Code Smell Detection in Eclipse

Stefan Slinger

Thesis Report
March 2005

Thesis Supervisor
Dr. Ing. L.M.F. Moonen

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Software Technology
Software Engineering Group
Software Evolution Research Lab

Code Smell Detection in Eclipse

Stefan Slinger

Thesis Report
March 2005

Thesis Supervisor
Dr. Ing. L.M.F. Moonen

Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Department of Software Technology
Software Engineering Group
Software Evolution Research Lab

Thesis Presentation

Name:	Stefan Slinger
Title:	Code Smell Detection in Eclipse
Supervisor:	Dr. Ing. L.M.F. Moonen
Faculty:	Electrical Engineering, Mathematics and Computer Science
Department:	Software Technology
Group:	Software Engineering
Lab:	Software Evolution Research Lab

Thesis committee:
Prof. Dr. A. van Deursen
Dr. Ing. L.M.F. Moonen
Ir. B.R. Sodoyer

Abstract:
Refactoring is a software engineering technique that, by applying a series of small behavior-preserving transformations, can improve a software system's design, readability and extensibility. Code smells are signs that indicate that source code might need refactoring.
The goal of this thesis project was to develop a prototype of a code smell detection plug-in for the Eclipse IDE framework. In earlier research by Van Emden and Moonen, a tool was developed to detect code smells in Java source code and visualize them in graph views. CodeNose, the plug-in prototype created in this thesis project, presents code smells in the Tasks View in Eclipse, similar to the way compiler errors and warnings are presented. These code smell reports provide feedback about the quality of a software system.
CodeNose uses the Eclipse JDT parser to build abstract syntax trees that represent the source code. A tree visitor detects primitive code smells and collects derived smell aspects, which are written to a fact database and passed to a relational algebra calculator, the Grok tool. The results of the calculations on these facts can be used to infer more complex code smells.
In a case study, the plug-in was tested by performing the code smell detection process on an existing software system. We present the case study results, focusing at performance of the plug-in and usefulness of the code smells that were detected.

Preface

This report is the final part of the thesis project of my Computer Science study at the Delft University of Technology. The goal of this project was to build a code smell detection plug-in for the Eclipse framework. Both the research for the thesis project and the actual project were performed at the Software Evolution Research Lab (SWERL) of the Software Engineering Group in Delft. This thesis project was an extension of earlier research performed at CWI in Amsterdam by Eva Van Emden and Leon Moonen.
The Eclipse plug-in developed in this thesis project, which was named CodeNose, provides feedback about a software system's quality to software developers. Automatic detection of code smells is helpful to a programmer for deciding when and where to refactor his or her source code. This thesis report describes the project itself, discusses the concepts involved, describes in detail the software that was used and developed during this project, and presents the results of a case study that was performed using the plug-in developed.

Stefan Slinger
Badhoevedorp, March 2005

Summary

Refactoring is a software engineering technique that is used for restructuring existing source code, altering its internal structure without changing its external behavior. By applying a series of small behavior-preserving transformations, a software system's design, readability and extensibility can be improved. Code smells are signs that indicate that source code might need refactoring.
The goal of this thesis project was to develop a prototype of a code smell detection plug-in for the Eclipse IDE framework. In earlier research by Van Emden and Moonen, a tool was developed to detect code smells in Java source code and visualize them in graph views. CodeNose, the plug-in prototype that was created in this thesis project, presents code smells in the Tasks View in Eclipse, similar to the way compiler errors and warnings are presented. These code smell reports provide feedback about the quality of a software system.
We present an overview of books, articles and software that are related to this project. We discuss the concept of code smells and describe the Eclipse framework and the Eclipse features that are used by the plug-in prototype. The code smell detection process is described in full detail. The steps necessary to add a code smell to the detection process is illustrated by an example. CodeNose uses the Eclipse JDT parser to build abstract syntax trees that represent the source code. A tree visitor travels through these abstract syntax trees, detecting primitive code smells and collecting derived smell aspects, which are written to a fact database and passed to a relational algebra calculator, the Grok tool. The results of the calculations on these facts can be used to infer more complex code smells.
The code smells that were implemented in this prototype are Switch Statement, Long Method, Long Parameter List, Message Chain, Empty Catch Clause, Refused Bequest, Feature Envy and some class size code smells. Parameters for the code smell detection process can be set using Eclipse's preference pages. In a case study, the plug-in was tested by performing the code smell detection process on an existing software system. We present the case study results, focusing at performance of the plug-in and value of the code smells that were detected.
Most of the goals that were set out for the development of the code smell

plug-in prototype were met. The use of preference pages to set parameters makes adding a new smell somewhat harder than we would have liked, because the preference pages for these new smells have to be integrated within the existing page hierarchy. The performance of the plug-in could be enhanced by changing the way the plug-in operates, from the current on-call basis to constant monitoring of projects. For this prototype to become a widely-used tool, development will have to keep up with new versions of Eclipse.

Contents

Chapter 1

Introduction

This report is part of the thesis project of my Computer Science study at the Delft University of Technology. The goal of this project is to build a code smell detection plug-in for the Eclipse IDE framework. This idea is an extension of earlier research performed at CWI in Amsterdam by Van Emden and Moonen. In that project, a prototype of a code smell detection tool was built, using Rigi [19] to visualize the results in graph views. The Eclipse plug-in developed in this thesis project aims to provide almost instant feedback about the quality of the software system to a software engineer while he is developing software with Eclipse. As described in Fowler's Refactoring book, a programmer should divide his time between adding new function and refactoring his code to make future additions easier [11]. Detection of code smells can give the developer information on the code he is working on, indicating to him that he just produced some potentially troublesome code and might want to apply one or more refactorings before continuing with adding new functions.
This thesis report describes the project itself, gives an overview of related work, discusses the concept of code smells, introduces the Eclipse framework, presents the code smell detection process that was followed and discusses a case study that was performed using the plug-in developed. In the remainder of this chapter we discuss the history and goals of our *CodeNose* code smell plug-in project.

1.1 Project description

The main goal for this project is to develop a prototype of an Eclipse plug-in for the detection and presentation of code smells in Java source code. Eclipse is a universal platform, a development environment for anything. It can be used to develop programs and tools, to write documents or browse through a collection of documents or source code files. By acting as a platform for plug-ins, it can be used for many purposes, including integrated develop-

ment environment (IDE) plug-ins that can be used to create all sorts of applications: web sites, (embedded) Java programs, C++ programs and Java applets or JavaBeans.
In an earlier project performed at CWI (the Dutch National Research Institute for Mathematics and Computer Science) in Amsterdam, a prototype code smell detection tool was developed, aimed at impact analysis applications. The tool, JCosmo, detects a specific set of code smells and visualizes them as nodes in graphs [22]. The code smells that are detected by default by the (configurable) prototype are uses of typecasts, instanceof operators, and the Refused Bequest code smell, which is described in the Code Smells chapter (Ch. 3).
Another useful application of code smell detection is providing feedback about the quality of a software system to programmers during software development, which is the topic of this thesis. For this purpose, the integration of a code smell detection tool in a modern integrated development environment, in this project the Eclipse IDE framework, is investigated. A set of code smells will be identified specifically for the purpose of aiding the software developer. By implementing the detection of smells in separate modules, adding new code smells to the plug-in's set of detected code smells should be easy. The plug-in user should be able to make a selection from the set of available code smells, either by making a selection prior to the start of the smell detection process, or by filtering the plug-in's results after the detection has taken place. He should also be able to export the detected code smells to a formatted text file for use in other applications or reports.
A problem with the JCosmo tool prototype is its difficult deployment due to the (re)use of various complex program analysis and visualization tools (ASF+SDF Meta-Environment [20] c.q. Rigi [19]. By implementing the code smell detection tool as an Eclipse plug-in, we aim to develop a prototype that is easy to install and use. The plug-in, which was named CodeNose, will be fully implemented in Eclipse, including both the detection process and presentation of the results. The Grok tool, a relational algebra calculator which was already used in the JCosmo project to derive code smells from a collection of facts, will be the only non-Eclipse part of the plug-in.
Another difference between the prototype that was developed in this thesis project and the JCosmo tool lies in the presentation of code smells. While JCosmo gives an overview in graphs of which (cluster of) resources contain code smells, the Eclipse plug-in uses a list view, which can be used to browse all detected code smells and immediately jump to the location of the individual code smells in the Java source code to start refactoring in the Eclipse Java code editor.

1.2 This report

The next chapter contains abstracts of books and articles related to this project, as well as short descriptions of related projects. Chapter 3 describes the concept of code smells. The code smells selected to be detected by a prototype developed during this project are described in more detail. In chapter 4 we give a general overview of the Eclipse framework, as well as a more detailed description of specific Eclipse features that are used by the CodeNose plug-in. Chapter 5 presents the code smell detection approach and gives an overview of the techniques used in the detection process. Consecutively, we present the results of a case study that was done with the plug-in prototype, analyzing a software system. Finally, we draw some conclusions and discuss directions for future work on the plug-in.

Chapter 2

Related Work

This chapter presents an overview of books, articles, projects and software tools that are relevant to our code smell detection project. It describes the papers and books that were studied during this thesis project, providing insight into the worlds of software evolution research, Eclipse tool development, refactoring and code smells. Also, we provide an overview of software inspection projects and tools related to this thesis project.

2.1 Books and articles

In this section, we describe the main books and articles that were studied during this thesis project. The first articles offer a broad overview of the software evolution and reverse engineering research fields. Next, we describe the Refactoring book by Fowler [11], followed by a paper on Java quality assurance using code smell detection by Van Emden and Moonen [22]. Consecutively, we describe a book on tool development using Eclipse [3] and two articles that describe the use of the relational algebra tool Grok for manipulating facts derived from source code [5, 14]. Finally, we describe an article on test code smells [21] and a paper on automated code smell detection using a source transformation programming language [13].

Software Maintenance and Evolution: A Roadmap
K.H.Bennett & V.T.Rajlich **[4]**

This paper was studied to get an overview of the software evolution research field and investigate the main problems in this field of research. It offers a roadmap for research in software maintenance and evolution in the first decade of the new millennium. The authors give definitions of software maintenance and evolution, and describe its importance and main problem: incorporation of new user requirements. Two strategies are presented for solving this problem. First, the staged model, a new model based on current practices, consisting of five stages: initial development, evolution,

servicing, phase-out and close-down. The second model is a longer term vision on software evolution, in which software is no longer sold as a product but rather as a bundle of services, making software change easier.
This thesis project also aims at making software change easier by offering a tool to stimulate and assist refactoring during software development, which should help to produce source code that is easier to understand and change.

Reverse Engineering: A Roadmap
Müller, Jahnke, Smith, Storey, Tilley & Wong **[17]**

This article focuses on present and future of reverse engineering. The authors call reverse engineering vital and critical to the software industry, but state that in corporate settings reverse engineering tools are still not part of the standard toolset. They call for more research to make the reverse engineering process more repeatable and better managed. Better tools are needed to provide more adequate support for human reasoning in the reverse engineering process. The biggest challenge to increased effectiveness of these tools is wider adoption, which could be achieved by teaching program analysis techniques in software engineering curriculums and by tools that are better integrated with common development environments and easier to use. More effective tools will reduce the time wasted on trying to understand software. Refactoring can be used as a reverse engineering technique. A tool to find code smells in existing software, like the prototype that was developed in this thesis project, could provide good starting points for refactoring.

Refactoring: Improving the Design of Existing Code
M. Fowler **[11]**

This book, the standard work on refactoring and code smells, starts with an example program. The author shows how to make this example program's code more readable, (more) object-oriented and easier to change and maintain by introducing and applying *refactorings*. A refactoring is defined as "'a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour."' It is a technique for cleaning up code in a controlled manner. Each small step is followed by tests to check if the program is still working. Refactoring helps to prevent code decay, makes source code more readable and can be helpful in finding bugs.
Deciding when and where to start refactoring is just as important as knowing how to refactor. 'When' might be right before adding a new feature or when a bug is found. 'Where' should be decided by human intuition. All the authors can offer as hints for starting points is a list of indications of bad coding practices, dubbed *code smells*. Fowler describes these smells extensively and suggests refactorings to get rid of them. Fowler stresses the importance of testing as a part of refactoring. He claims that building tests

doesn't cost him extra time, but saves him a lot of time finding bugs instead. The book describes his testing approach, using the testing framework JUnit. The remainder of the book forms an initial catalog of refactorings. It describes a large number of these refactorings: when and why you might need them, the mechanics to carry out the refactorings and one or more examples.
The description of the use of refactoring in software development and the code smell descriptions in this book were essential to the choice and implementation of the code smells that are detected by the CodeNose plug-in.

Java Quality Assurance by Detecting Code Smells
***Van Emden & Moonen* [22]**

This article describes the earlier research performed at CWI, mentioned in the introduction (Ch. 1), that dealt with the detection of code smells in Java source code in the context of software inspections. It discusses the concepts of code smells and coding standards, presents a generic approach to software inpection tool development and describes the JCosmo tool, which detects some code smells in java source code, using a rather complex parser generator/term rewriting tool, the ASF+SDF meta-environment [20], and visualizes these code smells using Rigi [19] graphs.
During the development of the CodeNose plug-in, the prototype that was developed in this thesis project, the approach presented in this paper was followed as much as possible. Installation and use of this tool in the software development cycle is easier, because it is integrated into an integrated development environment and uses Eclipse's own features to create parse trees. Code smells that are found are presented in much the same way as compiler errors instead of graphs. The set of code smells that are detected is aimed at use in software development rather than as a quality assurance tool.

Contributing to Eclipse
***Beck, Gamma* [3]**

Contributing to Eclipse is a book meant as a guide to help tool developers get started with Eclipse. It explains Eclipse's ways, principles and features in a step-by-step approach, going through a development cycle four times, each time making the circle a bit wider, going deeper into each step. It doesn't aim to give a complete map of Eclipse, but rather shows you the basic principles and helps you find your way around the wizards, perspectives and search options. A preliminary version of this book that was studied at the start of this thesis project was helpful in learning how to set up a plug-in and profit from Eclipse's features for tool development.

Linux as a Case Study
***Bowman, Holt & Brewster* [5]**

In this article, the authors describe a case study in which they extracted architectural documentation from the source code of a large software system, the Linux kernel. This method can be used to recover the understanding of an undocumented software system which would otherwise be difficult to understand and maintain. The case study revealed unexpected dependencies between Linux kernel subsystems. The article presents the use of automated tools to help extract architectural information from a software system implementation. One of the tools that were used is Grok, which is also used by our CodeNose plug-in prototype.

Structural Manipulation of Software Architecture Using Tarski Relational Algebra
***Richard C. Holt* [14]**

This paper further explains the manipulation of software architecture facts using Grok, a relational algebra calculator which is used in this thesis project to derive code smells from facts that are extracted from source code. The article describes transformations made to architectural diagrams to make them easier to understand. The algebraic operations on relations and sets in the Grok language and the use of the Grok interpreter are presented, as well as how to combine these operations to automatically perform structural manipulations of architectural facts.
The CodeNose plug-in builds a fact repository containing smell aspects and uses Grok to infer the more complex code smells. Grok scripts can read a database of facts, perform relational calculations on these facts and write the results to a file, which can then be used by the plug-in for further processing.

Refactoring Test Code
***Van Deursen, Moonen, van den Bergh, Kok* [21]**

This article describes a special category of code smells: those found in test code. A large part of the total code in XP projects is test code, which makes it necessary to make this code easier to read and change, like production code. The test smells and the refactorings to clean them up are different from those found in 'regular' code. This article presents a list of both. The test code smells are not implemented yet in the prototype CodeNose plug-in, but they could be added in the future to make the plug-in interesting for programmers who like to use the test-driven development method.

An Interactive Interface for Refactoring Using Source Transformation
***Grant & Cordy* [13]**

In this article, the authors present a web-based interface which can be

used to automatically perform refactorings. Code smells are identified using rules in a programming language called TXL. The source code containing the smell is wrapped in XML tags. The blocks of tagged source code are presented in framed hypertext windows. Users of the web-based interface can point-and-click to automatically perform suggested refactorings. Although this article presents an interesting approach to automatically perform refactorings, Eclipse already provides support for various refactorings on Java source code. Therefore, we have no use for the TXL language's source transformation capabilities. In addition, the TXL environment has the same drawbacks as the JCosmo tool with respect to ease of installation at a user's site (it is very similar to the ASF+SDF Meta-Environment [20]). In the CodeNose plug-in, we use the Java programming language itself and the Eclipse JDT API to detect code smells. The only non-Eclipse tool it uses is Grok, which is very easy to install.

2.2 Other related work

This section describes a small selection of books, projects, tools, websites and programs related to the Code Smell Detection in Eclipse project.

2.2.1 Refactoring

Martin Fowler, author of the classic Refactoring book [11], maintains a website[1] where the latest developments on refactoring can be found. Most development environments (Borland's JBuilder, IntelliJ IDEA, Oracle's JDeveloper) offer some kind of support for automatic refactoring or can be extended with third party refactoring tools like JRefactory[2] and RefactorIt[3] for Java, Ref++[4] for C++ programs or ReSharper[5] for C#. The Eclipse framework[6] has gradually added more refactorings to its Refactoring menu.
Kerievsky explains in his book Refactoring to Patterns [16] how he incorporates patterns in his Extreme Programming design practices, helping him to avoid both over- and under-engineering. AntiPatterns [6] are very similar to code smells, but generally they refer to source code entities at a somewhat higher level. They are patterns that describe frequently observed bad solutions for a problem. Several books discussing AntiPatterns are available [6, 9, 7], as well as an online tutorial[7].

[1] http://www.refactoring.com
[2] http://jrefactory.sourceforge.net/
[3] http://www.refactorit.com/, http://developers.sun.com/prodtech/javatools/jsstandard/reference/techart/refactorIT.html
[4] http://www.ideat-solutions.com/refpp/
[5] http://www.jetbrains.com/resharper/
[6] http://www.eclipse.org
[7] http://www.antipatterns.com/briefing/

Finally, Cunningham & Cunningham have a Wiki website[8] which describes the concept of code smells, presents a list of smells and provides a forum for discussion on (individual) code smells.

2.2.2 Code inspection

Several tools that are able to perform or assist in automatic code inspection. Most of them operate on the level of source code, detecting type violations, memory allocation problems, etc. Basic examples are C code analyzer Lint [15] and its Java variant Jlint [2]. More sophisticated tools also exist, performing more complex analysis and providing all sorts of metrics. RevJava [10] is a review assistant for Java bytecode. It checks *critics* that point out possible design and style improvements, like problems with field declarations, class behaviour and spread user interface or database knowledge.
An open source tool with a different approach is Checkstyle, which can be found at SourceForge[9]. Meant as a tool that can help to enforce a coding standard, it can detect a few problems that are closely related to our thesis project. It can check size violations and some class design, (field) declaration and coding problems. It works best when integrated in a development environment. Two plug-ins to integrate Checkstyle with Eclipse/WSAD (WebSphere Studio Application Developer, IBM's development environment for web services, portals and J2EE applications, based on the Eclipse platform) are available at this time. JDepend[10] generates design quality metrics for Java packages in terms of extensibility, reusability and maintainability. It reports package dependency cycles and the hierarchical paths of the packages in these cycles. An Eclipse plug-in containing JDepend, JDepend4Eclipse, is freely available on the web[11].
PMD, an open source project at SourceForge[12], is a tool that is the most similar to the plug-in developed in this project. It scans Java source code and looks for potential problems, using techniques very similar to the ones used in the CodeNose plug-in. It detects naming problems, unused code, some design smells, empty try/catch/finally/switch blocks, code size smells and much more. Extensions can be made either in Java or by writing rules in XPath. It can be used from the command line, as an Ant task or as a plug-in for many integrated development environments, including Eclipse. Several commercial alternatives are available as well (JStyle[13], CodePro An-

[8] http://c2.com/cgi/wiki?CodeSmell
[9] http://checkstyle.sourceforge.net
[10] http://www.clarkware.com/software/JDepend.html
[11] http://andrei.gmxhome.de/jdepend4eclipse/
[12] http://pmd.sourceforge.net
[13] http://www.mmsindia.com/jstyle.html

alytiX[14], Assent[15], Aubjex[16]), some of them combining code smell detection with coding standards enforcement and/or calculation of software metrics. While PMD can detect a lot more code smells and coding standard violations than the CodeNose plug-in, it does not detect the Switch Statement or Lazy Class code smells as defined by Fowler [11], nor does it contain a tool or scripting language that could be used to derive more complex smells like Refused Bequest or Feature Envy. CodeNose detects switch Statements and small or lazy classes, and uses the Grok tool to derive Refused Bequest and Feature Envy code smells.

2.2.3 Software metrics

Software metrics are another way of determining the quality of a software system. These metrics are computed by extracting facts from a software system's source files. The simplest example of a software metric is Lines of Code. Some metrics can help in estimating code complexity (McCabe), others in calculating programming effort (Halstead). CodeCrawler [8] is a language independent reverse engineering tool which combines software visualization and software metrics. Program entities are represented in graphs by rectangles varying in size and colour.
JMetric[17] is a metrics collection and analysis tool developed at Swinburne University of Technology, Australia. It collects information from Java source files and compiles a metrics model that includes metrics information such as LOCs, Statement Count and Cyclomatic Complexity. From the model, other metrics can be calculated, including number of classes, packages and methods. Source files for JMetric are available for download.
Systä, Yu and Müller published a paper on Analyzing Java Software by Combining Metrics and Program Visualization [18]. In this paper, they describe Shimba, a tool prototype which measures properties of classes, the inheritance hierarchy and the interaction among classes of Java software components. The measures can be exported to a spreadsheet or viewed as directed graphs using the Rigi [19] tool, similar to the way it was used in JCosmo [22]. Metrics[18] is an open source Eclipse plug-in that is easy to install and use. It calculates standard, object-oriented and coupling metrics for any project in Eclipse the user attaches it to.

[14] http://www.instantiations.com/codepro/analytix/default.htm
[15] http://www.tcs.com/0_products/assent/
[16] http://www.alajava.com/aubjex/
[17] http://www.it.swin.edu.au/projects/jmetric/products/jmetric/default.htm
[18] http://metrics.sourceforge.net

Chapter 3

Code Smells

In this chapter we describe the concept of code smells in general and the selection of code smells implemented in the CodeNose plug-in in particular. An overview of the techniques that are used by the plug-in protoype to detect these smells will be presented at a later time, in the chapter on smell detection (Ch. 5).

3.1 Concept of code smells

Refactoring is a technique that is used to make source code easier to read and change. But until tools are developed that automatically rewrite source code, the software developer will have to indicate where to perform a refactoring. Even in development environments that support the use of refactorings, some user interaction is usually required to find the code that needs to be refactored and/or to choose the appropriate refactoring. A Code smell is a sign that there might be a problem in source code that could be removed by performing one or more refactorings. Code smells are not the same as syntax errors or other compiler warnings - compilers can find those for us - but rather indications of bad program design or bad programming practices that could pose a problem to you (or another programmer working on the code some time in the future) when the program needs to be changed, e.g. ported to a new platform or adding of new functionality. When found, this possibly troublesome code could be changed by applying a refactoring. We say 'could', because not every smell means there is a problem. It is just an indication, no more, no less. As stated by Fowler and Beck: 'no set of metrics rivals human intuition' [11, p. 75].

3.2 Software development code smells

There are a few things to consider when deciding which code smells to detect in the Eclipse CodeNose plug-in. Adding all code smells mentioned

by Fowler in the Refactoring book [11] is not an option, since the plug-in developed in this thesis project is merely a prototype. Some smells can be directly observed in the code (so-called *primitive smells* [22]), some can be derived from facts extracted from the code (*derived smells* [22]) and others can only be spotted by the human eye or possibly determined by using the data from a version management system like CVS (maintenance smells).
As stated above, adding all code smells is not an option. Another reason why we cannot detect every code smell is simply the fact that there is no absolute, complete list of code smells. Even though Fowler's list is a good start, there will always be domains and projects where a different set of code smells is more appriopriate, e.g. the list of test smells presented in the article Refactoring Test Smells by Van Deursen et al. [21] In the JCosmo project [22], initially a tool was developed to detect the use of switch statements, instanceof operators and typecasts. Note that only switch statements appear in Fowler's Refactoring book [11]. The derived smell Refused Bequest, also on Fowler's list, was added to the JCosmo tool at a later time.
The main goal for this project is to build a prototype of a code smell detection tool that aids software developers while they are developing code. Therefore, we selected a small number of code smells that can probably help developers the most during the software development process, without adding any domain-specific code smells for now. The selection of code smells is mainly focused at size of classes or methods and on dependency problems. The code smells that were chosen for implementation in the CodeNose plug-in are Switch Statements, Long Method, Long Parameter List, Message Chains, Refused Bequest, Feature Envy, Lazy Class and a few other class size code smells that are meant to enable setting minimum and maximum limits on the number of fields and methods.
We now describe each selected code smell in more detail. Information on the techniques used to detect each code smell can be found in chapter five.

3.2.1 Switch Statement

The trouble with switch statements is that they usually don't come alone. Often you see the same switch statement returning in various parts of the program code. Adding or removing a switch clause means (finding and) changing all occurrences of the switch statement involved. Object-oriented code does not need to have (too many) switch statemtents in it. There is usually a better solution for the problem, polymorphism for example. Fowler [11] suggests a few refactorings that can be used to get rid of switch statements, especially applicable when they switch on type code.

3.2.2 Long Method

No matter what the program paradigm is, long procedures, functions or methods are hard to understand. The longer they are, the more parameters and variables they use, and long methods are more likely to do more than their name suggests. Short methods provide for better explanation of what your code does, more reuse of code and more flexibility. Of course, with shorter methods you have more elements to take care of. More calls to short methods means more times having to check what the called method actually does. But giving your methods a good name, that shows what its purpose is, can take away the need to look at the body of methods and the need to add comments to clarify the code. Comments in code, conditionals and loops are all signs that a new method should be extracted from a long method.

3.2.3 Long Parameter List

In procedural programming languages, global data was considered bad programming, so data was passed through to functions and procedures in parameters as much as possible. This is different in object-oriented programming, because most of the data a method needs can be directly obtained from the objects themselves if they are visible to the method, or they can be derived by making a request on another parameter. Therefore, parameter lists should be shorter in object-oriented programs. Long Parameter Lists are hard to read and difficult to use and they change a lot when you need more (or other) data. Making a change to a parameter list means changing every reference to the method involved. So, keep it short.

3.2.4 Message Chain

In object-oriented code, you can encounter chains of method invocations of the form `A.getB().getC().getD()`. Sometimes temporary variables are used to shorten the chain, but nevertheless the dependency stays: a change in the interface of a method anywhere in the chain means changing each and every occurrence of these chains. These dependencies can be reduced by using the Hide Delegate refactoring anywhere in the chain, by providing a new method that takes care of a much used combination of Get-methods, so the change will only be in one place.

3.2.5 Lazy Class / Middle Man

Fowler defines a Lazy Class as a class that is not doing enough [11, p. 83]. Because each class takes time and effort to create and maintain, you should eliminate these lazy classes. They may be classes that have shrunk due to one or more refactorings, subclasses that are too small to exist on their own or classes that were added for future behaviour that was never implemented.

Fowler does not give a clear definition of what 'lazy' exactly is. Middle man classes also show lazy behaviour. They delegate all or most of their work to other classes. While encapsulation (hiding implementation details) can be useful, a class that does nothing but relay messages might as well be eliminated. If the Middle Man class adds some functionality along with the indirection, making it a subclass of the class it uses for most of the desired behaviour might be a good idea.

3.2.6 Class Size

Large Classes are classes with too many responsibilities. They have too much data and/or too many methods. The problem behind this smell is that these classes are hard to maintain and understand because of their size. Large Class code smells often coincide with Duplicated Code or Shotgun Surgery smells. The remedy is to find clumps of data and methods that seem to go together well and Extract Class. Small Classes are not mentioned explicitly in the Refactoring book [11], but classes with too few field members and/or methods might well be Lazy Classes or Data Classes (classes with nothing more than field members and getting and setting methods). These classes need to be eliminated by moving the methods to the class that use them or should be beefed up by moving methods into the class that use the data it holds.

3.2.7 Empty Catch Clause

Sometimes programmers leave the catch part of a try/catch block empty, because they meant to fill it in later but forgot, or because the compiler told them an exception could be raised by a method they called, but they did not really feel like handling the exception. But catch clauses really should not be empty. Either the code can handle the exception, or it can't. In the first case there should be some code in the catch clause handling the exception, in the latter case there shouldn't be a try/catch block, but the exception should be passed through. There is some discussion on the validity of this code smell [1]. Arguments in favour of empty catch clauses often cite examples with exceptions that 'can never happen' or 'can happen but don't matter if they do'. In order to allow a more relaxed approach to the empty catch clause, it is possible to configure the CodeNose plug-in to let catch clauses that contain a comment and/or an empty statement (;) pass without reporting a code smell.

[1] `http://c2.com/cgi/wiki?EmptyCatchClause`

3.2.8 Refused Bequest

This type of code smell is about subclasses that make none or little use of the methods and fields they get from their superclass by inheritance. Fowler and Beck call this code smell one of the faintest smells [11]. In their opinion, the only time this smell needs to be removed is if it causes confusion or if the subclass refuses the interface of the superclass. Refactorings are suggested for the traditional smell and the refused interface case, but the Refactoring book authors claim you do not have to change anything ninety percent of the time.

3.2.9 Feature Envy

Methods should mostly use the data of the class they are in. If a method make more use of data of an other class than it makes use of its own class' data, use the Move Method refactoring to put it in the right place. If a method uses data of several classes, you could try to break it up into pieces using Extract Method and then move each piece to the most appropriate owner. Often moving a method to another class makes renaming it a good idea as well. There are exceptions to the rule that says data should be in the same class as the data it uses. Examples are the Strategy and Visitor patterns as described by the Gang of Four [12], which are designed to prevent the Divergent Change smell and make changing of their behaviour easy, because all you need to do is override the right methods.

Chapter 4

Eclipse

This chapter contains a general overview of the Eclipse framework and its most prominent components, as well as a more detailed description of some aspects of Eclipse that are used by the CodeNose plug-in and are essential for understanding how the plug-in works.

4.1 Eclipse framework

The Eclipse framework is a very generic integrated development environment, designed to support (the construction of) tools that can be used to develop applications and tools or to manipulate all kinds of documents. At the core of Eclipse is a relatively small plug-in loader. All additional functionality in Eclipse is performed by plug-ins that are loaded when the framework is started or when called upon. All plug-ins can be removed or replaced at will, and most standard plug-ins offer some extension points that can be used by additional plug-ins or applications.
The standard SDK Eclipse distribution comes with the Java Development Tooling plug-ins and layout, but with the right plug-ins, Eclipse can turn into an IDE that can be used to create all sorts of Java applications, web sites, embedded programs, documents, C++ programs and/or JavaBeans, or to develop additional Eclipse plug-ins. Development environments for Cobol, Pascal and other programming languages or document formats are being developed or are already available.
The keyword when describing Eclipse is 'open'. Not only is it an open platform for integrating tools, editors, views and other plug-ins, it is also operating under an open source paradigm. Although the initiative for Eclipse was taken by big IT industry companies such as IBM, Borland, Rational and Red Hat, its source code is freely available and anyone can contribute by building their own new plug-ins, by joining in discussions in newsgroups or by submitting bug reports.

4.2 Menus, views and editors

The *workbench* is the main desktop development environment in Eclipse. It's appearance is defined by the chosen *perspective*: a collection of *views*, *editors* and *explorers* that are needed to perform a specific task, such as program development or debugging. Using perspectives, the Eclipse user can instantly switch between tasks. He or she will immediately have all views, editors and information windows that are needed to perform that task on screen at the desired location. The standard Eclipse setup (a Java integrated development environment) comes with pre-defined perspectives for resource exploration, Java development (Fig. 4.1), plug-in development and debugging. Each perspective has its own set of views, but their location and shape are freely adjustable by the user. It is also possible to save and load your own perspectives.

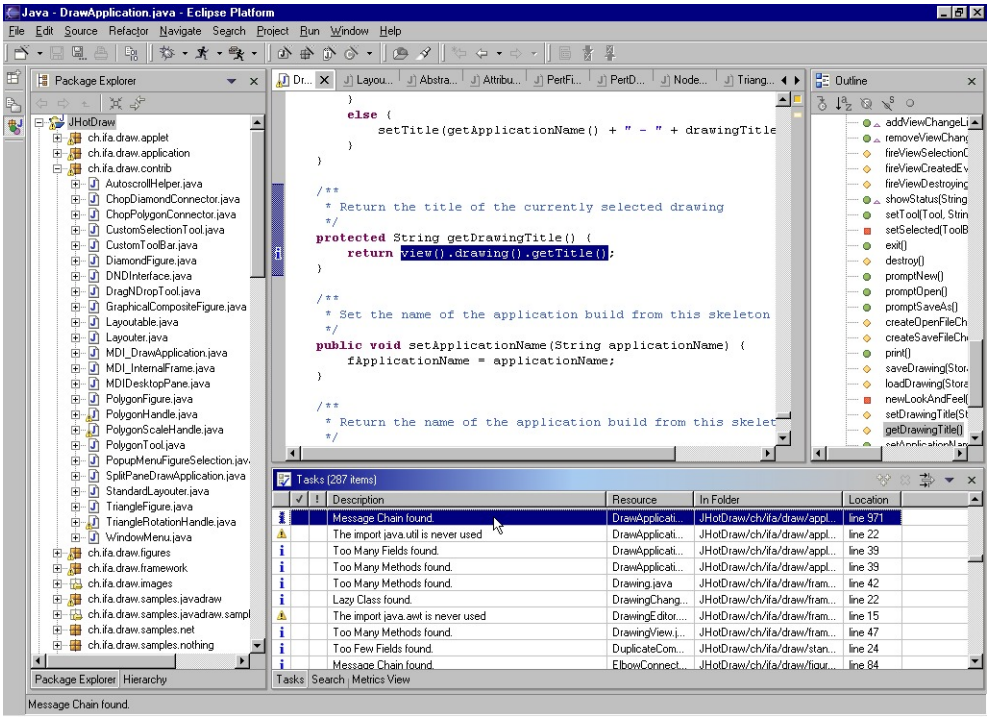

Figure 4.1: The Java Development perspective in Eclipse.

The main menu is at the top of the screen. Most objects in navigator or explorer windows and views also have their own context menu, accessible by right-clicking the object. Various file objects like Java files or plug-in definition files (plug-in manifests) can be edited with file type specific editors, either the standard Eclipse editor for that file format or an editor that was specifically developed for a certain file type and that was loaded within a plug-in. If no associated editor for a resource type is found, Eclipse will try

to launch an external editor. Wizards are available for various actions that are used a lot, require a number of steps or that have to ask for information from the user, e.g. create a new project or class.

4.3 Plug-ins

The Eclipse platform is structured around the concept of extension points: clearly defined access points where other tools can be plugged in and contribute functionality and/or their own new extension points. For example, the JDT (Java development tooling) integrated development environment is itself a tool consisting of plug-ins and designed for the development of Java software. The Plug-in Development Environment (PDE) can be used to develop own plug-ins and extension points. When Eclipse is started, the platform dynamically discovers registered plug-ins and starts them when they are needed.
Adding own plug-ins starts with deciding how the plug-in will be integrated with the Eclipse platform, e.g. the main menu or toolbar or a context menu. The extensions associated with these user interface items have to be implemented, and a declaration of the implementation's classes and extension points will have to be provided in an XML-format plug-in manifest file (called plugin.xml). More information on developing plug-ins can be found on the Eclipse main website[1] and in the Eclipse platform's help files.

4.4 Refactoring in Eclipse

The Java Development Tooling plug-in for Eclipse provides support for refactoring Java source code. A number of code transformations as described in Fowler's Refactoring book [11] can be performed with the help of the Eclipse workbench. Refactorings can be started from the Workbench main menu or via the context menus of Java views (Package Explorer, Outline) or inside an editor. For some refactorings, Eclipse will then ask for additional information on the refactoring it is expected to perform. If the user chooses, Eclipse can present a preview of all the changes that will result from the chosen refactoring - before and after views side-by-side. In some cases, Eclipse will indicate to the user that it is not able to perform the refactoring and give the reason for that failure. An undo/redo system is available for the refactorings, although undo may not be an available operation after every possible refactoring.
Eclipse refactorings are implemented as a three-step process. The first step is to determine what compilation units (source code files) are affected by the chosen refactoring. For a renaming refactoring, that may be quite a lot

[1] `http://www.eclipse.org`

of units, because at every location that the field, method or type that you want to rename is called, the reference must be changed. Consecutively, the structure of the code of these compilation units is analyzed in two ways. One way is the Java Model, which makes it possible to look at all declared types, import statements, fields and methods. For a closer look at the code, Eclipse makes use of abstract syntax trees. These give access to the code at the statement level, which is not accessible from the Java Model. The Final step in the implementation of Eclipse refactorings is the process of making the code changes that result from the chosen refactoring. This is done in a way that makes it possible to undo the operation later.
Most refactorings in Eclipse 2.1 seem to be simple operations, but even apparently straight-forward transformations like renaming or moving an element could trigger a whole cascade of changes in the file you are currently working on and in other files that depend on or use that file or element as well.
Refactorings available in Eclipse 2.1 are:

- renaming an element
- moving an element
- changing the signature (parameter names, types, order) of a method
- converting an anonymous class to a nested class, a nested class to a top level type or a local variable to a field
- moving methods and/or fields to or from a subclass (push down/pull up)
- inlining or extracting variables, methods or constants
- extracting an interface
- use supertype where possible
- encapsulating fields

These refactorings are available from the Refactor menu, and from context menus in the Java editor and some views (e.g. the Package Explorer).

4.5 Markers

Markers are the Eclipse mechanism for resource annotations. They can be used to present information (compiler errors and warnings, bookmarks, to-do list items) to the Eclipse developer. This information could be generated by Eclipse or the JDT plug-in, but can also be generated by self-built Eclipse plug-ins. For example, the Eclipse Java Development Tooling plug-in reports

compiler errors and warnings with the help of markers. These errors and warnings show up in the Tasks view and can be used to immediately navigate to the corresponding location in the program code. The Java editor feature that surrounds a piece of code with a try/catch block automatically creates a to-do-marker in the catch clause of the try/catch block. These to-do-markers are visible in the source code as a simple Java comment starting with `// TODO:`, and in the Tasks view as to-do items. For the CodeNose plug-in project, we use markers to show the user what code smells were detected by the plug-in and where in the source code they can be found.

Eclipse has three pre-defined marker types:

- **Problems:** compiler errors and warnings
- **Tasks:** to-do items
- **Bookmarks:** other locations the user might find useful to jump to quickly

These marker types can be subclassed to create your own marker types for your Eclipse project. These marker subclasses can be used to add attributes or change their appearance in the Eclipse Tasks view, creating more marker 'categories' which is a good idea when you plan on generating a lot of markers. New marker types are declared in the plug-in manifest file.
For the CodeNose plug-in, we created a subclass of the problem marker type. We chose this type to inform the plug-in user on detected code smells, because the smells provide the user with information about the source code in the analyzed resources, much the same way compiler errors and warnings do. To distinguish the code smells from compiler-generated messages, a lower-severity label was used. Using this lower severity causes Eclipse to show the code smell markers generated by the CodeNose plug-in with an icon different from the compiler-generated errors and warnings, which should prevent mistaking code smells for compiler errors.

4.6 Preference pages

Eclipse's preferences dialog, available from the Window menu, contains a hierarchical list of user preferences. Each item in this list corresponds with a preference page which is displayed when the item is selected. These preference pages are used to set preferences for the CodeNose plug-in, like enabling or disabling detection of a certain type of code smell or setting maximum lengths for methods or parameter lists. By using preference pages, setting parameters for the detection of code smells using the CodeNose plug-in is fully integrated into the Eclipse user interface.

Preference pages for a plug-in can be added to the Eclipse preferences dialog by defining an extension point for `org.eclipse.ui.preferencePages`

in the plug-in manifest file. Attributes like the corresponding class file name, the path of the preference page in the hierarchical list in the preference dialog and the preference page's name can also be specified in the plug-in manifest. The class file must be an implementation of the `WorkbenchPreferencePage` interface. The class contains Java source code for putting text and input fields on the preference page. More information on adding preference pages can be found on the Eclipse website[2].

A slightly different approach to letting the user of a plug-in define his or her prefered settings would be the use of Properties instead of Preferences. Using property pages, the user would be able to set plug-in preferences on a per-project basis, as opposed to the current situation in which settings are made system-wide. This might be a useful feature if users of the plug-in want to use a different set of parameters for every project and tend to work on several projects at the same time. Large projects may require a different set of code smell preferences than a project with only a few classes. Since the plug-in developed in this thesis project is only a prototype, we chose to implement Preference Pages, but using properties could be a useful approach in a future version of the plug-in.

4.7 Adding actions

The detection of code smells can be invoked by selecting a project (or one or more packages or Java files) and clicking on Detect code smells in the object's context menu. Both the Detect code smells and Flush code smells menu items are defined in the plug-in manifest file. ObjectContributions containing the CodeNose menu and the two action labels were added to the extension point `org.eclipse.ui.popupMenus` for the objects `IJavaProject` (a project), `IPackageFragment` (a package) and `ICompilationUnit` (a Java source code file). The Remove selected code smells and Write smells to file menu items are added to the code smell marker objects in the Tasks view in a similar manner. Code smell detection in the current version of CodeNose is restricted to processing of complete Java files, but future versions of the plug-in could present the user a more generous user interface, allowing the detection of code smells in selected types, methods or even fields. The `ASTParser` class that is new in Eclipse 3.0 makes it possible to parse any Java element into an abstract syntax tree. Java language parsers of type `ASTParser` are highly configurable, allowing the plug-in developer to set compiler options, the kind of constructs to be parsed or whether to build an abridged tree, focusing on the declaration at a given position in the source code.

[2] for example at `http://www.eclipse.org/articles/Article-Field-Editors/field_editors.html`

4.8 Responding to changes

The current version of the CodeNose plug-in only starts searching for code smells when asked by a user that selects the Detect code smells menu item from an object's context menu. A possible future feature for the plug-in could be to let it react immediately to any changes in the source code of a project and let it almost instantly (some processing time would of course be required) update the code smell markers in the Tasks view. An important prerequisite for this feature is that the processing that occurs as a result of a change in the source code does not become a source of annoyance to the plug-in user, for example by taking too much time to update the Tasks view or even freezing the screen for a while, due to heavy-load compilation tasks. The JDT Java compiler in Eclipse already works with this incremental build approach, recompiling only the files in a project that are affected by changes. At this time, there is no API (Application Programming Interface) to use the abstract syntax trees that are created in that compilation process, so the plug-in will still have to do it's own parsing to get the abstract syntax trees it needs to detect code smells. But since only the files that are affected by the change need recompilation, this process will take approximately the same amount of time as the compilation performed by Eclipse. After recompilation, detection of the more complex code smells has to be performed again, based on the updated facts file. This will take some processing time as well. An experiment (user study, case study or field observation [17]) with a version of the plug-in that includes this feature could show if the benefit of always having an up-to-date list of code smells outweighs the possible disadvantage of the extra processing that occurs each time the list of code smells in the Tasks View is updated.

Chapter 5

Smell Detection

This chapter decribes the approach followed for development of the Eclipse CodeNose plug-in and the techniques that it uses to detect code smells. It also explains how the plug-in is used. Finally, we discuss the process of adding a new code smell to the plug-in, using one of the already implemented code smells as an example.

5.1 Detection approach

Code smells are recognizable in source code by one or more *smell aspects* [22]. When all aspects of a particular code smell are found using static analysis of the code, that code smell has been detected. The detection of code smells can be achieved by the following three-step approach (Fig. 5.1). A parser analyzes the Java source code files and produces abstract syntax trees, containing all relevant structural information. Making use of the source code (instead of byte code) even makes it possible to analyze code that is not ready for compilation yet, or that still has a few bugs in it. The absence of program parts or the existence of compilation errors might hinder the detection process somewhat, because not all code can be transformed into abstract syntax trees, but this will not completely break down the code smell detection process. Whenever a syntax error is found during the parsing process, the abstract syntax tree node(s) involved are marked as being malformed by setting a flag constant on the node(s). Depending on the seriousness of the error, some source code may not be represented in the abstract syntax tree (see Section 5.4.8).
When the parser is done, an analyzer (visitor) traverses the abstract syntax trees, collects smell aspects and stores them in a repository. Primitive smells, smells that can be found directly in the source code, are presented in the Tasks view in Eclipse. Using the smell aspect (or *fact*) repository and Grok scripts, more complicated code smells can be derived. The derived code smells that were found by these scripts are then presented in the Tasks view

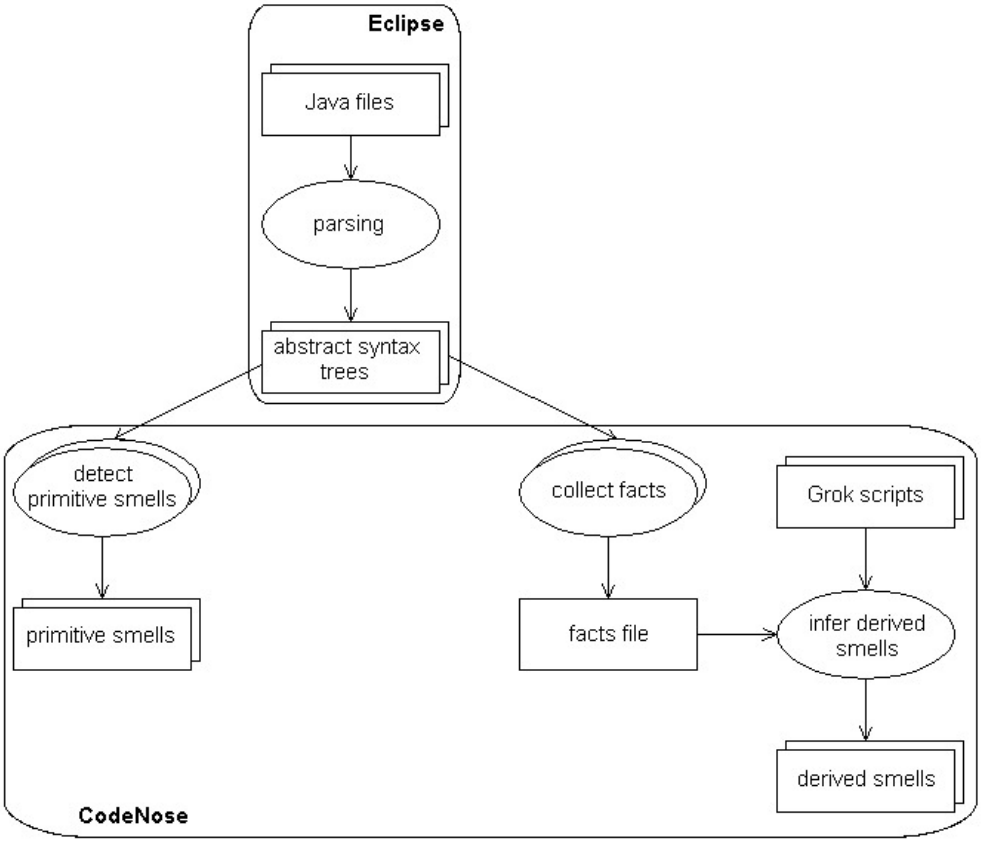

Figure 5.1: The code smell detection process.

as well.

Because the plug-in prototype developed in this thesis project will only detect a small number of code smells, it should be developed with ease of addition of new code smells in mind. As we discussed in the introduction, we set out as design requirements that the user interface part of the program and the classes that actually perform the detection of code smells should be put in separate packages, and each code smell should have its own class. Putting every code smell in its own separate module should make it easier to add new smells, although performance might be affected.

Obtaining the abstract syntax trees to be used for the detection of code smells can only be achieved by explicitly calling a method that parses a Java file. The ASTs that are built by Eclipse JDT during its own compilation processes are not made available through API. Building the abstract syntax trees is probably the task that requires the largest amount of time. Building the trees just once and reusing them for all smell detectors is likely to give the best performance results, keeping extensibility in mind.

5.1.1 Presenting the results

Eclipse's flexible nature gives us several options for presenting code smells that were detected in the source code. The most obvious option is to use the Tasks view, which is visible in the Java Development perspective and other perspectives. Eclipse presents compiler errors and warnings, to-do items and user-added notes and bookmarks in the Tasks view. Adding code smell information to this view will give the software developer a useful overview of all detected code smells in the same area as the aforementioned compiler errors and warnings. Eclipse users can filter or sort items in the Tasks view. If he or she presumes a code smell to be solved or as being harmless, the user can remove a code smell indication from the view. It is also possible to remove code smells associated with a selection of Java files or packages, or even all code smells that were detected in an Eclipse project.
Eclipse users can find the location of a compiler error in the source code by clicking on them in the Tasks view. It should be just as easy for the user of the plug-in to find the locations where code smells were found. This can be achieved very easily, using Eclipse's marker feature. Marker types connect compiler errors and warnings, to-do list items and bookmarks to locations in Java source files. Eclipse tool developers can add own marker types and configure the way they are displayed in the Tasks view. Clicking on a code smell marker, that was created by the plug-in, in the Tasks view should take you to the position in the source code where that code smell was detected.

5.1.2 Case study

After the code smell detection plug-in prototype has been developed, it will have to undergo a serious test. The plug-in will have to perform code smell detection on a Java software system that has a reasonable number of lines of code, at least a few thousand. Any software system from which we can obtain the Java source code and has a significant number of LOCs will be sufficient for this task. This case study will focus on performance of the plug-in and the usefulness of the code smells that were detected.
We chose to use JHotDraw[1] as subject for our case study. This software package is used as example and subject system for various other projects and courses within the Software Evolution Research Lab at the Delft University of Technology.
The results of the performance test and a study of the distribution and value of the code smells that were detected in the JHotDraw source code can be found in the Case Study chapter (Ch.6).

[1] Version 5.3, available at `http://www.jhotdraw.org`

5.2 Abstract syntax trees

Abstract syntax trees are structures in an intermediate phase of a compiler [1, 23]. They are meant as interfaces between the parser and later phases of the compiler. Abstract syntax trees hold all information on the syntactic structure of a source program, but without any semantic interpretation. These trees can be traversed to explore the source code, looking for primitive or derived smell aspects. Eclipse contains a ready-made tool for the traversal of these trees, the `ASTVisitor` class.
Every node of the tree is a Java object of type `ASTNode`. `ASTNode` has various subclasses which represent all of the different abstract syntax tree node types, e.g. `ForStatement`, `SuperMethodInvocation` or `Block` (see the `org.eclipse.jdt.core.dom.ASTNode` declaration for all types). The root of an abstract syntax tree has the type `CompilationUnit`, which is a subclass of `ASTNode` as well. It usually spans the whole range of a Java source code file and contains some extra fields and members, e.g. lists that contain information on compiler messages and problems encountered during parsing or type checking of this file. We will give an example of a very simple Java method and its corresponding abstract syntax tree, using the `getFillColor` method found in the `ch.ifa.draw.figures.AbstractLineDecoration` class of JHotDraw. This is the source code:

```
/**
 * Returns color with which arrow is filled
 */
public Color getFillColor() {
  return fFillColor;
}
```

This Java code, when parsed by the Eclipse Java compiler, is transformed into the abstract syntax tree that is shown in Figure 5.2.

5.2.1 ASTVisitor

`ASTVisitor` is an abstract Java class in the Eclipse JDT (Java Development Tools) library that can be used to traverse abstract syntax trees. The `ASTNode` class contains a method `accept` that is passed an `ASTVisitor` parameter which will visit that node. For each different abstract syntax tree node type, the `ASTVisitor` object has a `visit` and an `endVisit` method that can be used to perform type-specific actions when a node of a concrete AST node type is encountered. The `boolean` return type of the `visit` method is used to decide whether the given node's child nodes will be visited or not. The `endVisit` method is called after all of the node's children (if any) have been visited. `ASTVisitor` also has two methods for performing actions before and after the visit of any node, regardless of type. These methods are empty

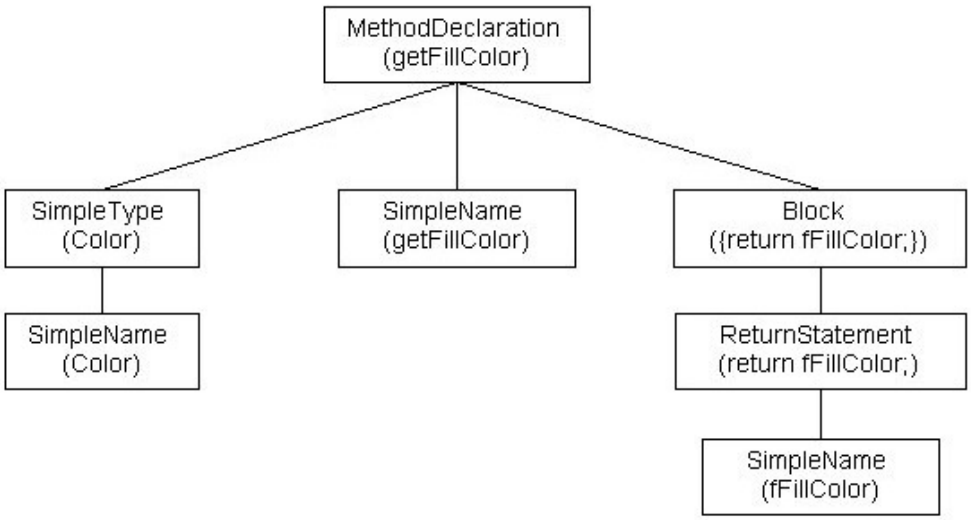

Figure 5.2: Abstract syntax tree for the `getFillColor` method.

by default, but subclasses of the abstract `ASTVisitor` class can reimplement these methods to perform some appropriate action before, during or after the visit of a node. By subclassing the `ASTVisitor` class and reimplementing the `visit` and `endVisit` methods, we can find those parts of source code we consider to be primitive code smells and collect facts from which we can derive more complex code smells.

5.3 Grok

Derived smell aspects are inferred from the collected facts using a tool called Grok. This tool, developed by R.C. Holt at the University of Waterloo in Canada, was meant to be used in a software system called the Portable Bookshelf[2]. It is a calculator for relational algebra, which enables us to perform calculations on relations and sets, for example to determine the union of two sets or the transitive closure of a relation. In PBS, Grok is used to infer higher level views from low level facts. With the help of a small program, a Grok script, we can combine facts and relations in the facts repository and infer derived code smell aspects that may indicate more complex code smells, such as the Refused Bequest or Feature Envy code smells. These Grok scripts usually will read facts from one or more fact bases, perform some operations on the fact relations and sets, and write an output set or relation to a file. The contents of this file can then be used by Java code in Eclipse for further processing. If it is determined, during that

[2] `http://www.swag.uwaterloo.ca/pbs/`

processing, that all aspects of a derived code smells were found in the facts database, a code smell will be reported.

5.4 Detecting smell aspects

In this section we will discuss which detection techniques, Eclipse API features and newly-built classes are used by the prototype CodeNose plug-in to detect the primary and derived code smell aspects. A description in full detail of the implementation of one of the code smells can be found in the Adding New Smells section (5.6).

5.4.1 CodeSmellCollector class

The `ASTVisitor` class in the Eclipse API is very suitable for use in the code smell detection process. A subclass of `ASTVisitor`, the `CodeSmellCollector` class, was created to add some methods and fields that are common to the concrete code smell detection classes to `ASTVisitor`'s functionality, and offer some additional methods for reporting code smells and getting information from the abstract syntax tree or Java source code. The `CodeSmellCollector` class, like its parent `ASTVisitor`, is abstract. It contains only one abstract method, the `collectSmells` method that is used for starting the detection process on an abstract syntax tree node. The following subsections contain descriptions of all concrete classes implemented in the CodeNose plug-in that are subclasses of the `CodeSmellCollector` class. These classes are meant to detect specific primitive code smells.

5.4.2 Switch Statement

The simplest code smell detection class is the `SwitchCollector` class. This class detects primitive code smells of the Switch Statement type, as described in the Code Smell chapter (Ch. 3). `SwitchCollector` is a concrete subclass of the abstract `CodeSmellCollector` class.
As described in the description of `ASTVisitor` (Subsection 5.2.1), at least one of the `visit` or `endVisit` methods in the `ASTVisitor` class needs to be overridden. In the case of the Switch Statement code smell, we want to report a code smell every time we encounter a switch statement in the abstract syntax tree. Java switch statements are translated into the `SwitchStatement` AST node type by the Eclipse parser, so we have to override the `visit` method for the `SwitchStatement` node type. In the body of the `visit` method, a method from the `CodeSmellCollector` class is used to report the detection of a code smell. The method returns the value *true*, indicating that the child nodes in the abstract syntax tree of the `SwitchStatement` node have to be visited as well, because they might contain more Switch Statement code smells.
The abstract syntax tree that will be analyzed by the plug-in is passed as a

parameter to the constructor method of the `SwitchCollector` object. Code smell detection will start when the `collectSmells` method of that object is called. This method, the only abstract method in the `CodeSmellCollector` class, uses the `findSmellsInAST` method in the same class to start the detection of code smells in the abstract syntax tree.

5.4.3 Long Parameter List

The `LongParameterListCollector` class is only slightly more complicated than the very simple `SwitchCollector`. If, during the traversal of an abstract syntax tree, a parameter list is encountered, the plug-in checks whether the number of parameters in that list is higher than the maximum set for parameter list lengths. This maximum can be set by the user of the plug-in through the preference page mechanism in Eclipse. When asked to detect code smells, the CodeNose plug-in creates a `LongParameterListCollector` object and sets the maximum parameter list length to the value on the Long Parameter List Preference Page.
Parameter lists are found in `MethodDeclaration` nodes in the abstract syntax tree. If the size of the parameter list exceeds the maximum, the plug-in reports a code smell at the source code file offset of the method declaration.

5.4.4 Long Method

As described in the chapter on Code Smells (Ch. 3), Long Method code smells are methods that simply contain too many lines of code. Unfortunately, there is no agreement on what the maximum length of a method should be, nor on what exactly comprises a line of code. The `LongMethodCollector` class has an integer field for the maximum method length. To handle different interpretations of what a line of code is, the `LOCCounter` class was created. Objects of this type can be configured to count or disregard lines that contain comments, white lines and/or lines that contain nothing more than an opening { or a closing brace }. Also, lines occupied by the header information of a method declaration can be counted or not. This way, users that want to set a maximum for the number of lines a method occupies on their editor screen and users that prefer a more strict interpretation of Line of Code can both adjust the method of counting lines to their liking.
Interface classes contain method declarations without method bodies, so these classes can be disregarded when checking for Long Method code smells. Therefore, an overriding `visit` method for the `TypeDeclaration` abstract syntax tree node type prevents the `LongMethodCollector` from wasting its time on interface classes. The overriding `visit` method for method declarations in the `LongMethodCollector` class is more complicated than in the previously described classes. First, a check is done if the method is abstract or

not. Abstract methods don't have a method body, so they can be disregarded. In order to obtain the source code that corresponds with the complete method (header and body) that is declared in the `MethodDeclaration` AST node found, the method `getSourceFromNode` from `CodeSmellCollector` is called. The `MethodDeclaration` node has two methods (`getStartPosition` and `getLength`) to determine the start and end positions of the source code corresponding with this abstract syntax tree node. A `LOCCounter` object with the appropriate parameters is created. By calling its `getLOC` method, the number of lines of code is determined. If the number of LOCs exceeds the maximum set, a Long Method code smell is reported.

5.4.5 Message Chain

Message Chain code smells indicate that there is a method calling a method that possibly calls another method, e.g. `objectA.getB().getC()`. To find these smells, we will have to find the first method invocation in the abstract syntax tree (`exp.getC()`) and see whether there are more method invocations in the expression part of this invocation (in the case of the previous example `exp` is `A.getB()`). The `MethodInvocation` AST node type is defined as:

```
MethodInvocation:
  [ Expression . ] Identifier
    ( [ Expression { , Expression } ] )
```

When a `MethodInvocation` AST node is found in the abstract syntax tree, the plug-in starts a search for more method invocations in the expression subtree of this node. Note that the bottom line of the `MethodInvocation` definition above defines the parameter list of the method invocation. These parameter lists could contain Message Chain code smells themselves, so each parameter in the list will have to be searched using a separate `MessageChainCollector` object. The expression subtree is searched by the `ChainDepthSearch` inner class object, which updates a counter while traversing the subtree and checks on each increase of the counter if the message chain exceeds the maximum length, in which case a code smell is reported. If the user wants to ignore message chain code smells caused by certain often-used methods, such as `trim` for `String` objects, he or she can add the names of these methods to a list in the Preference Page menu area.
The goal of avoiding these message chains is to achieve that an object has no structural knowledge of another object. If the other object changes, all objects using its structural information will have to be changed as well. While avoiding the use of message chains is a good start, it does not solve the dependency problem completely, because the same unwanted behaviour can be achieved using temporary variables, method invocations passed as a parameter, or a combination of these, possibly intertwined with purely

innocent statements and expressions. When available, this code smell class could be replaced by a more complex Law of Demeter code smell detection mechanism that can detect the use of temporary variables that cause dependency problems.

5.4.6 Class Size

The code smell collector classes for the Class Size code smells are `TooFewFieldsCollector`, `TooManyFieldsCollector`, two similar classes for the number of methods and the `LazyClassDetector`. The code smell collectors for too few or too many methods or fields are very basic `CodeSmellCollector` subclasses that override the `visit` method for `TypeDeclaration` AST nodes and check if the number of methods or fields in that declaration exceeds a certain maximum or minimum set. The `LazyClassCollector` computes the average number of lines of code of concrete classes. If this average is below a minimum set in the object, a Lazy Class code smell is reported. If a `MethodDeclaration` is encountered in the abstract syntax tree, the method counter is incremented and the number of lines in the body of the method (not counting any white space, comment lines or lines with nothing more than an opening or a closing brace) is added to another counter. At the end of the visit to the whole AST tree, the average number of lines of code per method is computed and compared to the minimum set in an overriding `endVisit` method for the `CompilationUnit` AST node.
To set a minimum average number of lines of code, you need an edit field on the preference page for adding preferences of type `double`, because the minimum will likely be a floating point number like 1.8 or 2.5. The standard Eclipse preference page edit fields include fields for editing strings and integers, but not for preferences of type `double`. Therefore, the `DoubleFieldEditor` class was created, which can be used on new preference pages for user-added smells in the future.

5.4.7 Empty Catch Clause

Empty Catch Clause code smells can be found by checking the content of the body part of a `CatchClause` AST node. Parameters can be set to determine whether catch clauses with a comment or an empty statement (`;`) in the clause block should be considered empty. Checking for the character combinations that are at the start of a comment, `//` or `/*`, is made a little less trivial by the fact that these character combinations might also appear in string literals.

5.4.8 FactsCollector class

To detect more complex smells like Refused Bequest and Feature Envy, a collection of facts, smell aspects needed to find these smells, has to be

made. For this purpose, the `FactsCollector` class was developed, a subclass of `ASTVisitor` that traverses an abstract syntax tree to find all relevant facts and write them to a facts file. The Refused Bequest Grok script needs information on parent classes, methods, fields, and the methods and fields that are used by a class. For detection of Feature Envy code smells, facts stating what methods a class has and which methods it uses are needed. When new smells need other facts, the plug-in user can add `addFact` lines to the existing `visit` methods in `FactsCollector`, or add completely new `visit` methods that are not implemented yet for other AST node types. For the `FactsCollector` to be able to find all relevant facts, two things have to be kept in mind. First, the abstract syntax trees have to be built with the `resolveBindings` parameter of the `AST.parseCompilationUnit` method set to `true`, because in some cases the techniques used to find facts require the availability of information on *bindings*. Second, if the Java source code to be searched contains any *assert* statements, the Eclipse workbench that the CodeNose plug-in is running in will have to use a Java Virtual Machine of version 1.4 or higher, because assert statements were new in Java 1.4. Methods' body sources containing assert statements will be empty in abstract syntax trees built with a Java Virtual Machine with a version number lower than 1.4, and therefore any facts within these methods will escape detection.

5.4.9 GrokSmellCollector class

The `GrokSmellCollector` class is an abstract Java parent class for concrete subclasses that use the Grok tool to derive code smells from facts collected in a file. Starting a Grok script to start the detection process for a particular smell is made easy by the `findGrokSmells` method, which takes care of handling all file input and output caused by running the Grok tool. Processing the results of the Grok tool should be done by implementing the abstract `processGrokOutput` method, for example by reading the results from a file where Grok wrote the relevant relation(s) to. The input/output stream handling scheme is based on an article by Michael C. Daconta for JavaWorld[3]. `GrokSmellCollector` also contains methods for reporting code smells and for finding back the correct Java object of a class or method that corresponds with the name or number of a class or method found in a text file that contains Grok results. Recovery of the right object is necessary for linking a code smell marker to the right location in the Java source file.

5.4.10 Refused Bequest

The Java class part of detecting the Refused Bequest code smell consists of implementing the `processGrokOutput` method of `GrokSmellCollector` by

[3] `http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html`

reading the results of the accompanying Grok script, finding the class object associated with the class names read from the file and reporting a Refused Bequest code smell for each of these classes. The more complicated part of the process is done by a Grok script that finds all classes that do not use any of the methods or fields offered to them by their parent class. The heart of the script looks like this:

```
supersmethods := (inv super) o method
supersfields := (inv super) o field
supermethodscalled := supersmethods ^ call
superfieldsaccessed := supersfields ^ varaccess
supersweknow := (dom super) ^ (dom method)
refbeqcandidates := supersweknow . super
refbeqclasses := refbeqcandidates - (dom supermethodscalled) -
  (dom supermethodinvoc) - (dom superfieldsaccessed)
```

The first two lines of the script create relations between classes and all methods or fields of their parent classes. The next two lines use these relations to determine which parent class methods or fields are actually used by a class. Then the script determines which classes are possible candidates for the Refused Bequest code smell. These are all classes that have a parent class which was part of the project or selection of packages or Java files to be searched for code smells. In the last line of the Grok script, the classes that use a method or field that was inherited from their parent class are taken out of the pool of candidates. The remaining classes are written to the results file and a Refused Bequest code smell will be reported for these classes.

5.4.11 Feature Envy

In the detection process for finding Feature Envy code smells, the Java source code plays a bigger role than just reading the results of the Grok process from a file. Some processing of these results has to be done to find the code smells. The script is used to throw away facts that are not relevant for the Feature Envy detection process. The script part is fairly simple:

```
domain := foreignuse . dom method
range := dom method ^ rng foreignuse
envycombs := domain X range
envytupels := foreignuse - (foreignuse - envycombs)
```

The script determines all possible combinations of methods and foreignuses (a method using methods or fields of other objects) within the scope of the selection of Java files that are being searched for code smells.

All foreignuse tupels not relating to classes within this scope are then subtracted from the foreignuse relation. The remaining tupels, the envytupels relation, are of interest for the detection of Feature Envy code smells.
In the `processGrokOutput` method of the `FeatureEnvyCollector` class, the envytupels relation is read from the results file. The results are placed in a `Map` object which has the envytupels as keys and the number of times each tupel shows up in the results file as value. When all tupel occurrences are counted, the number of occurrences for each combination of calling method and envied class is checked. If this value exceeds a certain maximum set for foreign use of methods and/or fields, a Feature Envy code smell is reported.

5.5 Using the plug-in

In this section, the locations in the Eclipse user interface where the CodeNose plug-in's features can be found are described. CodeNose menu options show up in the context menus of markers and Java resource objects.

5.5.1 Java resource objects

The CodeNose menu can be found in the context menu of Java objects (projects, packages and Java source code files - see Fig. 5.3). Right-clicking on one or a selection of these objects in for example the Package Explorer in Eclipse will cause this context menu to appear. The CodeNose menu is near the bottom of the context menu and contains two menu items: Detect code smells and Flush code smells.

The first option will start the code smell detection process for the selected Java resource object(s). The second option will remove the code smell markers associated with the selected Java object(s) from the Tasks view.

5.5.2 Marker objects

The code smell markers in the Tasks view area of the Eclipse screen have their own CodeNose menu, accessible by right-clicking one or a selection of lines in the Tasks view (see Fig. 5.4). This menu has two options as well: you can either write the selected code smells to a text file or remove them from the Tasks view.

Selecting the first option will cause Eclipse to ask you for a file name. The selected code smell(s) will be written in text format to a file with the chosen name. The second option will remove the selected markers from the Tasks view. If, in a subsequent code smell detection process, one of the removed code smells is found again, it will reappear in the Tasks view. Future versions of CodeNose could include a list of removed code smells to prevent this. An option would be to present this list on a preference or

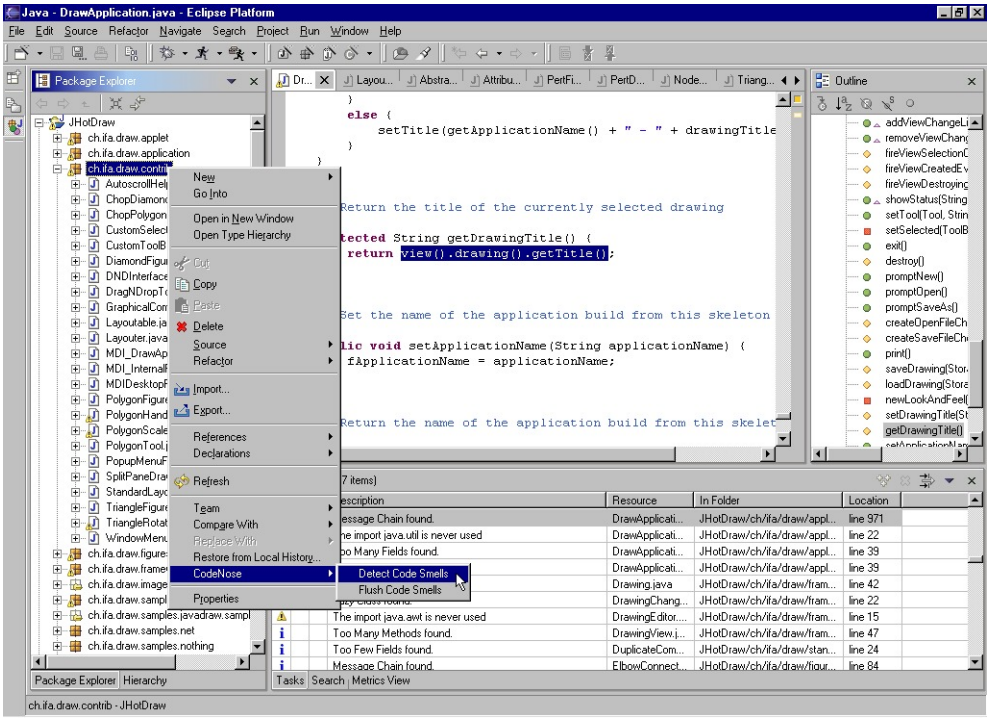

Figure 5.3: Java resource objects context menu.

properties page, with added facilities to reinstate one or more of the removed code smells.

5.6 Adding new smells

Contributing a new code smell to be detected by the CodeNose plug-in requires the creation of a code smell collector class, a preference page class, adding a few lines to the plug-in manifest file and adding some code to the plug-in classes that will set default preferences and include the new code smell in the detection process. We chose to let a user who wants to add a new smell develop it in the plug-in's Java code, because adding some kind of scripting language to define code smells would either reduce the freedom and ability to use all information available in the abstract syntax tree and Java source code to find the code smells he or she is interested in, or - if the scripting language would be just as powerful as the Java code plus abstract syntax tree - force the developer to learn a new scripting language, while he or she already understands Java. After all, this plug-in is designed to be used by Java software developers.
The number of steps to add a new smell is mainly as high as it is because of the use of preferences. Hard-coding the maximum length for for example

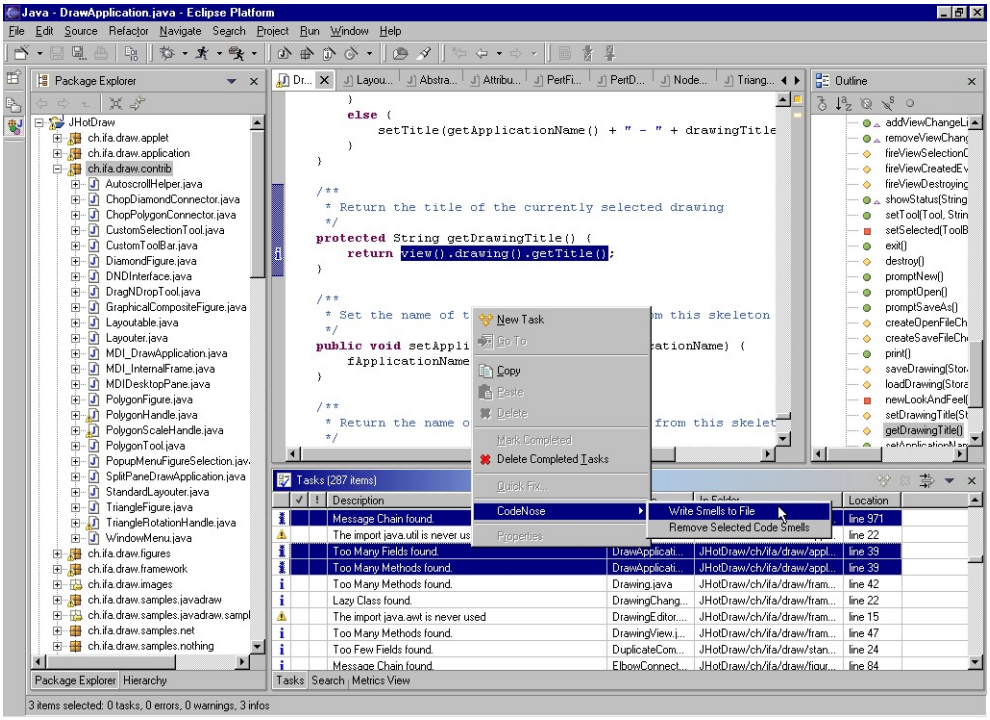

Figure 5.4: Marker objects context menu.

the Long Method code smell would reduce the number of steps to two, while making the step where the new smell is added to the detection process easier as well, because no preferences have to be set. The plug-in user is free to add this type of hard-coded parameter code smells, but this was never a serious option for the development of the code smells that were implemented in this project. Every code smell in the CodeNose plug-in has at least one preference: to enable or disable detection of that code smell through a menu option. If there is any type of parametrization possible for a code smell, the plug-in should offer setting these parameters on a preference (or properties) page.

In a future version of CodeNose, some sort of mechanism could be implemented to enable the plug-in user to add new smells without having to edit and recompile the plug-in's classes. This could be achieved by picking up all available code smell detection classes using the `CLASSPATH` environment variable or by using an XML file that lists all code smells with their classes and preferences. Default preferences could be put into an initialization method in the code smell detection class or could be read from the XML file. Code Smell Import and Export Wizards could be created to keep the user from having to edit this list by hand. A mechanism that uses JAR-files to store the code smell classes and these wizards to import and export them into the

plug-in would greatly facilitate the exchange of code smell detectors.
In the following subsections, a detailed example of the five steps necessary for adding a new code smell to the CodeNose plug-in is presented.

5.6.1 TooFewFieldsCollector class

Adding a new code smell starts with the creation of the code smell collector class. When creating a new class, using the New Java Class wizard in Eclipse may speed up the process, because it can create method stubs for constructors and/or abstract methods. Copying an existing class, renaming its names and members where necessary and adjusting its behaviour will probably speed up the process some more (following Gamma and Beck's Monkey See Monkey Do rule [3]). All code smell collectors and related classes are located in the same package.

```
package org.eclipse.contribution.codenose.codesmells;
```

The import statements are necessary, but it is not difficult to determine which classes to import. The developer can make Eclipse produce these lines by using the Organize Imports feature in Eclipse.

```
import org.eclipse.jdt.core.ICompilationUnit;
import org.eclipse.jdt.core.dom.CompilationUnit;
import org.eclipse.jdt.core.dom.TypeDeclaration;
```

After the main class comment, the class declaration starts. `TooFewFields-Collector` is a subclass of `CodeSmellCollector` and has one private field for remembering the minimum number of fields allowed.

```
public class TooFewFieldsCollector
  extends CodeSmellCollector {

  private int _minFields = 1;
```

The `TooFewFieldsCollector` has four constructors. One without parameters, one with a Java source file as a parameter (that will be passed to the compiler to build the abstract syntax tree), a third constructor with source file and abstract syntax tree as parameters (to enhance performance, a user of this object may want to build an abstract syntax tree and reuse it with several code smell collectors) and a fourth where, in addition to the Java source file and the abstract syntax tree, the minimum number of fields can be set right away. The constructors for the `TooFewFieldsCollector` class call the constructor of superclass `CodeSmellCollector` with the same parameters, an `ICompilationUnit` being a Java source code file and a `CompilationUnit` being the root of an abstract syntax tree. This abstract syntax tree can be obtained

by passing it as a parameter to the `AST.parseCompilationUnit` method in the Eclipse API. The reason we need both the Java source code and the abstract syntax tree is that we need to connect objects in the abstract syntax tree with locations in the Java source code, for example to enable the user to jump to the location of interest in the source code by clicking on a code smell marker in the Tasks View.

```
/**
 * Constructor for TooFewFieldsCollector.
 */
public TooFewFieldsCollector() {
  super();
}

/**
 * Constructor for TooFewFieldsCollector.
 * @param icu java source file
 */
public TooFewFieldsCollector(ICompilationUnit icu) {
  super(icu);
}

/**
 * Constructor for TooFewFieldsCollector.
 *
 * @param icu java source file
 * @param cu abstract syntax tree
 */
public TooFewFieldsCollector(ICompilationUnit icu,
                             CompilationUnit cu) {
  super(icu, cu);
}

/**
 * Constructor for TooFewFieldsCollector with certain
 * minimum number of fields.
 *
 * @param icu java source file
 * @param cu abstract syntax tree
 * @param minFields minimum number of fields
 */
public TooFewFieldsCollector(
  ICompilationUnit icu,
  CompilationUnit cu,
```

```
    int minFields) {
    super(icu, cu);
    this.setMinFields(minFields);
  }
```

The real work is done by overriding one of the `visit` methods inherited from `ASTVisitor`. In the case of the Too Few Fields collector, we are obviously interested in the number of fields a class declares. Therefore, we override the `visit` method for the AST node type `TypeDeclaration`. Interfaces are skipped, because they only contain constant fields. If the class is not an interface, the number of fields that are declared in this type declaration is compared to the minimum set for this `TooFewFieldsCollector` object. If `node.getFields().length` is lower, a code smell is reported. The value `false` is returned, because there is no need to visit the tree any further, because only top level classes are of interest.

```
  /**
   * Override for the visit method for TypeDeclarations in
   * ASTVisitor, check the number of fields and report a code
   * smell if necessary.
   * @param node the node to visit
   * @return <code>true</code> if child nodes have to be
   * visited.
   */
  public boolean visit(TypeDeclaration node) {
    if (!node.isInterface()) {
      if (node.getFields().length < getMinFields()) {
        reportSmellAtOffset(node.getName().getStartPosition(),
            "Too Few Instance Variables found.");
      }
    }
    return false;
  }
```

It is considered good practice to create getter and setter methods for all fields, in this case `_minFields`.

```
  /**
   * Get minimum number of fields.
   * @return minimum number of fields
   */
  public int getMinFields() {
    return _minFields;
  }
```

```
/**
 * Set minimum number of fields.
 * @param minFields new minimum number of fields
 */
public void setMinFields(int minFields) {
  _minFields = minFields;
}
```

Finally, the abstract `collectSmells` method from `CodeSmellCollector` has to be implemented. Most code smell collector classes call the `findSmellsInAST` method inherited from `CodeSmellCollector` with the abstract syntax tree passed as a parameter.

```
/* (non-Javadoc)
 * @see org.eclipse.contribution.codenose.
     codesmells.CodeSmellCollector#collectSmells()
 */
public int collectSmells() {
  return findSmellsInAST(compunit);
}
```

5.6.2 ClassSizePreferencePage

All preference pages and related classes are located in one package. The preferences for all class size code smells (too few or too many methods/fields) are bundled on one preference page.

```
package org.eclipse.contribution.codenose.preferences;
```

The import statements can be produced by Eclipse using the Organize Imports feature again, just like was done for the code smell collector class.

```
import org.eclipse.contribution.codenose.CodeNosePlugin;
[...]
```

After the main class comment, the class declaration starts. This preference page is a subclass of `FieldEditorPreferencePage`, one of the standard preference page types in Eclipse, and implements the `IWorkbenchPreferencePage` interface.

```
public class ClassSizePreferencePage
  extends FieldEditorPreferencePage
  implements IWorkbenchPreferencePage {
```

The constructor for this class calls the constructor of its superclass with a parameter that determines the layout style for this page. An `IPreferenceStore` is created to remember our preferences.

```
/**
 * Constructor for the Preference Page.
 */
public ClassSizePreferencePage() {
  super(FieldEditorPreferencePage.GRID);
  IPreferenceStore store =
      CodeNosePlugin.getDefault().getPreferenceStore();
  setPreferenceStore(store);
}
```

The preference page is filled with text and field editors by implementing the abstract method `createFieldEditors` that is inherited from `FieldEditorPreferencePage`. The field editors for the Too Few Fields code smell are created in the `addMinFieldsFields` method.

```
/* (non-Javadoc)
 * @see org.eclipse.jface.preference.
     FieldEditorPreferencePage#createFieldEditors()
 *
 * Create the fields on the Preference Page.
 */
protected void createFieldEditors() {
  addMinMethFields();
  addMaxMethFields();
  addMinFieldsFields();
  addMaxFieldsFields();
}
```

Here is the declaration of that method. Fields are added to the grid layout of this preference page by calling the `addField` method. A `BooleanFieldEditor` (a checkbox) is created so plug-in users can enable or disable detection of this code smell. The minimum number of fields can be set in an `IntegerFieldEditor`, a text input box where a number must be entered. The range of valid values for the minimum is set to 1 through 9. Note that the `IntegerFieldEditor` is added to the preference page *after* its valid range has been set.

```
/**
 * Add fields for the Too Few Instance Variables
 * code smell to the Preference Page.
 */
private void addMinFieldsFields() {
  addField(new BooleanFieldEditor("MinFieldsActive",
```

```
        "Detect Too Few Instance Variables Codesmell",
        getFieldEditorParent()));
    IntegerFieldEditor minFields = new IntegerFieldEditor(
        "MinFieldsMin",
        "Minimum number of instance variables",
        getFieldEditorParent());
    minFields.setValidRange(1,9);
    addField(minFields);
  }
```

Finally, the `IWorkBenchPreferencePage` interface requires the implementation of the `init(IWorkbench)` method, but no initialization is necessary.

```
  /* (non-Javadoc)
   * @see org.eclipse.ui.IWorkbenchPreferencePage#init(
       org.eclipse.ui.IWorkbench)
   */
  public void init(IWorkbench workbench) {
  }
```

5.6.3 Plug-in manifest

For the created page to actually show up in the Eclipse main menu under Window→Preferences pages, the preference page has to be put in an extension contributed to the workbench's preference pages extension point (`org.eclipse.ui.preferencePages`) in the plug-in's manifest file. The page name XML tag determines the name that will appear in the preference pages tree. `<page class>` is the class that was created in the previous subsection.

```
<?xml version="1.0" encoding="UTF-8"?>
<plugin

  [... general plug-in info, runtime library, required
   resources, pop up menu extensions for the UI, marker
   extension for code smell reports in the Tasks view,
   ...]

  <extension point="org.eclipse.ui.preferencePages">

    [... other preference pages ...]

    <page
       name="Class Size Preferences"
       category="org.eclipse.contribution.codenose.
           CodeNosePreferencePage"
```

```
      class="org.eclipse.contribution.codenose.
          preferences.ClassSizePreferencePage"
      id="org.eclipse.contribution.codenose.classsizepref">
    </page>

    [... even more preference pages ...]

  </extension>
</plugin>
```

5.6.4 Plug-in defaults

For every preference in the CodeNose plug-in, a default value is set in the `CodeNosePlugin` class. The source code lines that set the default values for the preferences for the Too Few Fields code smell are:

```
public IPreferenceStore getPreferenceStore() {
  [...]
  super.getPreferenceStore().setDefault(
      "MinFieldsActive", true);
  super.getPreferenceStore().setDefault(
      "MinFieldsMin", 1);
  [...]
}
```

5.6.5 CodeNoseUtil

The final step in adding a new code smell is to implement the start of the code smell collector and make it use the preferences set by the user of the plug-in. Starting code smell detection in the case of the `TooFewFields-Collector` is done in the `findPrimitiveSmells` method of the `CodeNoseUtil` class. The stats fields are meant to be used for a report dialog after the detection process has ended. If the detection of Too Few Fields code smells is enabled by marking the checkbox on the preference page (the `MinFieldsActive` preference), a `TooFewFieldsCollector` object is created to search the Java source code file passed as a parameter to `findPrimitiveSmells`. Every code smell collector with more preferences than just the Active preference has its own `setPreferences` method. For the `TooFewFieldsCollector`, this method sets the minimum number of fields to the value set on the Class Size preference page.

```
public static void findPrimitiveSmells(
  ICompilationUnit icu) {

  int switchStats, longParStats, longMethStats,
```

```
    msgChainStats, lazyClassStats, emptyCatchStats,
    minMethStats, maxMethStats, minFieldStats,
    maxFieldStats;

  CompilationUnit compunit =
      AST.parseCompilationUnit(icu, true);

  [...]
  if (getBooleanPreference("MinFieldsActive")) {
    TooFewFieldsCollector tffc =
        new TooFewFieldsCollector(icu,compunit);
    setPreferences(tffc);
    minFieldStats = collectCodeSmells(tffc);
  }
  [...]
}

/**
 * Set a TooFewFieldsCollector's parameters to the
 * preferences set on its Preference Page.
 *
 * @param tffc the Too Few Fields Collector to be
 *   initialized
 */
public static void setPreferences(
  TooFewFieldsCollector tffc) {

  tffc.setMinFields(
    CodeNoseUtil.getIntPreference("MinFieldsMin"));
}
```

After these steps, the new code smell has been added to the CodeNose plug-in. Simple primitive smells like the Class Size smells can be added in just a few minutes. These five steps may seem like a lot of work, but much can be copy/pasted from existing code smells (the Monkey-See-Monkey-Do rule [3]).

Chapter 6

Case Study

To test the performance of the CodeNose plug-in and the usefulness of the detected code smells, a case study was performed using the source code of an existing open source software system of reasonable size, JHotDraw[1]. It is a Java graphical user interface framework for technical and structured graphics, and was originally designed by Erich Gamma and Thomas Eggenschwiler as a design exercise for patterns. For this case study, version 5.3 of JHotDraw was used as subject system.

6.1 Performance

The CodeNose plug-in has to be able to perform the detection of code smells as fast as possible. The more time it takes to find all code smells in the Java source code, the less inclined the user will be to use the plug-in on a regular basis, for example every time he or she 'switches hats' from adding a new function to the software system to refactoring the existing code [11].
Of course, the performance of the CodeNose code smell detection plug-in depends on the size of the software system it is analyzing. To get an idea of what the results of the case study mean, we have to first determine what the size of the JHotDraw software system is. We used version 1.3.4 of the Metrics[2] plug-in for Eclipse to perform this task. The main size metrics for JHotDraw are:

- Lines of code: 8353
- Number of packages: 11
- Number of classes: 208
- Number of attributes: 380

[1] `http://www.jhotdraw.org`
[2] `http://metrics.sourceforge.net`

- Number of static attributes: 107
- Number of methods: 1896

The plug-in was tested in three different setups, two on the same personal computer. The first PC is a relatively old machine, a Pentium III desktop computer with a CPU speed of 1.0 GHz and 256 Mb RAM memory. It was used to test the performance of the CodeNose plug-in under the Windows 98SE operating system and under a Linux distribution, Fedora Core 1. The third test was performed on a more modern personal computer. This machine was running the Windows XP operating system and had a 3.00 GHz CPU and 512 Mb RAM memory. All code smell parameters were kept the same during the performance tests, using the plug-in's default values.
The performance tests consisted of running a complete code smell detection process on the whole JHotDraw project. The tests were performed five times in a row, in order to investigate if there were any significant differences in process duration, perhaps due to caching or loading. In each test configuration, the first test that was performed was somewhat slower than all subsequent tests. This is almost certainly due to the fact that Eclipse does not load a plug-in until it really needs it. Therefore, each time the first test was performed, some additional time was needed to load the plug-in ahead of performing the plug-in task. The performance test results are in Table 6.1.

Configuration	*test 1*	*test 2-5*
1.0 GHz, 256 Mb PC - Windows 98SE	61s	51s
1.0 GHz, 256 Mb PC - Linux	80s	72s
3.0 GHz, 512 Mb PC - Windows XP	21s	17s

Table 6.1: Performance Test Results

6.2 Playing with parameters

Good performance means very little if the results are not useful to the user who requested them. Depending on the size of the software system being analyzed, some code smells might be of more use than others. Setting the wrong parameters would probably mean having to do the detection over and over again until the user gets the results he or she wanted. We tested the CodeNose plug-in on the JHotDraw source code smell-per-smell, each time with different parameters, to find out what parameter values and what code smells are useful for this particular software system. The following subsections present an overview of the test results for each code smell.

Java file	*method*	*switch param.*
TriangleFigure	`getPolygon`	`fRotation`
TriangleFigure	`connectionInsets`	`fRotation`
ShortestDistanceConnector	`findPoint`	`i`
StandardDrawingView	`handleCursorKey`	`key`
GraphLayout	`relax`	`REPULSION_TYPE`
StorableOutput	`writeString`	`c`

Table 6.2: Switch statements found in JHotDraw

6.2.1 Switch Statement

The Switch Statement code smell does not offer a parameter to play with, so we will just show the six switch statements found in Table 6.2. The problem with switch statements, according to Fowler's Refactoring book [11], is duplication of code. You will often see the same switch statement in different places in the source code. But six switch statements in thousands of lines of code is not a bad result. Only two switch statements switch on the same expression, the `fRotation` field in the `TriangleFigure` class. Only one of the switch statements switches on type. Another thing that attracts attention is the large size of the `findPoints` and `relax` methods. These switch statements could be good candidates for an Extract Method refactoring, which could be used to reduce the size of the methods they are in. Performing the Replace Conditional with Polymorphism refactoring on the new method will eliminate the switch statement altogether [11].

6.2.2 Long Method

The Long Method code smell has a parameter to set the maximum number of lines of code, as well as parameters to determine what exactly comprises a line of code, i.e. whether lines with comments, method header lines or lines with nothing but opening and closing braces or white space should be counted. We set the maximum to 25 lines of code, so any method having more than 25 lines of code will show up in the Tasks view as a Long Method code smell. First, we started the plug-in with all options for counting additional lines of code disabled. Then we gradually enabled one option at a time, in order to see how this changed the results (see Table 6.3).

A few methods really jump out when it comes to size, but most of the detected Long Method code smells concern methods that are just a bit over the maximum. Almost all methods in the JHotDraw source code have one-line headers, so that option did not change the number of code smells that were detected by the CodeNose plug-in. Adding comments and white space did have an effect on the results, but not on the top five of longest methods, shown in Table 6.4. For all parameter settings possible, the same five

LOC count parameters	*smells found*
no braces, headers, comments, white space	21
with braces	37
with braces and headers	37
with braces, headers and comments	53
with braces, headers, comments and white space	68

Table 6.3: Long Method code smells in JHotDraw

class	*method*	*LOCs-*	*LOCs+*
`GraphLayout`	`relax`	76	102
`ShortestDistanceConnector`	`findPoint`	74	102
`Bounds`	`cropLine`	71	108
`DragNDropTool`	`drop`	60	80
`DrawApplet`	`createButtons`	45	77

Table 6.4: Top 5 longest methods in JHotDraw

methods show up in this top five, albeit sometimes in a different order. In Table 6.4, the *LOCs-* column shows the number of lines with all options for counting additional lines disabled, and the *LOCs+* column shows the number of lines with all parameters for white space, comments, braces and header lines enabled.

6.2.3 Long Parameter List

With the maximum set to six, we found just one Long Parameter List code smell in JHotDraw. This method was the `intersect` method in the `Geom` class, which calculates the intersection point of two lines. Its eight parameters are the co-ordinates of two lines, so using the Introduce Parameter Object [11] refactoring might be a good idea, especially if this data clump of four co-ordinates appears elsewhere in the source code as well.
With the maximum lowered to five, we find three more code smells. Two of these methods with six parameters are in the same `Geom` class and deal with lines as well (`distanceFromLine` and `lineContainsPoint`). If we lower the maximum number of parameters further, the number of detected code smells really explodes, to 41 in total with the maximum at four and sixty methods with the maximum number of parameters at three.

6.2.4 Too Few Fields

The CodeNose plug-in found 43 classes in the JHotDraw source code that contain no field declarations at all. This code smell is intended to find classes that don't do enough to justify their existence, like Middle Man classes, Lazy Classes or maybe even the Speculative Generality code smell, as described

class function	*occurrences*
Handle	14
Command	13
Tool	6
App(let)	3
Exception	2
Adapter	1
Connection	1
Figure	1
Format	1
Multicaster	1

Table 6.5: Class functions with 0 fields in JHotDraw

by Fowler [11].
JHotDraw makes heavy use of design patterns [12], e.g. the Command pattern. It also uses a lot of small classes that represent (part of) an object in the drawing figures or that perform a small task. This function can easily be recognized by looking at the last part of the class name. In some cases, the function of a class is the cause for the detection of a code smell in that class. For example, Command classes in the Command pattern do not have any fields and the Handle classes inherit some fields from their parent classes. Therefore, most of the code smells that were detected can be ignored, based on their function in the JHotDraw code. Table 6.5 presents a list of the last parts of the names of the 43 classes with no fields of their own.

6.2.5 Too Many Fields

Classes with a lot of fields tend to be very large, making them hard to understand and maintain. Identifying a group of instance variables that go together well and that often appear in the same methods, and then performing an Extract Class refactoring can solve this issue [11]. Another option is to find a cluster of variables that could go into a subclass. There are a few classes in JHotDraw with an exceptionally high number of fields, as shown in Table 6.6.

The classes with the highest number of fields (`DrawApplet`, `DrawApplication` and `StandardDrawingView` with eighteen) are indeed very large. But these classes also show a weakness of the implementation of this code smell: because it uses a call to `getFields().length` on a `TypeDeclaration` node, it does not discriminate between instance variables and constants (with modifiers final static). `DrawApplet` has only three constants and `DrawApplication` has eight, so there are still at least ten instance variables in these classes.

maximum	*code smells*
4	26
5	16
6	9
7	8
8	6
9	5
10	4

Table 6.6: Too Many Fields code smells in JHotDraw

6.2.6 Too Few Methods

The good news is: there were no classes found in JHotDraw that have no methods at all. Of course, concrete and abstract classes should have at least one constructor, and interfaces without any methods could be eliminated. The CodeNose plug-in found fifteen classes with just one method. Eight of these classes are interfaces, two are small classes that are used and live in another class' Java source code file. The five remaining classes are normal object classes. Of these five, two are subclasses of the `DrawApplet` class, two are implementations of one of the eight interfaces with just one method, and the last class is a subclass of `AbstractTool`, which is a very small class as well. When the minimum number of methods was raised to three, the plug-in found 39 Too Few Methods code smells.
Some of the classes with just one or two methods might be the result of a design that looked great on paper. But every class costs time and effort to maintain and understand, so maybe some of these classes could be eliminated using the Collapse Hierarchy and Inline Class refactorings [11].

6.2.7 Too Many Methods

Classes with a large number of methods are good Large Class code smell candidates. Of course, there are classes that can't help having a lot of methods, for example the `ASTVisitor` class, that needs to have `visit` and `endVisit` methods for every possible node type in abstract syntax trees. But these classes are the exceptions, and we only conclude that there is a smell, not that there necessarily is a problem. The distribution of classes with too many methods can be found in Table 6.7.

The class with the highest number of methods is `DrawApplication`, which is indeed a huge class. The two classes in the 50-70 methods range are `StandardDrawingView`, which is simply a large class because it is the implementation of the main drawing view in JHotDraw, and `NullDrawingView`, which has loads of one-line methods and even methods that contain no statements at all, but that only contain a comment stating 'ignore: do nothing'.

maximum	*code smells*
10	60
20	23
30	16
40	7
50	3
70	1
75	0

Table 6.7: Too Many Methods code smells in JHotDraw

min avg LOCs	*code smells*
1.0	4
1.01	9
1.1	9
1.2	9
1.3	14
1.4	18
1.5	19
2.0	20

Table 6.8: Lazy Classes in JHotDraw

`NullDrawingView` is used instead of a null reference to avoid null pointer exception in the Null-value object bug pattern.

6.2.8 Lazy Class

A Lazy Class code smell is reported when the average number of lines of code per method is lower than a minimum set in the preferences. The purpose of this code smell is to catch those Lazy Class, Data Class and Middle Man [11] classes that are not detected by the code smell collectors for too few methods or too few fields. These classes have enough fields and methods, but don't do enough to justify the time and costs of maintaining them. They are just dumb data holders or data indirectors. An overview of the distribution of these Lazy Classes and their average number of lines of code per method is in Table 6.8.

There are apparently four classes in the JHotDraw source code that actually have more methods than lines of code. These classes are `ChopPolygonConnector` (two constructors and one normal method, two lines of code in total), `FigureChangeAdapter` (five methods which are all empty), `NullDrawingView` (many methods with one or no lines of code) and `ClipBoard` (a small class, replacement for the 'global' clipboard, appears useful).

We were particulary interested in classes with exactly one line per method,

max length	*code smells*
4	0
3	2
2	48
1	692

Table 6.9: Message Chains in JHotDraw

because that is the trademark of the proverbial Middle Man class. The CodeNose plug-in found five classes with an average of 1.0 lines of code per method. The code smells found for parameter values of 1.3 and higher usually concern classes that have a lot of one-line methods and a small number of methods with some more body volume.

6.2.9 Message Chain

As we discussed in Subsection 5.4.5, the Law of Demeter is a guideline that states that objects should not have structural knowledge of any object other than itself. Avoiding message chains is not enough to abide by this principle, but future versions of CodeNose could include a more general implementation of this rule, for example preventing the use of temporary variables that actually serve as a message chain in disguise.
Message Chain code smells, objects asking for other objects, show up in source code as a long line of get methods. This makes the whole chain of objects depend on each other. A change anywhere in the chain means this code has to be changed as well. The best way to get rid of these dependencies, according to Fowler [11], is to use Hide Delegate or to see whether you can use Extract Method to take out a piece of code and then move it down the chain with the Move Method refactoring. Therefore, our attention in this part of the case study, apart from the distribution of Message Chain code smells, should be on whether we can identify (pieces of) message chains that appear quite frequently.
The distribution of code smells is presented in Table 6.9. In this table, a *max length* of two means that `getThis().getThat()` is allowed, but adding `.getSomeMore()` to the chain would result in a code smell. For a software system the size of JHotDraw, two is apparently the smallest value for the maximum message chain length parameter that gives an amount of results that a user might want to explore.
The two longest message chains found, with a length of four methods, are both in the same `invokeEnd` method in UndoableHandle.java, and they both start with `getDrawingView().editor().getUndoManager()`. Browsing through the message chains found with the maximum length set to two, you will find a lot of similarities. Five of them are in the `DrawApplication` class:

- two start with `view().drawing().setTitle`

- three start with `view().drawing().getTitle`, including one in a `getDrawingTitle` method that is not referenced anywhere in JHotDraw.

Other message chains that show up a few times in the `ch.ifa.draw.figures` package:

- `getDrawingView().drawing().orphan`
- `getDrawingView().drawing().orphanAll`
- `getDrawingView().drawing().add`
- `getDrawingView().drawing().addAll`

The same dependency can be found several times in the other large packages in JHotDraw, the `standard` and the `util` package, which contains five more message chains starting with `getDrawingView().editor().getUndoManager()`. These results are clearly useful for a user who is trying to bring down the number of dependencies in his source code, even though finding similarities between the Message Chain code smells still requires some human labour.

6.2.10 Empty Catch Clause

With the options for empty statements and comments disabled, five Empty Catch Clause smells were found in the JHotDraw source code. We did not detect any empty statements in catch clauses that were otherwise empty. Two of the empty catch clauses contain a comment, so three code smells remain if the option to skip commented catch clauses is enabled.
Three of the empty catch clauses are used in a scheme to wait for an interruption and a fourth empty clause catches a `java.beans.PropertyVetoException` but does not handle it. For the last empty catch clause, there was no comment inside the clause, nor could be determined by examining the try block what exception could be thrown by which statement in the block.

6.2.11 Refused Bequest

The CodeNose plug-in found twenty-two Refused Bequest code smells in the JHotDraw source code. Careful examination of the classes that are involved sheds some light on what caused the reporting of these code smells. Most of the Refused Bequest classes can be grouped together:

- `ChopDiamondConnector`, `ChopEllipseConnector` and `PolyLineConnector` are all subclasses of `ChopBoxConnector`. The `ChopBoxConnector` object has three methods: `chop`, `findStart` and `findEnd`. The three code smell classes all override the `chop` method and leave the other two methods unused.

- `PolyLineHandle`, `FontSizeHandle` and `NullHandle` are all subclasses of `LocatorHandle`, a class with two methods (`locate` and `getLocator`) which are not used by these Refused Bequest classes. The `NullHandle` class adds only one method to these two, the `draw` method. Another class that caused a Refused Bequest smell, `GroupHandle`, is a subclass of `NullHandle`, but it overrides the `draw` method that it inherits from `NullHandle`.

- In BoxKitHandle.java, the `ResizeHandle` class is another class that extends `LocatorHandle` without using the two methods in that class. The BoxKitHandle java source file contains eight more small classes (`NorthHandle`, `NorthEastHandle`, `EastHandle`, etc.) that are subclasses of this `ResizeHandle` class, but again without using the methods that were inherited from `ResizeHandle`.

- Two of the Refused Bequest classes, `ElbowTextLocator` and `PolyLineLocator`, are subclasses of `AbstractLocator`. This `AbstractLocator` class is a very minimal abstract implementation of the Locator interface, containing only a `clone` method and two methods (`read` and `write`) with an empty body. Neither the `clone` method nor the empty methods are used by the two subclasses. A third subclass of `AbstractLocator`, `RelativeLocator`, is polite enough to call `AbstractLocator`'s empty `read` and `write` methods in its overriding implementations of these methods.

Four Refused Bequest classes remain that do not fall into one of these groups. `GraphLayout` extends `FigureChangeAdapter`, but that class is an empty implementation of the `FigureChangeListener` interface. The `NullTool` class takes its name seriously: it contains nothing more than a constructor. `PertFigureCreationTool` has a method that returns a new `PertFigure` object, without using anything from its parent `CreationTool`. And finally, the `BorderTool` class is an extension of the abstract `ActionTool` class. It implements the one abstract method its parent has, but does not use any of the other methods in `ActionTool` in the process.
Most of the Refused Bequest code smells have to do with subclasses of parents that have just a small bequest to offer. If this is the case, the actual problem lies in the parent class instead of the child class. In other cases, the Refused Bequest classes are subclasses that only want to override its parents' methods, or subclasses that only add one or two methods to the ones inherited from their parents, without using any of the methods in the parent class. If a parent class is quite large and contains a lot of methods that are not used by any of its subclasses, the solution would be to split the parent into two, keeping only the common code in the parent class while pushing all the unused methods into a new sibling.
In a few cases, the cause of the Refused Bequest seems to be that a design or

max	code smells
10	6
9	6
8	10
7	12
6	15
5	24

Table 6.10: Feature Envy code smells in JHotDraw

envying method	*envied class*	*for. uses*
`GraphLayout.relax`	`GraphNode`	26
`DrawApplication.createEditMenu`	`CommandMenu`	17
`DragNDropTool.setCursor`	`RelativeLocator`	16
`FigureAttributes.write`	`StorableOutput`	14
`DragNDropTool.setCursor`	`DrawingView`	12
`FigureAttributes.read`	`StorableInput`	11

Table 6.11: Top Feature Envy methods in JHotDraw

design pattern prescribes the existence of a class, but not much code could be put into the actual implementation of the class. But putting all common code in a parent class, while pushing code that is different in some cases into subclasses, is usually not considered bad practice. Fowler even states: "'Nine times out of ten this smell is too faint to be worth cleaning."' [11]

6.2.12 Feature Envy

Feature Envy code smells are reported on methods that seem to have a lot of interest in another class than the one they live in. The user can set the maximum amount of calls to methods and uses of fields from that other class that he or she considers acceptable. For different values of this maximum, the number of code smells found in the JHotDraw source code is in Table 6.10. Using ten methods or fields from another class seems like a lot. A closer look at these code smell methods seems appropriate. The top six Feature Envy methods are listed in Table 6.11. `GraphNode` is a small class in the same Java source file as `GraphLayout`. Tool or utility classes like menu objects and input/output classes are usually called a lot by methods in other classes. The `setCursor` method in the `DragNDropTool` class shows up twice in this top six. That is because it contains code that sets a new cursor type for the view for all eight wind directions, which we already encountered in the discussion on the Refused Bequest code smell.
As is the case with most code smells, its occurrence does not necessarily imply that there is a problem with the code. For example, classes that can

class	*code smells*
`DrawApplication`	14
`DragNDropTool`	9
`GraphLayout`	6
`UngroupCommand`	6
`ElbowConnection`	5
`GroupCommand`	5
`SendToBackCommand`	5
`StandardDrawingView`	5
6 classes	4

Table 6.12: Top code smell classes in JHotDraw

be used to create a menu can be the cause of detection of a Feature Envy code smell, because its methods have to be called quite a number of times to define the menu. Only after investigation of the method body in which the Feature Envy code smell occurs can we decide if there really is some code in there that really wants to live elsewhere. The function of the class that is called the most in that body is usually a good indication for the severity of the problem. A future version of CodeNose could include an ignore list for certain classes for this code smell. Not counting the occurrences of calls to menu and other utility classes could bring down the number of 'false positives'.

6.2.13 Places of putrefaction

When deciding where to begin the refactoring of a software system, the class that contains the most code smells is often seen as the best place to start. Therefore, we have listed the classes that produced the highest number of code smells in a detection process with all parameters at their default values in Table 6.12.

Actually, `BoxHandleKit.java` is the resource that produced the most code smells (20), but this Java file really contains ten classes, including the wind direction Handle classes. Nine of these classes cause the same two code smells to be reported: Refused Bequest and Too Few Fields.

6.2.14 Coincidence of code smells

After investigation of the distribution of code smells found in the JHotDraw software system, we can draw the following conclusion: the size and function of a class often determine which code smells can be found in that class. There seem to be large class smells and small class smells, which tend to show up together in groups. The code smells that were often found in large classes are Too Many Methods, Too Many Fields, Long Method and Feature

Envy. Their small class counterparts are Lazy Class, Too Few Fields, Too Few Methods and Refused Bequest. The function of a class also has a big influence on the type of code smell that can be found in that class. Almost all of the Command classes in JHotDraw contain Message Chain code smells and have Too Few Fields. The Handle classes tend to be Lazy Classes or Refuse Bequest classes, and they also have few fields. Most of the Figure classes have Too Many Fields and/or Methods. There are of course exceptions to the rule: in the Tool classes group, `DragNDropTool` is a huge class with Long Methods, Too Many Methods and Too Many Fields, but `PertFigureCreationTool` is lazy, refuses bequest and has no fields at all.
The detection of Empty Catch Clause and Long Parameter List code smells seems to be unrelated to size or function of the class it is in. While the two largest methods in JHotDraw both contain a switch, the remaining switch statements found in the source code are in methods of medium or small length.

Chapter 7

Conclusions

The main goal for this project was to develop a prototype of an Eclipse plug-in for the detection and presentation of code smells in Java source code, aimed at providing feedback of a system's quality to software programmers during software development. The CodeNose plug-in, a code smell detection tool integrated into the Eclipse framework, was developed for this purpose. A set of code smells was identified specifically for the purpose of aiding the software developer. Our goal was to implement the detection of code smells in separate modules, keeping business logic separated from the user interface. The user should be able to direct the code smell detection process by setting parameters and making a selection from the available code smells. Adding new code smells to the plug-in's set of detected code smells should be easy. The user should also be able to filter the detection process results, or export them to a text file for use in other applications or reports. Installing the plug-in had to be easy as well.
The Eclipse CodeNose plug-in uses the Tasks view to present its results. This view can be used to browse all detected code smells and to immediately jump to the location of the individual code smells in the Java source code to start refactoring in the Eclipse Java editor. The Tasks view's contents can be filtered and/or sorted, and code smells can be exported to a text file. The code smell detection classes have no knowledge of the inner workings of the user interface modules, and vice versa. Code smells that were removed from the Tasks view but still exist in the code will reappear when the user starts a new detection process. In a future version of CodeNode, a list of removed smells could be added to the preferences. This list will prevent the code smells from reappearing in the Tasks view, unless the user decides to reinstate them.
The user can choose which code smells should be detected by the plug-in, and in some cases the user can set parameters for the detection, e.g. maximum length of a method, by changing the values on a preference page in the Eclipse menu system. As we discussed in Section 5.6, the use of preference

pages does imply that adding a new code smell to the detection process requires the creation or editing of five files. This is a trade-off between extensibility and parametrization. We chose ease of use and good modularity over hard-coded parameters and fast addition of new code smells for people who know the chosen scripting language. A future version of CodeNose could include a mechanism to facilitate the addition and exchange of code smells using wizards and JAR-files.
The performance of the CodeNose plug-in for a software system of the size of JHotDraw seems reasonable. Changing the way the plug-in works, from the on-call basis it uses now to constant monitoring of projects using ActionListeners (like for example the Metrics tool), would improve performance considerably. Compilation of classes to obtain the abstract syntax trees is the task that takes up most of the time. The plug-in could profit from the fact that compilation of classes is taking place anyway when changes to a source file are saved. It would then only have to respond to the changes made in the classes involved and re-evaluate the 'global' code smells, like Feature Envy and Refused Bequest.
The CodeNose plug-in was developed for version 2.1.3 of Eclipse. The most recent version of Eclipse right now is 3.0.1, and version 3.1 will probably be available within a few months. For the CodeNose plug-in to have a serious chance of growing from a research prototype to a popular software development tool, development of a CodeNose version for Eclipse 3 is necessary.
When this thesis project was initiated, no general code smell detection tool that was integrated into an IDE was available, but nowadays a few of these tools are available. PMD is an open source project at SourceForge.net, has plug-ins for most of the popular integrated development environments and can detect a huge number of problems. Some commercial alternatives are available as well, each with their own set of rules or detected problems.

Bibliography

[1] A.W. Appel. *Modern Compiler Implementation in C: Basic Techniques.* Cambridge University Press, 1997.

[2] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Proceedings 13th Australian Software Engineering Conference (ASWEC 2001)*, pages 68–75, 2001.

[3] K. Beck and E. Gamma. *Contributing to Eclipse.* Addison-Wesley Professional, October 2003. June 2003 preliminary version was used for this report.

[4] K.H. Bennett and V.T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 73–87, June 2000.

[5] I.T. Bowman, R.C. Holt, and N.V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. of the International Conference on Software Engineering (ICSE-21)*, May 1999.

[6] W.J. Brown, R.C. Malveau, H.W. McCormick III, and Th.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley & Sons Inc., 1998.

[7] W. Crawford and J. Kaplan. *J2EE Design Patterns*, chapter 12. O'Reilly & Associates, 2003.

[8] S. Demeyer, S. Ducasse, and M. Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In *Proceedings 6th Working Conference on Reverse Engineering (WCRE'99)*, pages 175–186, October 1999.

[9] B. Dudney, S. Asbury, J. Krozak, and K. Wittkopf. *J2EE AntiPatterns.* Wiley Publishing, Inc., August 2003.

[10] G. Florijn. RevJava - Design critiques and architectural conformance checking for Java software. White paper, SERC, the Netherlands, 2002.

[11] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[13] S. Grant and J.R. Cordy. An interactive interface for refactoring using source transformation. In *Proceedings REFACE03, WCRE Workshop on Refactoring: Achievements, Challenges, Effects*, November 2003.

[14] R.C. Holt. Structural manipulations of software architecture using Tarski relational algebra. In *Proc. 5th Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219, 1998.

[15] S.C. Johnson. Lint, a C program checker. In *Unix Programmer's Manual*, volume 2A, chapter 15, pages 292–303. Bell Laboratories, 1978.

[16] J. Kerievsky. *Refactoring to Patterns.* Addison-Wesley Professional, 2004.

[17] H.A. Müller, J.H. Janke, D.B. Smith, M.-A. Storey, S.R. Tilley, and K. Wong. Reverse engineering: A roadmap. *22nd International Conference on Sofware Engineering - Future of Software Engineering Track*, pages 47–60, 2000.

[18] T. Systä, P. Yu, and H. Müller. Analyzing Java software by combining metrics and progam visualization. In *Proc. 4th European Conference on Software Maintenance and Reengineering*, 2000.

[19] S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, pages 501–520, December 1994.

[20] M.G.J. van den Brand et al. The ASF+SDF Meta-Environment: a component-based language development environment. In *Proc. Compiler Construction 2001*, pages 265–370, 2001.

[21] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proc. 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering*, May 2001.

[22] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proc. 9th Working Conference on Reverse Engineering.* IEEE Computer Society Press, October 2002.

[23] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley Publishing Company, 1995.