level classes, as is the case for this example, is a good indication that top-level classes should be moved to appropriate subpackages.

As shown in the tree view, the **`jmeter`** package has five top-level classes. For improved package structure, it is good practice to move these classes to subpackages. However, care should be taken to choose suitable packages based on the layer where the top-level classes are located.

1. Right-click on the diagram background and clear the **Classes Collapsed** option in the context menu. Notice how the ADP score increases to 95% because cyclic dependencies associated with the “collapsed” classes are no longer recognized.
2. In the diagram, click the light bulb icon beside the red dependency between **`util`** and **`JMeterUtils`** to display the suggested solution.
3. Click the option **Move JMeterUtils to util**. The package tree and diagram are redrawn to reflect this change. Notice how **`util`** is relocated to the lowest layer. Notice also that the file name displayed in the title bar has been flagged with an asterisk to indicate a change to the model that has yet to be saved.
4. In the tree view, right-click the **`jmeter`** package, select **New Package** and name the new package **`top`**. Click **OK**.

5 In the tree view, drag and drop the classes **`Driver`**, **`JMeter`**, **`JMeterEngine`**, and **`TextDriver`** to the **`top`** package. The resulting package diagram is already much clearer than the original, even though this move did not directly improve the ADP.

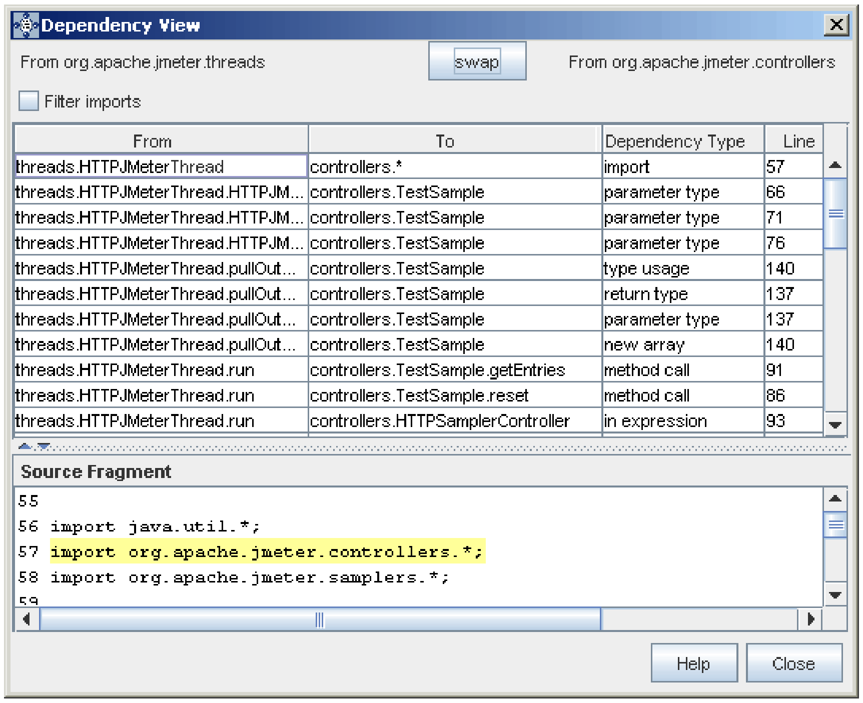


Step 4 – Fix an undesirable dependency by moving classes

1 In the diagram, click the light bulb icon beside the red dependency between **visualizers** and **samplers** to display the suggested solution.

2 Click the option **Move URLSampler to visualizers**. The package tree and diagram are redrawn to reflect this change.

3 Double-click the dependency between **threads** and **controllers** to display the Dependency View.


Dependency View
From org.apache.jmeter.threads
swap
From org.apache.jmeter.controllers
Filter imports
Source Fragment
Help
Close

From	To	Dependency Type	Line
threads.HTTPJMeterThread	controllers.*	import	57
threads.HTTPJMeterThread.HTTPJM...	controllers.TestSample	parameter type	66
threads.HTTPJMeterThread.HTTPJM...	controllers.TestSample	parameter type	71
threads.HTTPJMeterThread.HTTPJM...	controllers.TestSample	parameter type	76
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	type usage	140
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	return type	137
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	parameter type	137
threads.HTTPJMeterThread.pullOut...	controllers.TestSample	new array	140
threads.HTTPJMeterThread.run	controllers.TestSample.getEntries	method call	91
threads.HTTPJMeterThread.run	controllers.TestSample.reset	method call	86
threads.HTTPJMeterThread.run	controllers.HTTPSamplerController	in expression	93

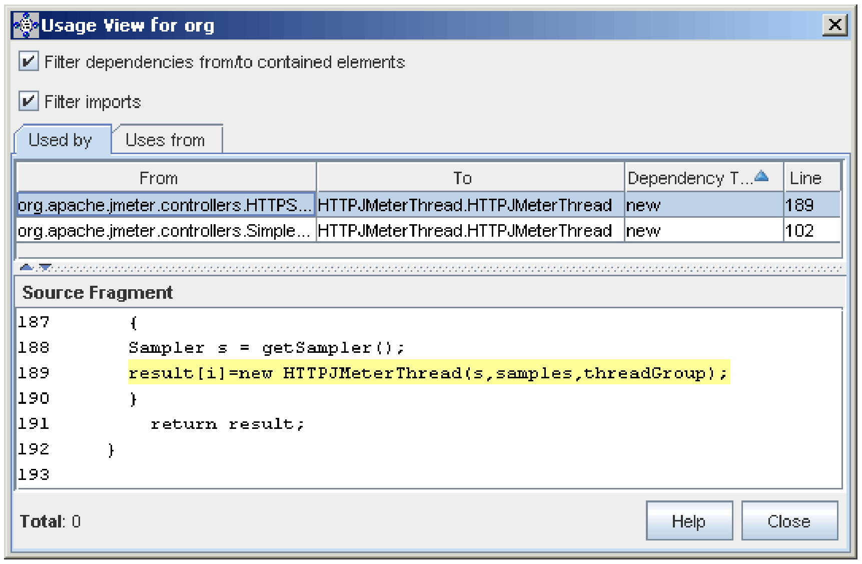
```
55
56 import java.util.*;
57 import org.apache.jmeter.controllers.*;
58 import org.apache.jmeter.samplers.*;
```

The **From** column shows that all dependencies originate from two classes in the threads package: **`HTTPJMeterThread`** and **`JMeterThreadAdapter`**. We now focus on the usage of these two classes.

4 Close the Dependency View and double-click the **threads** package.

5 Right-click **`JMeterThreadAdapter`** and choose **Usage View** from the popup menu. You can see that **`JMeterThreadAdapter`** is only used by **`HTTPJMeterThread`**.

6 Close the Usage View for **`JMeterThreadAdapter`** and open it for **`HTTPJMeter-Thread`**. This shows that **`HTTPJMeterThread`** is only used by classes from the **`con-trollers`** package.



From	To	Dependency T...	Line
org.apache.jmeter.controllers.HTTPS...	HTTPJMeterThread.HTTPJMeterThread	new	189
org.apache.jmeter.controllers.Simple...	HTTPJMeterThread.HTTPJMeterThread	new	102

```
187         {
188         Sampler s = getSampler();
189         result[i]=new HTTPJMeterThread(s,samples,threadGroup);
190         }
191           return result;
192       }
193
```

You can therefore move these two classes from the **threads** package to the **controllers** package.

7 Close the Usage View.

8 In the tree view, drag **HTTPJMeterThread** and **JMeterThreadAdapter** from the **threads** package to the **controllers** package.

9 Navigate to the **jmeter** package and notice that the undesirable dependency is completely removed and that the ADP score is 99%.

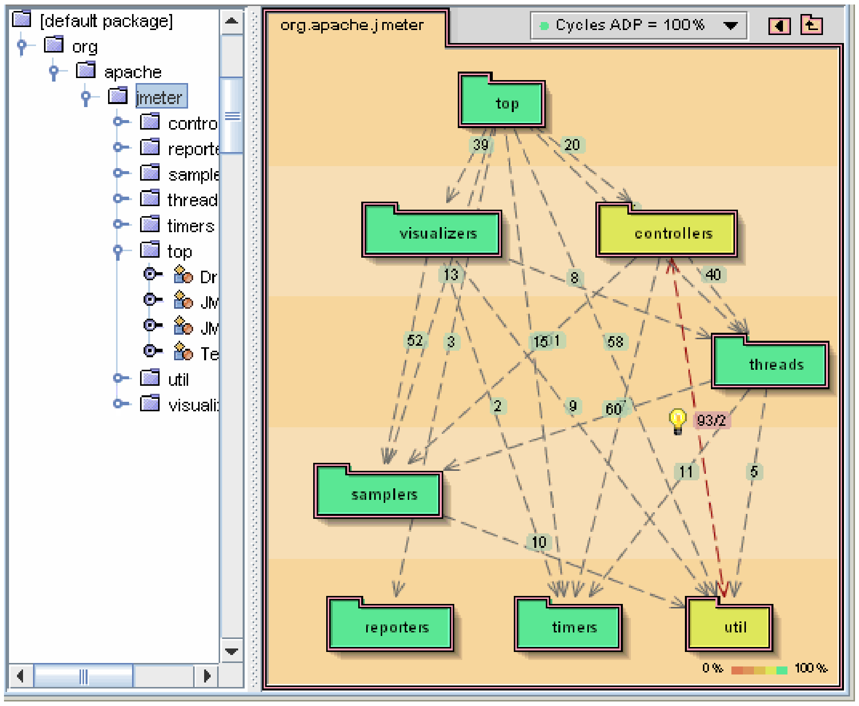
10 For a more precise score, choose **View>Metrics** and click the **Cycles** tab. The exact ADP score for **org.apache.jmeter** is displayed in the **ADP** column.

Step 5 – Fix an undesirable dependency by removing an import

There are two undesirable dependencies left.

1 Click the light bulb icon beside the dependency between **util** and **visualizers**. It suggests that you **Remove Unused Import**. To confirm the suggested fix, double-click the dependency to display the Dependency View, and select **Filter imports**. The import dependency is listed in the table, which indicates that the **visualizers** package is imported but not used in the **util** package.

2 Return to the light bulb icon and delete the dependency by clicking **Remove Unused Import**. The undesirable dependency is removed and there is only one red arrow left in the diagram.

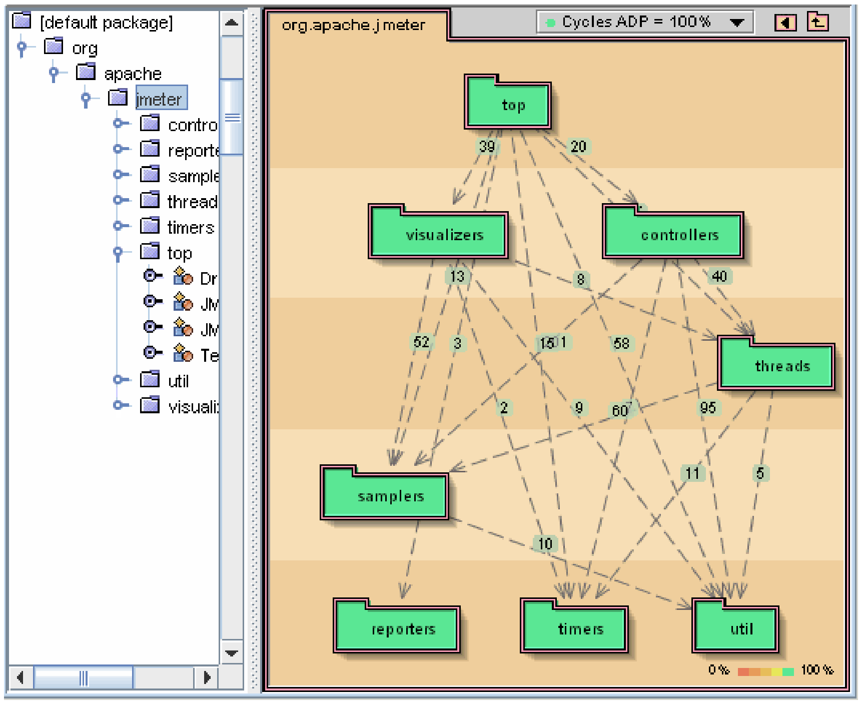


Step 6 – Invert dependencies

There is only one undesirable dependency left and the weight of this dependency is 2.

1 To examine the remaining cyclic dependency (between **`util`** and **`controllers`**) in more detail, double-click the dependency to display the Dependency View. This shows that the dependency is between the classes **`JMeterUtils`** and **`SamplerController`**. It consists of a cast to **`SamplerController`** in the code of **`JMeterUtils`** and a related method call to **`setPropName`**. One way to fix this dependency is to use an interface to reverse the dependency. For more information, see "Dependency Inversion Refactoring" on page 29.

2 To fix this dependency, click the light bulb icon for the undesirable dependency and choose **Invert with listener(s)**.

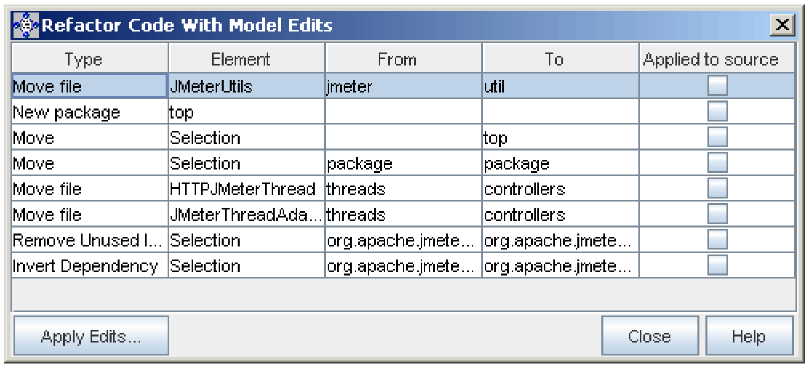


Step 7 – Apply the changes to the code

So far, we have made changes to the model of the code but the code itself has not been changed. Any change made to the model can also be applied to the code.

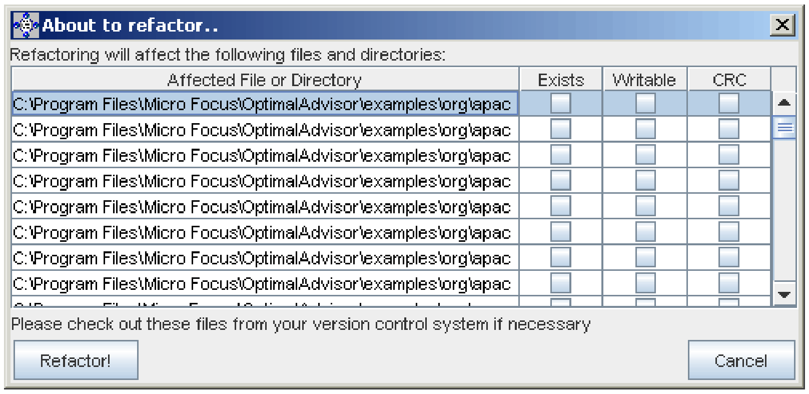
1 Prior to refactoring code, it is highly recommended to make a backup of your source files because you cannot choose to undo the changes.

2 From the **Tools** menu, choose **Refactor Code with Model Edits**. The **Refactor Code With Model Edits** dialog box opens, showing a table with all the changes listed.



Type	Element	From	To	Applied to source
Move file	JMeterUtils	jmeter	util	☐
New package	top			☐
Move	Selection		top	☐
Move	Selection	package	package	☐
Move file	HTTPJMeterThread	threads	controllers	☐
Move file	JMeterThreadAda...	threads	controllers	☐
Remove Unused I...	Selection	org.apache.jmete...	org.apache.jmete...	☐
Invert Dependency	Selection	org.apache.jmete...	org.apache.jmete...	☐

3 Click **Apply Edits** to display the **About to refactor** dialog box, which lists all the files that need to be changed to apply this refactoring. The tool checks whether all the files exist, whether they are writable and whether they have been edited with another program. For more information, see "Synchronizing the Source Code with the Code Model" on page 32.



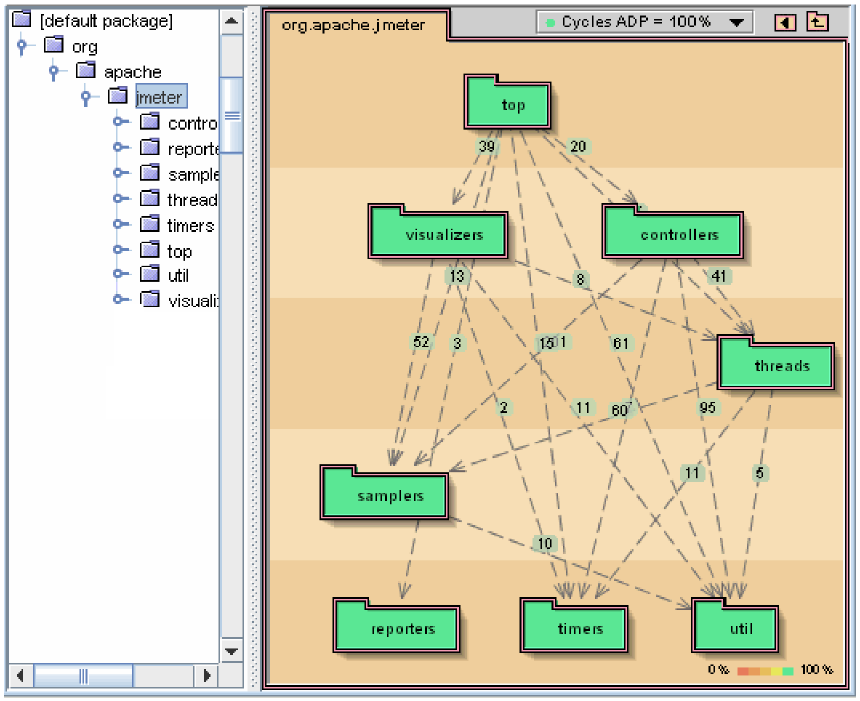
Refactoring will affect the following files and directories:

Affected File or Directory	Exists	Writable	CRC
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐
C:\Program Files\Micro Focus\OptimalAdvisor\examples\org\apac	☐	☐	☐

Please check out these files from your version control system if necessary

4 Click **Refactor!**. When refactoring is complete, the **About to refactor** dialog box closes and the **Refactor Code With Model Edits** dialog box reappears with all the **Applied to Source** options selected.

Note that slight changes in the number of dependencies may be apparent compared with the diagram prior to updating the model from code. These are to be expected and are due to the manner in which code edits are applied.



5 You have significantly cleaned up the architecture. The JMeter package structure now conforms to the Acyclic Dependencies Principle (ADP).

Code Review can indicate problems in Java package structures and help you to improve them.

In the case of JMeter, the structure could be fixed easily in a few steps. However, it is not always possible to repair a package structure by just moving classes and packages. Sometimes you can make additional fixes through the use of the interfaces but often the structure is so polluted that you need to thoroughly rewrite the code.

If you use Code Review from the beginning of the development process, you can identify and repair problems immediately and with little effort. This results in a clean package structure, improving comprehensibility and product flexibility, and cutting development time.

To assist the development process, the majority of Code Review interface features for analysis, refactoring, and reporting can be implemented at the command line. There are a variety of circumstances where the command line interface is particularly useful. For instance, an automated analysis could be incorporated into the build process of your product and the results included with the build report. For more information, see "Command Line Interface" on page 35.

Improving Java Code by Applying Coding Rules

This tutorial demonstrates how you can use the code validation component of Code Review to improve the quality of source code.

Prerequisites

To execute the tutorial, you need a copy of the JMeter source files. A zipped copy of the JMeter source tree used for this example is included with the DevPartner Java Edition installation, in the **`samples`** folder. Before starting the tutorial, unzip the file to a local folder.

This tutorial is based on the 1.4.1-dev version of the JMeter sources. It requires the Sun J2EE SDK version 1.2.1 **`j2ee.jar`** library (available from http://java.sun.com/j2ee/sdk_1.2.1/). To build the JMeter code model without compile errors, the classpath must include a reference to the **`j2ee.jar`** library.

Although working with an incomplete model is often possible, there may be occasions where the classpath must be defined to build the code model. When analyzing your own programs, therefore, it is recommended that you provide a complete classpath.

Duration

This tutorial takes approximately 30 minutes to complete.

Objectives

In this tutorial, you will build a code model from source code, evaluate the code violation reports and implement fixes according to best Java coding practices.

This tutorial demonstrates the function of the code validation components of Code Review and illustrates the selection of code rules. It shows how to do the following:

- Work with reporting tools.
- Interpret the results of the analysis.
- Edit source code and update the code model.

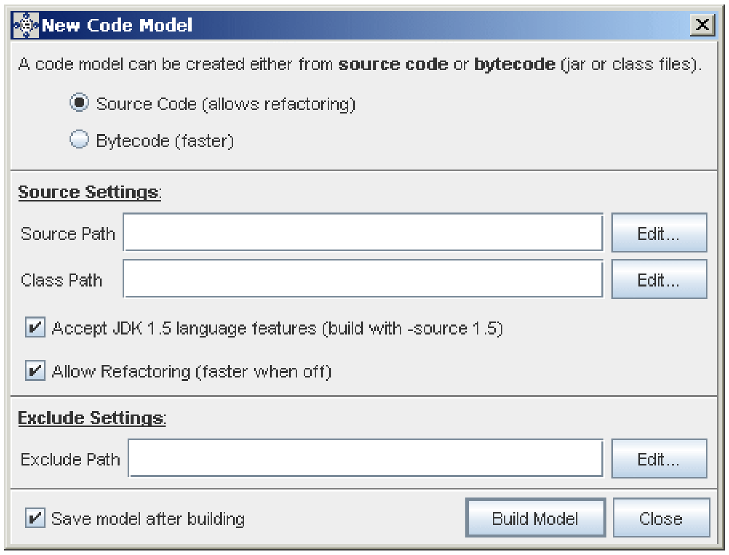
Steps

Note: The screens shown below are examples only. Your actual results may not exactly match the results shown in this tutorial.

Step 1 – Build the source model

In the first step, you make a new Java source model from the source files of JMeter.

1 From the **File** menu, choose **New Code Model** to display the **New Code Model** dialog box.



New Code Model

A code model can be created either from **source code** or **bytecode** (jar or class files).

[● Source Code (allows refactoring)]

○ Bytecode (faster)

Source Settings:

Source Path

Class Path

☑ Accept JDK 1.5 language features (build with -source 1.5)

☑ Allow Refactoring (faster when off)

Exclude Settings:

Exclude Path

☑ Save model after building

2 Select the **Source Code (allows refactoring)** option and enter the JMeter source root folder.

3 Whether you are editing a code model or refactoring source code, saving the code model to file is good practice for several reasons, including the following:

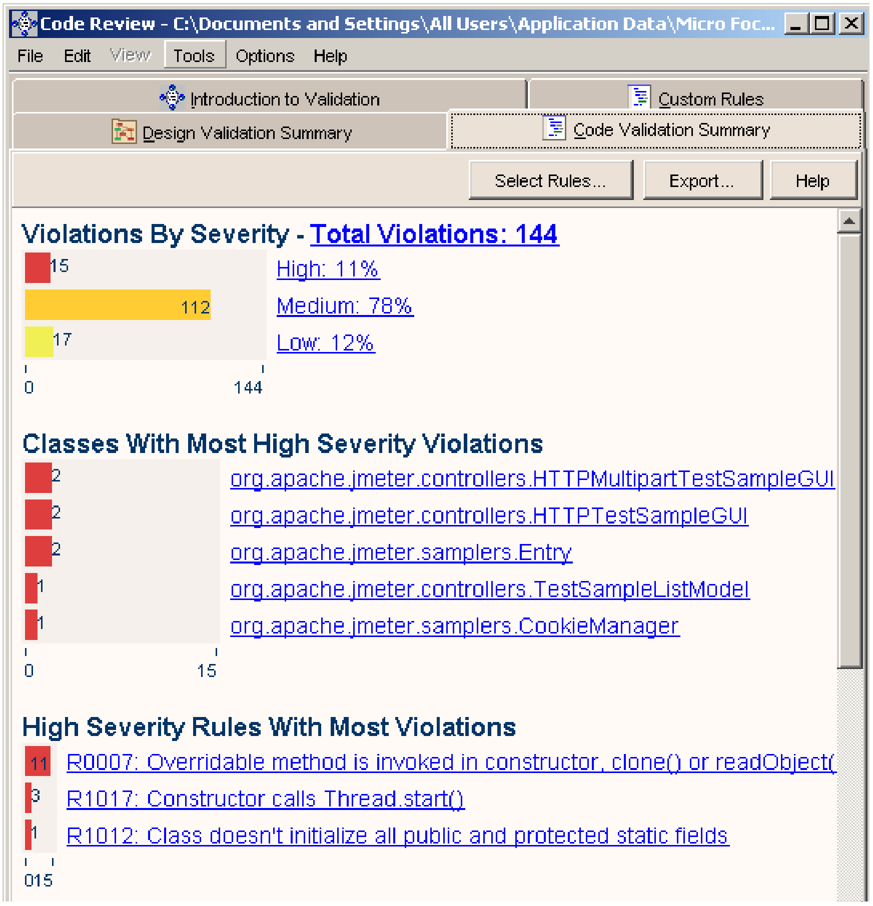
- It enables a model to be loaded at the start of a session more quickly than rebuilding.
- It simplifies redistribution of the model amongst team members.
- It provides a snapshot of the model for later reference.

To save the code model automatically, select **Save model after building**. Alternatively, the code model can be saved to file at any stage after the model has been built by choosing **File>Save** or **File>Save As**.

4 Click **Build Model** to build a code model. The progress of the build and error messages are reported in **Build Status**. In this tutorial, building JMeter with errors will have no affect on the results.

5 When the build is complete, if no compile errors were encountered, the dialog box closes automatically. Otherwise, click **Close** to close the **New Code Model** dialog box. The results of the rules and design validation analysis are displayed in the Code Validation Summary page and Design Validation Summary page, respectively.

6 Select the Code Validation Summary page, which provides an overview of the code violations, grouped into three categories. For more information, see "Code Validation" on page 10.


Code Review - C:\Documents and Settings\All Users\Application Data\Micro Foc...
File
Edit
View
Tools
Options
Help
Introduction to Validation
Custom Rules
Design Validation Summary
Code Validation Summary
Select Rules...
Export...
Help

Violations By Severity - Total Violations: 144

15	High: 11%
112	Medium: 78%
17	Low: 12%

0 144

Classes With Most High Severity Violations

2	org.apache.jmeter.controllers.HTTPMultipartTestSampleGUI
2	org.apache.jmeter.controllers.HTTPTestSampleGUI
2	org.apache.jmeter.samplers.Entry
1	org.apache.jmeter.controllers.TestSampleListModel
1	org.apache.jmeter.samplers.CookieManager

0 15

High Severity Rules With Most Violations

11	R0007: Overridable method is invoked in constructor, clone() or readObject(
3	R1017: Constructor calls Thread.start()
1	R1012: Class doesn't initialize all public and protected static fields

0 15

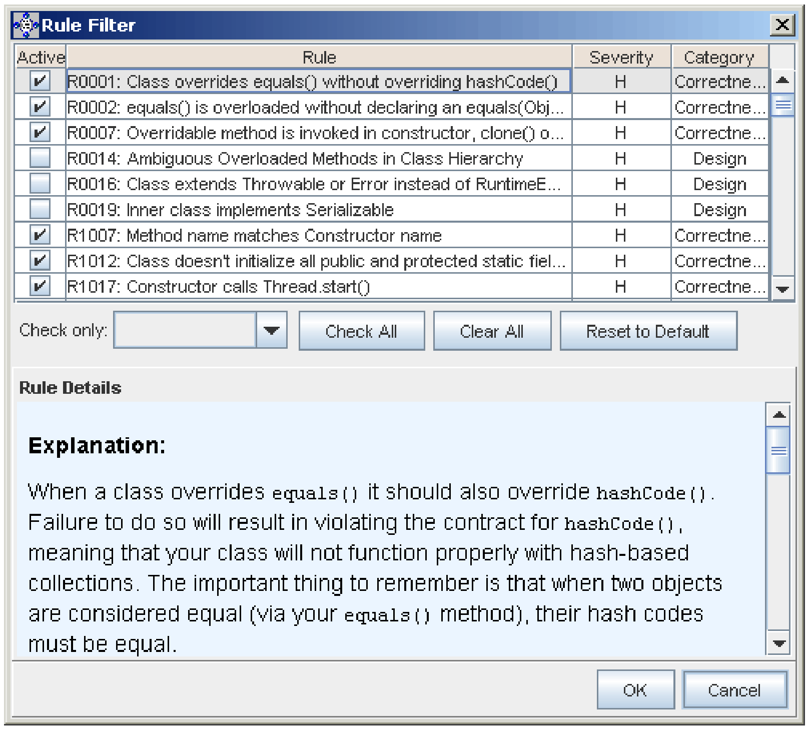
Step 2 – Select the Ruleset

The results shown on the Code Validation Summary page and Code Validation Details page depend on the active ruleset. A *ruleset* is the list of rules that will be used to perform code validation. Any rule that is not appropriate to your situation can be suppressed. The ruleset can be chosen either from predefined categories or customized according to specific requirements. As a general guide, high severity rule violations will identify parts of code most in need of attention:

1 In the Code Validation Summary page, click **Select Rules** to display the **Rule Filter** dialog box.

2 A table shows a list of all rules that were violated, with short descriptions, category, and severity. A detailed description for the selected rule is shown in the **Rule Details** pane, which also includes suggested remedies and links to reference material. The **Active** option defines whether or not a rule is enabled.

3 From the **Check only** list, select **Correctness** to activate all rules in the correctness category.



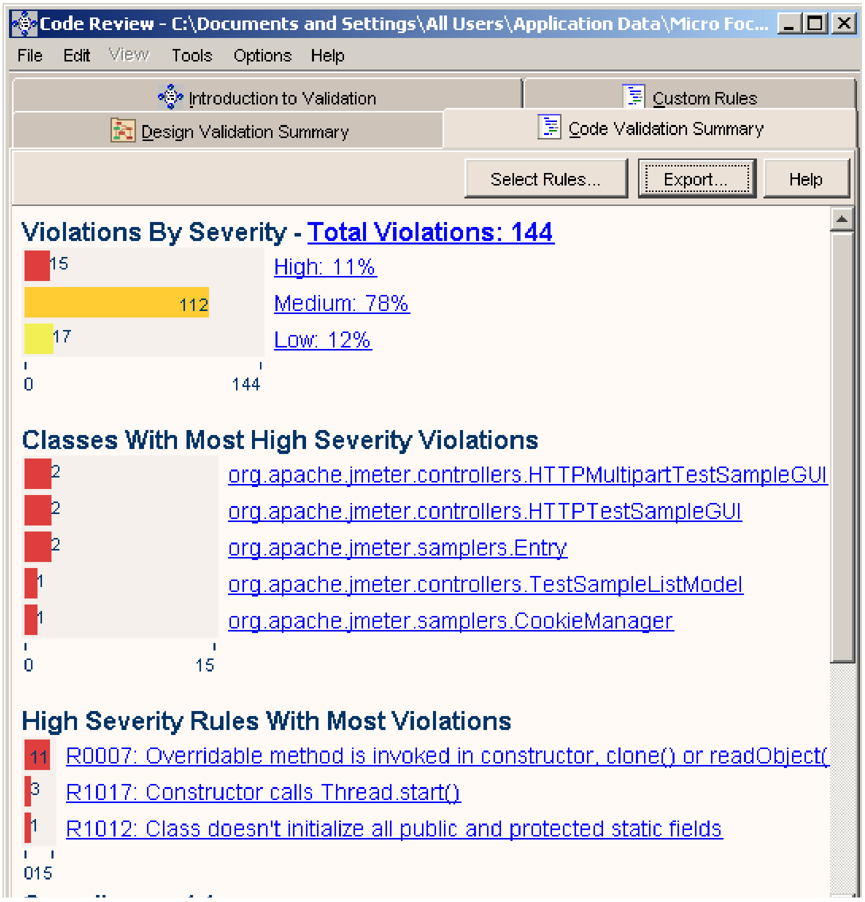
Active	Rule	Severity	Category
✓	R0001: Class overrides equals() without overriding hashCode()	H	Correctne...
✓	R0002: equals() is overloaded without declaring an equals(Obj...	H	Correctne...
✓	R0007: Overridable method is invoked in constructor, clone() o...	H	Correctne...
	R0014: Ambiguous Overloaded Methods in Class Hierarchy	H	Design
	R0016: Class extends Throwable or Error instead of RuntimeE...	H	Design
	R0019: Inner class implements Serializable	H	Design
✓	R1007: Method name matches Constructor name	H	Correctne...
✓	R1012: Class doesn't initialize all public and protected static fiel...	H	Correctne...
✓	R1017: Constructor calls Thread.start()	H	Correctne...

Explanation:

When a class overrides `equals()` it should also override `hashCode()`. Failure to do so will result in violating the contract for `hashCode()`, meaning that your class will not function properly with hash-based collections. The important thing to remember is that when two objects are considered equal (via your `equals()` method), their hash codes must be equal.

Note: To define a custom ruleset, select the item from the **Check only** list that most closely matches your requirements and select or clear **Active** options as necessary.

4 Click **OK** to return to the Code Validation Summary page, where the code violation results will be updated for an analysis based upon the correctness ruleset.


Code Review - C:\Documents and Settings\All Users\Application Data\Micro Foc...
File Edit View Tools Options Help
Introduction to Validation
Custom Rules
Design Validation Summary
Code Validation Summary
Select Rules...
Export...
Help

Violations By Severity - Total Violations: 144

15	High: 11%
112	Medium: 78%
17	Low: 12%

0 144

Classes With Most High Severity Violations

2	org.apache.jmeter.controllers.HTTPMultipartTestSampleGUI
2	org.apache.jmeter.controllers.HTTPTestSampleGUI
2	org.apache.jmeter.samplers.Entry
1	org.apache.jmeter.controllers.TestSampleListModel
1	org.apache.jmeter.samplers.CookieManager

0 15

High Severity Rules With Most Violations

11	R0007: Overridable method is invoked in constructor, clone() or readObject(
3	R1017: Constructor calls Thread.start()
1	R1012: Class doesn't initialize all public and protected static fields

015

Note: The active ruleset is persistent through sessions until changed.

Step 3 – Export Rule Violation Reports

From the Code Validation Summary page, reports can be saved or printed in a number of different ways:

1 To save a report of the records displayed in the Code Validation Summary, click **Export** to display the **Export Violations** dialog box.

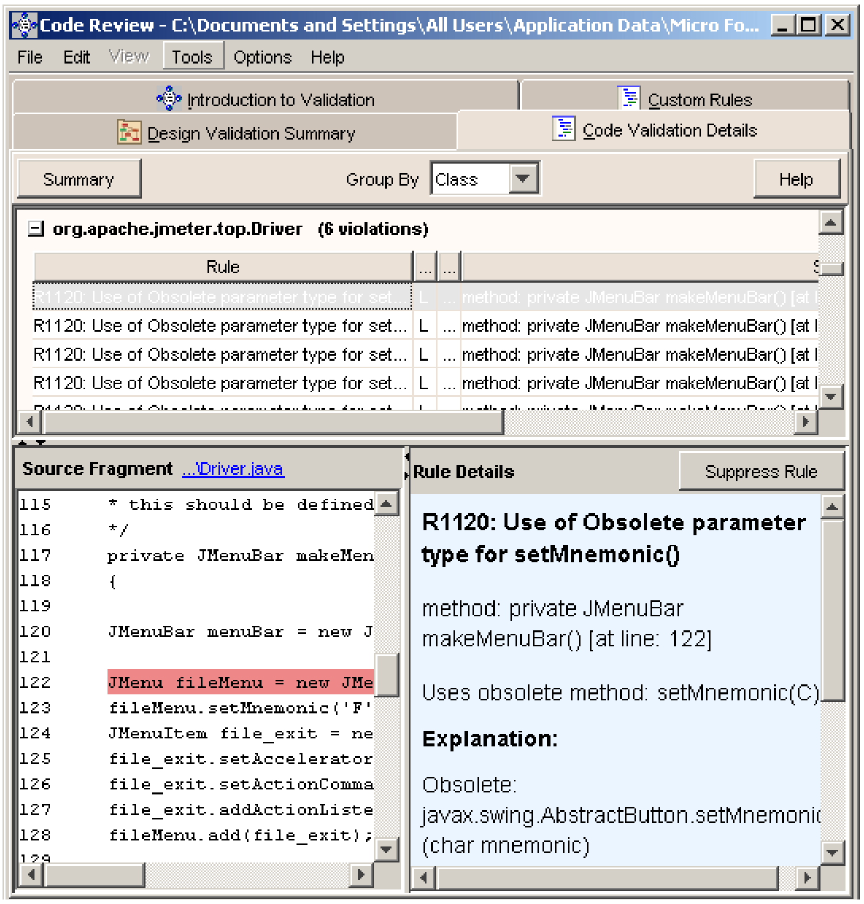
2 Select **Summary to HTML**, specify the location and file name, and click **Save**. After the report is saved, the dialog box closes automatically.

3 Start your Web browser and open the HTML report. The report is identical to the information shown on the Code Validation Summary page. Leave the HTML report open and return to Code Review.

Step 4 – Correct the Rule Violations

The analysis of the JMeter code based on the correctness ruleset yields a considerable number of violations. Here, we shall correct the high severity violations reported for the `org.apache.jmeter.Driver` class:

1 On the Code Validation Summary page, click the **org.apache.jmeter.Driver** link to view the Code Validation Details page.


Code Review - C:\Documents and Settings\All Users\Application Data\Micro Fo...
File Edit View Tools Options Help
Introduction to Validation
Custom Rules
Design Validation Summary
Code Validation Details
Summary
Group By Class
Help
Source Fragment ...\Driver.java
Rule Details
Suppress Rule

org.apache.jmeter.top.Driver (6 violations)

Rule	...	...	S
R1120: Use of Obsolete parameter type for set...	L	...	method: private JMenuBar makeMenuBar() [at l
R1120: Use of Obsolete parameter type for set...	L	...	method: private JMenuBar makeMenuBar() [at l
R1120: Use of Obsolete parameter type for set...	L	...	method: private JMenuBar makeMenuBar() [at l
R1120: Use of Obsolete parameter type for set...	L	...	method: private JMenuBar makeMenuBar() [at l

```
115      * this should be defined
116      */
117      private JMenuBar makeMen
118      {
119
120      JMenuBar menuBar = new J
121
122      JMenu fileMenu = new JMe
123      fileMenu.setMnemonic('F'
124      JMenuItem file_exit = ne
125      file_exit.setAccelerator
126      file_exit.setActionComma
127      file_exit.addActionListe
128      fileMenu.add(file_exit);
```

R1120: Use of Obsolete parameter type for setMnemonic()

method: private JMenuBar makeMenuBar() [at line: 122]

Uses obsolete method: setMnemonic(C)

Explanation:

Obsolete: javax.swing.AbstractButton.setMnemonic (char mnemonic)

2 As shown in the figure, there are two reported occurrences of the high severity violation "Overridable method is invoked in constructor, clone() or readObject()".

3 Examine the code in the **Source Fragment** pane. In this case, **`Driver()`** was probably not meant to be subclassed. To correct the problem, **`init()`** and **`initGUI()`** should be changed to **`final`**.

4 To display the source code in an external editor, do one of the following:

- ◇ Double-click the item in the table.
- ◇ Click the **Source Fragment** link.
- ◇ Double-click inside the **Source Fragment** pane.

5 Edit **`Driver.java`** for corrections to **`init()`** and **`initGUI()`** and save the changes.

Note: To configure Code Review for an external editor, see "Configuring an External Editor" on page 34.

6 Although changes to source code are saved, the code model is not updated automatically. To refresh the model for code edits and to display an updated summary of rule violations, it is necessary to synchronize the code model with source code edits.

Step 5 – Synchronize the Code Model with Source Code Edits

To synchronize the code model with the source code it is necessary to update the model from the code:

1. From the **Tools** menu, choose **Update Model from Code Edits** and, if necessary, save the code model when prompted.
2. The code model is rebuilt, with the progress shown in a status bar.
3. After the code model is rebuilt, the Code Validation Summary page displays the rule violations updated for the corrections made to **`org.apache.jmeter.Driver`**.
4. A comparison of the Code Validation Summary page with the earlier HTML report verifies the changed summary report.

This tutorial has demonstrated how to produce and analyze code violation reports in anticipation of making improvements to source code. We have seen that Code Review rule descriptions include details of possible remedies to individual violations, which guide the developer in their choice of solution. Edits are always performed manually, using an external editor. Consequently, the final stage in correcting for code violations should always be to synchronize the code model with the code edits. If the code model is saved to file before exiting Code Review, the model can be redistributed amongst members of the development team and loaded at the start of the next Code Review session more quickly than rebuilding the model.

Further Reading

Effective Java Programming Language Guide, written by Joshua Bloch and published by Addison-Wesley (ISBN 0201310058), provides detailed information for many coding rules. See http://java.sun.com/docs/books/effective/ for details and sample chapters.

Enhancing Javadoc with UML Diagrams

Prerequisites

For this tutorial you need a copy of JUnit. Download the zipped file from http://prdownloads.sourceforge.net/junit/ and extract the archive to your local disk.

You also need the JUnit source code. Download the source **`.jar`** file and extract the archive into a folder, **`source`**, under the folder to which you downloaded JUnit.

Duration

This tutorial takes approximately 20 minutes to complete.

Objectives

This tutorial demonstrates how to enhance Javadoc by incorporating UML class diagrams into the Javadoc package descriptions. In this tutorial, you will enhance the Javadoc for JUnit (project Web site at http://www.junit.org/index.htm).

Steps

Step 1 – Write the script

To generate Javadoc enhanced with UML class diagrams, you need to start Code Review via the command line interface.

1 Create a script file with the following two lines:

```
newjava <JUNIT>\source
```

```
uml2javadoc <JUNIT>\javadoc
```

where `<JUNIT>` is the fully qualified path to the folder to which you extracted the file. If the path contains spaces, enclose the argument between quotation marks (`" "`). This applies whenever a path is used as an argument, irrespective of whether from the command line or within a script.

2 Save the script as **`myscript.txt`**.

Step 2 – Start Code Review from the command line

Now start Code Review from the command line:

```
1 | java -DADVISOR_HOME="%ADVISOR_HOME%" -Djava.library.
| path="%PATH%" -Xmx500m -cp
| "%JAVA_HOME%\lib\tools.jar";"%ADVISOR_HOME%\lib\DLM40JNI.
| jar";"%ADVISOR_HOME%\bin\Code Review.jar" com.compuware.
| advisor.application.commandLine.CommandLine script
| <SCRIPTPATH>\myscript.txt
```

where `<SCRIPTPATH>` is the fully qualified path to **`myscript.txt`**. If any spaces occur in the path to the script file, it is necessary to enclose the argument between double quotation marks. This applies whenever a path is used as an argument, irrespective of whether at the command line or within a script. The environment variables `JAVA_HOME` and `ADVISOR_HOME` are set to the home folder of your JDK installation and the home folder of your DevPartner Java Edition installation, respectively.

The Javadoc is now enhanced with UML class diagrams.

Glossary

ADP: The Acyclic Dependency Principle is an object-oriented design principle: “The dependencies between packages must form no cycles”. See *Design Principles and Design Patterns* by Robert C. Martin, at http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF.

ADP metric: The Acyclic Dependency Principle metric measures the amount of work necessary to fix a package structure that violates the ADP. For a more detailed description of the metric, see “Quality Metrics” on page 24. The metric can be viewed in the Metrics window.

code model: A model of a Java program that is used in Code Review to provide metrics and views. Java source models can be built from either source code or bytecode, and stored to disk. Java source models can be verified against a design model.

collapsed: A collapsed box in a diagram represents the set of classes in the package shown. Classes can be collapsed into a single box, to allow for a simplified view. The collapse classes option can be switched on or off in the diagram popup menu.

dependency: A *class dependency* is a reference from the definition of a class to another class.

- A *file dependency* is a reference from one file to a class defined in the other file. It can be a class dependency or an import statement.
- A *package dependency* is a reference from a file in one package to a file in another package, or an * import of a package.

dependency weight: The dependency weight between two elements (packages or classes) is the amount of references from one to the other. Dependency weights can optionally be shown via the diagram popup menu.

design reference model: A UML design consisting of packages and their dependencies. Design reference models can be defined in Code Review, and used to verify source models.

layering: The layering of the application determines the layout of the UML diagrams.

quality metrics: A set of metrics that indicate the quality of the Java code. Quality metrics can be shown in the Metrics window or as color highlighting in the Diagram View.

refactoring: A technique to restructure code in a disciplined way (as defined by Martin Fowler). Code Review supports a number of refactorings.

stability metrics: A set of metrics defined by Robert C. Martin to measure the quality of an object-oriented design. The metrics can be viewed in the Metrics window. See http://www.objectmentor.com.