Java Quality Assurance by Detecting Code Smells

Eva van Emden[*]
CWI
The Netherlands
http://www.cwi.nl/∼eva/
eva@cwi.nl

Leon Moonen[‡]
CWI
The Netherlands
http://www.cwi.nl/∼leon/
leon@cwi.nl

Abstract

Software inspection is a known technique for improving software quality. It involves carefully examining the code, the design, and the documentation of software and checking these for aspects that are known to be potentially problematic based on past experience.

Code smells are a metaphor to describe patterns that are generally associated with bad design and bad programming practices. Originally, code smells are used to find the places in software that could benefit from refactoring. In this paper, we investigate how the quality of code can be automatically assessed by checking for the presence of code smells and how this approach can contribute to automatic code inspection.

*We present an approach for the automatic detection and visualization of code smells and discuss how this approach can be used in the design of a software inspection tool. We illustrate the feasibility of our approach with the development of j*COSMO*, a prototype code smell browser that detects and visualizes code smells in* JAVA *source code. Finally, we show how this tool was applied in a case study.*

Keywords: *software inspection, quality assurance, Java, refactoring, code smells.*

1. Introduction

Software inspection is a known technique for improving software quality. It was first introduced in 1976 by Fagan [10] and has since been reported on by numerous others, for example [18, 13]. Software inspection involves carefully examining the code, the design, and the documentation of software and checking these for aspects that are known to be potentially problematic based on past experience.

It is generally accepted that the cost of repairing a bug is much lower when that bug is found early in the development cycle. One of the advantages of software inspection is that the software is analysed *before* it is tested. Thus, potential problems are identified in the beginning of the cycle so that they can be solved early, when it's still cheap to fix them.

Traditionally, software inspection is a formal process that involves labor-intensive manual analysis techniques such as formal code reviews and structured walk-throughs. Inspection is a systematic and disciplined process that is guided by well-defined rules. These strict requirements often backfire, resulting in code inspections that are not performed well or sometimes even not performed at all.

These problems are addressed by tools that automate the software inspection process. We distinguish two approaches:

1. Tools that *automate* the inspection *process*, making it easier to follow the guidelines and record the results.
2. Tools that perform *automatic code inspection*, relieving the programmers of the manual inspection burden.

We concentrate on the second type: tools that perform automatic inspection. Such tools are interesting since automatic inspection and reporting on the code's quality and conformance to coding standards allows early (and repeated) detection of signs of project deterioration. Early feedback enables early corrections, thereby lowering the development costs and increasing the chances for success.

1.1. Code smells

The existing tools that support automatic code inspection (for example, the well-known C analyzer LINT [15]) tend to focus on improving code quality from a technical perspective. The fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code. From this perspective, code inspection boils down to low-level bug-chasing and we see this reflected in the tools which typically look for problems with pointer arithmetic, memory (de)allocation, `null` references, array bounds errors, etc.

In this paper, we will focus on a different aspect of code quality: Inspired by the metaphor of "*code smells*" introduced in the refactoring book [12], we review the code for problems that are generally associated with bad program

[*] Visiting research assistant from the Rigi Group, Computer Science Department, University of Victoria, Victoria, B.C., Canada.

[‡] Corresponding author.

Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)
1095-1350/02 $17.00 © 2002 **IEEE**

design and bad programming practices.[1]

Beck and Fowler introduce the metaphor of "*code smells*" to describe the patterns in code that indicate that refactoring can be applied. "*Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure*". It improves the design of a software system after it was written by tidying up code and reducing its complexity. The resulting software is easier to understand and maintain.

Code smells can be used to answer the question of *when* and *what* to refactor. The idea is not necessarily that no code smells are permitted, but rather that code smells are hints which tell us that refactoring may be beneficial. Some examples of code smells are: duplicated code, methods that are too long, classes that contain too much functionality, classes that violate data hiding or encapsulation rules or classes that delegate the majority of their functionality to other classes.

1.2. Coding Standards

Another important code quality aspect of large scale software development is conformance to coding standards. Coding standards ensure that everyone in the company can understand (and work with) each others' code. If conformance is not achieved, i.e. if the code is not written and organized according to the programming guidelines, it becomes much harder for a large team of programmers to develop, integrate, and maintain a particular piece of software. This becomes even more important in an environment where developers are geographically distributed, as is, for example, commonly found in open source development projects.

Unfortunately, conformance to coding standards is not always easy to achieve in practice. All developers involved in the project have to know and appreciate the guidelines enough to build software according to them. Experience shows that just publishing a set of programming guidelines is not enough. If developers do not really understand (the ideas behind) a particular rule, feel restricted by it, or maybe just do not believe that this rule can be useful, they are more likely to ignore that rule during development. In other cases, the set of guidelines may be so large that it is easy to overlook some of them during development. When the project comes under time constraints, these effects are often even stronger.

Consequently, overall code quality can be improved by ensuring that the code conforms to the coding standards. This process is supported by automatic conformance checking. By allowing for the definition of additional (project specific) smells, automatic smell detection turns into a conformance checking process.

In this paper, we investigate whether detection of code smells can contribute to automatic software inspection. Section 2 discusses a general approach for building a software inspection tool that is based on detection of code smells. Section 3 introduces the case study that was used to investigate the feasibility of the approach. Section 4 describes the implementation of jCOSMO, the prototype code smell browser that was developed in the case study. We conclude with an evaluation of the case study, an overview of related work, and the discussion of future work and contributions in Section 5.

[1] Please note that we do not regard this type of quality to be more important or better than the former. Both aspects should be considered when trying to improve overall software quality. We decided to focus on the second type since it is currently much less supported than the first type.

2. Approach

There are a number of important questions that need to be answered before we can automate detection of code smells in program code and use them for software inspection: What code smells are we going to detect? How are we going to detect these smells? How are we going to present the results?

The remainder of this section discusses these questions and the issues that surround them in more detail. This results in a generic approach for building software inspection tools that are based on code smell detection.

2.1. What smells are we going to detect?

In the refactoring book, Beck and Fowler present a list of code smells that they use to look for refactoring potential [12]. These smells range from simple patterns that everyone discourages, such as "code duplication" and "long methods", to more complex patterns that originate from object oriented design issues, such as "parallel inheritance hierarchies" (if you extend or change one hierarchy you will need to do the same with the other) and "message chains" (also known as the Law of Demeter: a client should not navigate through the object structure, for example, `a.getThis().getThat().foo()`).

Other examples of code smells are: "large class" especially w.r.t. classes with many fields, "feature envy" for methods that access more methods and fields of another class than of its own class, "switch statements" where inheritance should be used for specialization, "data class" for classes that do not contain functionality, only fields, "refused bequest" for classes that leave many of the fields and methods they inherit unused, and "data clumps" for clusters of data that are often seen together as class members or in method signatures but are not grouped in a class.

We can make a few observations about this list that influence our design. First, such a list of code smells can never be complete: there will always be domains and projects where a different set of code smells should be applied. For example, in an earlier paper, we have identified a number of smells that can occur in unit test code and described the corresponding refactorings to remove them [8].

Second, code smells are subjective: they are based on opinions and experiences. Creators of a list include those

patterns that they found to be useful indicators of potentially problematic aspects of the code. However, not all smells may be supported by concrete evidence and some of them may be inspired by aesthetic considerations. For example, some developers strive to minimize the use of typecasts and consider typecasts to be a smell, while others see absolutely no harm in typecasts and do not want to regard them as smells.

Finally, code smells are not precise: "*One thing we won't try to do here is give precise criteria for when a refactoring is overdue. In our experience no set of metrics rivals informed human intuition*" [12, p.75]. This is related to the subjectivity of code smells. For each project, one needs to decide what the actual parameters are: for example, which variable naming convention are used and what is the maximum size of classes and methods that is allowed, etc.

From these observations, we can only conclude that one of the main design requirements for a code smell inspection tool is that the smells should be configurable by the user. As tool builders, we can predefine a number of smells but configurability is needed to allow for: (1) definition of additional smells, (2) removal of smells that should not be considered, and (3) more precise definition of a smell so that its parameters can be tuned.

2.2. How are we going to detect these smells?

Examination of the list of code smells shows that each of them is characterized by a number of *smell aspects* that are visible in source code entities such as packages, classes, methods, etc. A given code smell is detected when all its aspects are found in the code.

We distinguish two types of smell aspects: *primitive smell aspects* that can be observed directly in the code, and *derived smell aspects* that are inferred from other aspects. An example of a primitive aspect is "method *m* contains a switch statement", an example of a derived aspect is "class *C* does not use any methods offered by its superclasses".

This distinction is used in the design of the smell detection process that is separated into the following steps:

1. Find all entities of interest in the code.
2. Inspect them for primitive smell aspects.
3. Store information about entities and primitive smell aspects in a repository.
4. Infer derived smell aspects from the repository.

This process constructs so-called *source models* from the program text. These source models are the abstraction of a system's source code that is needed for smell detection. The structure of these source models is described by a meta-model. Our meta-model was designed with analysis of JAVA programs in mind. It contains information regarding program entities such as packages, classes, interfaces, exceptions, methods, constructors, static blocks, and fields. Furthermore, it describes the relations between these entities such as composition, inheritance, interface implementation, thrown exceptions, inner classes, method calls, field accesses, and field assignments.

Our meta-model is very similar to those used by other JAVA analysis tools such as Shimba [20] and RevJava [11]. Furthermore, it has considerable overlap with the Famix meta-model that was designed with generic OO reverse engineering and refactoring in mind [22]. Although our current focus is on the JAVA programming language, we feel that this overlap indicates that it is possible to generalize our approach to other object oriented languages.

So how do we find and inspect all entities of interest in the code? Since the code smells are described in terms of program patterns and not in terms of behavior patterns, dynamic (runtime) information is not needed for smell detection. Therefore, source models can be extracted using static analysis of the program: First, a parser reads the source code and produces a parse tree containing all structural information contained in the code. Second, an analyzer reads these parse trees and traverses them according to the program structure. During this traversal, the analyser visits all program entities and stores their structure and relations in the repository. When primitive smell aspects are observed, these are also stored in the repository.

Note that is is possible to extend our approach to include code smells that need runtime information for their detection. For this, we need to add a separate extractor that is used to augment the source models with the necessary dynamic information. In general, such a "dynamic" extractor will collect this information using source code instrumentation. In the case of JAVA analysis, an attractive alternative is monitoring the runtime environment via its debugging interface.

2.3. How are we going to present the results?

The next question we have to ask ourselves is how the smells should be presented to the user after they have been detected. There are several ways in which this can be done. The intended use of the tool will have substantial influence on the the type of presentation used. We distinguish three classes of users for the detection results:

a. Programmers that use detected smells during development or maintenance of a system to improve the code.

b. Code inspectors (or reviewers) that use detected smells to assess the quality of the code.

c. Tools that use the detected smells to perform further analysis or transformations on the code, for example, software refactoring tools. Generally, these tools do not need specific presentations, they just use the repository content for further processing.

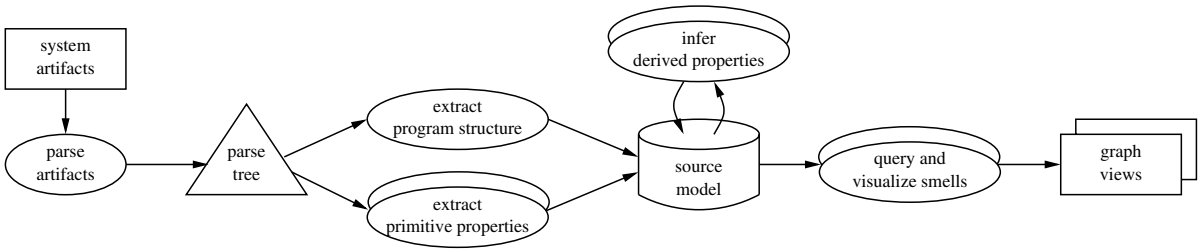

Figure 1. Architecture of code smell browser

Note that we do not consider software integrators as a separate class but assume that they switch between the inspector and developer roles.

Smell presentation for programmers should be integrated with the normal development process. This can be done without much intrusion by treating smells similarly to compilation errors and warnings. Depending on the IDE, smells will then be shown in separate message panes or integrated with the class browser (as for example is done with compiler errors in IBM's VisualAge for Java and the new Eclipse platform). Another possibility is building a dedicated smell browser that can be used to explore the repository (similar to the Windows Explorer interface). Since the main focus of this paper is software inspection, we will not investigate this approach any further.

Software inspectors have special presentation requirements for assessing the quality of (potentially large) software systems. They need to be able to get a quick overview of the complete system, showing *if* the system contains bad smells, *what* parts are affected, and *where* the concentration of smells is the highest.

We can support these requirements by generating graphical representations of the software system, in particular by visualizing the source model using structured graphs. The nodes in these graphs are the program entities of the source models (i.e., packages, classes, methods, etc.) and the edges are the relations between program entities (i.e., composition, inheritance, method calls, etc.). Since these nodes and edges can be distinguished based on their types, we can represent them using different colors and selectively hide them in various views on the graph. Graph layout algorithms can be applied in these views to improve their comprehensibility.

There are a number of options for the visualization of code smells in these graphs. For example, one can vary node attributes such as color and size for the nodes that represent code entities that possess code smells. Furthermore, code smells can be visualized using additional nodes that are connected to the entity in which they are present. It is also possible to use code smells during the computation of the graph layout, for example, by ordering them in such a way that nodes with the most smells come first (i.e., in the upper left corner of the picture).

2.4. Modular architecture

As we have seen, it is important that our code smell inspection tool is easily extendable because new smells are likely to be added during its lifetime. In particular, it should be possible for users to add their own code smells. We can support this in our design by using a modular architecture that encapsulates the detection of primitive smell aspects and inference of derived smell aspects in separate units. With such an architecture, addition of new code smells is as simple as extending the set of detectors or inference rules.

The presentation side of our tool should also be robust against addition of new smells. This has two aspects: (1) addition of new smells should not break existing visualizations, and (2) preferably, we want new smells to be included in the main views without extra work.

An overview of the architecture is shown in Figure 1. In this figure, the boxes depict inputs and outputs, the ellipses depict processing. Double lined shapes are used to indicate that this item can occur a number of times (for example, there exist several extractors for the different primitive smell aspects but only one extractor for the program structure).

3. Case Study

To investigate the feasibility of the described approach, we have performed a case study in which we developed a prototype software inspection tool that is based on code smell detection and applied it on a JAVA software system.

The system analysed, called CHARTOON, is a tool for developing 2 1/2 dimensional animations of facial expressions [19]. It was originally developed as a research prototype at the CWI under the direction of Paul ten Hagen. The research group has formed a spinoff company and is in the process of reviewing their code intensively in preparation for releasing it as a commercial product.

One of the principal software engineers involved in this task consulted with us to see whether we could provide tool support for refactoring and quality assurance. Based on our discussions and guided by the company's coding standards, we have added detection of some specific code smells to our prototype. The remainder of this section discusses these additional smells in more detail.

3.1. Instanceof as Code Smell

In JAVA, the `instanceof` operator is used to check that an object is an instance of a given class or implements a certain interface. These are considered code smell aspects because a concentration of `instanceof` operators in the same block of code may indicate a place where the introduction of an inheritance hierarchy or the use of method overloading might be a better solution.

We have found two typical patterns in which this occurs: The first is characterized by a sequence of conditional statements that test an object for its type. When the type is found, the object is cast to that type and a method is called. This can be refactored by introducing a common interface that defines the method and lets the runtime environment call the appropriate method using dynamic method dispatch (also known as late binding).

The second pattern is characterized by a method that takes a variable of the type Object (the supertype of all JAVA classes) as a parameter and has a body which contains a sequence of conditional statements that perform different actions depending on the object's type. In this case the original method can be broken up into a series of overloaded methods, each taking one of the types tested for before as a parameter. This removes the instanceof statements, making the code more modular and easier to understand.

3.2. Typecast as Code Smell

Another code smell that was added for the case study involves typecasts. Typecasts are used to explicitly convert an object from one class type into another. Many people consider typecasts to be problematic since it is possible to write illegal casting instructions in the source code which cannot be detected during compilation but result in runtime errors.

One typical pattern where typecasts create this smell can be observed when objects are stored in one of the container classes from the JAVA API. Because these classes are written as generic containers for objects of any type, items are automatically upcasted to their generic supertype Object when they are put in a container. When the programmer retrieves items from a container, they have to be explicitly downcasted to whatever type they used to be. However, since this type is not always known (and storage methods accept all types of objects), it is possible to perform illegal casting which results in a runtime error.

This pattern can be remediated by creating a wrapper around the container that is specific to the type that the container is going to hold. This way, casts are hidden in the wrapper class, and static type checking makes sure that the correct type is put into the container. In addition, the wrapper class can get a descriptive name that makes it clear what objects are contained. This is especially useful if the container is passed around in the program.

4. Prototype Implementation

To illustrate the approach described in Section 2, we have implemented jCOSMO, a prototype code smell browser. This tool was created following the architecture described in Figure 1. It consists of two main phases: the code smell extraction and the visualization. During extraction, the source code is parsed and a source model is generated that describes program structure and code smells. This source model is read during visualization to generate different views on the source code and its smells.

4.1. Extraction

The extraction of program structure and primitive smell aspects was implemented using the ASF+SDF META-ENVIRONMENT [4], an environment for developing language centered tooling that was developed at the CWI (Centre for Mathematics and Computer Science) in the Netherlands under the direction of Paul Klint. This environment supports the generation of parsers, syntax directed editors and language processing tools such as interpreters, typecheckers and source-to-source transformations. It takes two types of input: (1) *language syntax definitions* written in the formalism SDF [25] and (2) *language processing definitions* written in the term rewriting language ASF [3].

The parser generator produces *generalized LR* (GLR) parsers. Generalized parsing allows definition of the complete class of context-free grammars instead of restricting it to a non-ambiguous subclass such as LL(k), LR(k) or LALR(1), which is common to most other parser generators [24]. This allows for a more natural definition of the intended syntax because a grammar developer no longer needs to encode it in a restricted subclass. Moreover, since the full class of context-free grammars is closed under composition (unlike restricted subclasses), generalized parsing allows for better modularity and syntax reuse. For more information on SDF, we refer to [25].

Programming in ASF is done in a functional fashion by means of term rewriting: rules that describe how a given term can be translated into another term. These rewrite rules can be defined using pattern matching on concrete syntax defined in the SDF grammar. The patterns can contain variables that are bound during matching and can be reused to build the reduct. The use of concrete syntax has the advantage that the extractor writer does not have to learn a new language for processing terms.

For our prototype, we have instantiated the ASF+SDF META-ENVIRONMENT using an SDF definition of the JAVA grammar and a set of ASF modules that specify the processing that is needed for extraction of program structure and primitive smell aspects. The extraction is performed in two steps: first, the JAVA input is parsed using a parser generated from the SDF grammar. Then the parse trees are processed

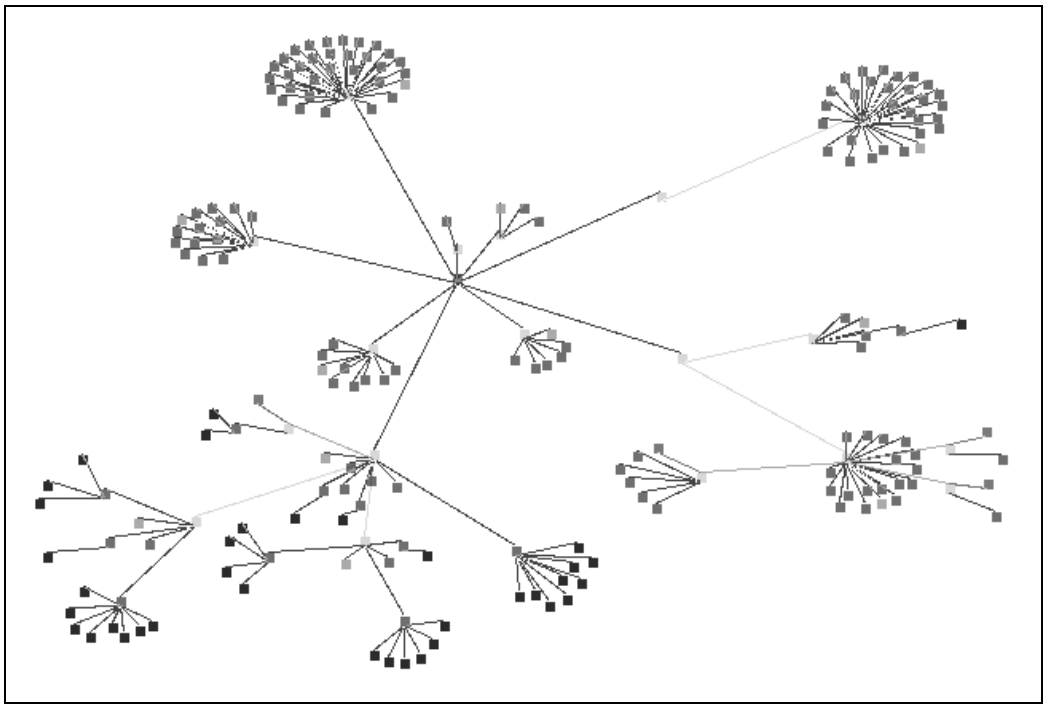

Figure 2. Complete smell detection graph for a small test system (12 classes, 1450 LOC)

using term rewriting traversals in ASF to derive the desired source model. This model describes the structure and primitive smell aspects detected in the code. The source models are represented in plain text in the three tuple notation that is known as RSF (Rigi Standard Format).

We have chosen to analyse JAVA source code instead of JAVA byte-code as is done by most other tools that operate on JAVA software. This choice has a number of advantages:

- Source code analysis allows us to treat parts of software systems that do not compile by themselves because of incompleteness. When the source code parsing is based on island grammars, it is even possible to analyse source code that contains syntax errors [16].
- Extraction of program structure and primitive smell aspects from source code can be expressed using pattern matching over the concrete syntax of the JAVA programming language. This makes it very easy for users to adapt and extend the extractor in order to detect new code smells since they do not have to learn a new language.
- Some coding standards cannot be checked on byte-code since the byte-code contains less information than the original source code (i.e., regarding layout and variable names).

In case only the byte-code for a system is available, it can still be analysed with our tool by first feeding it through a decompiler. However, one can wonder why the code quality of such a system needs to be assessed in the first place...

After extraction of program structure and primitive smell aspects, these facts are combined and abstracted to infer a number of derived smell aspects regarding code smells. These derived smell aspects are also stored in the repository. One of the tools we use for inferring these derived aspects is `grok` [14], a calculator for *relational algebra* [21]. Relational algebra provides operators for relational composition, for computing the transitive closure of a relation, for computing the difference between two relations, and so on. We use it, for example, to compute the "refused bequest" smell where child classes do not use the methods that were offered by their parents.

4.2. Visualization

The visualization was implemented using the Rigi software visualization tool that was developed at the University of Victoria, Canada, under the direction of Hausi Müller [23]. The Rigi infrastructure is based on a general graph model. This graph model is adapted to a specific domain by defining the entity types and relations of interest in a domain model. Usually, graphs are created using a parser that extracts facts from a software system and stores them in this graph model using Rigi Standard Format (RSF). The graphs are visualized using a programmable graph editor.

Generally, Rigi graphs consist of the artifacts that software engineers use to understand a software system. Ex-

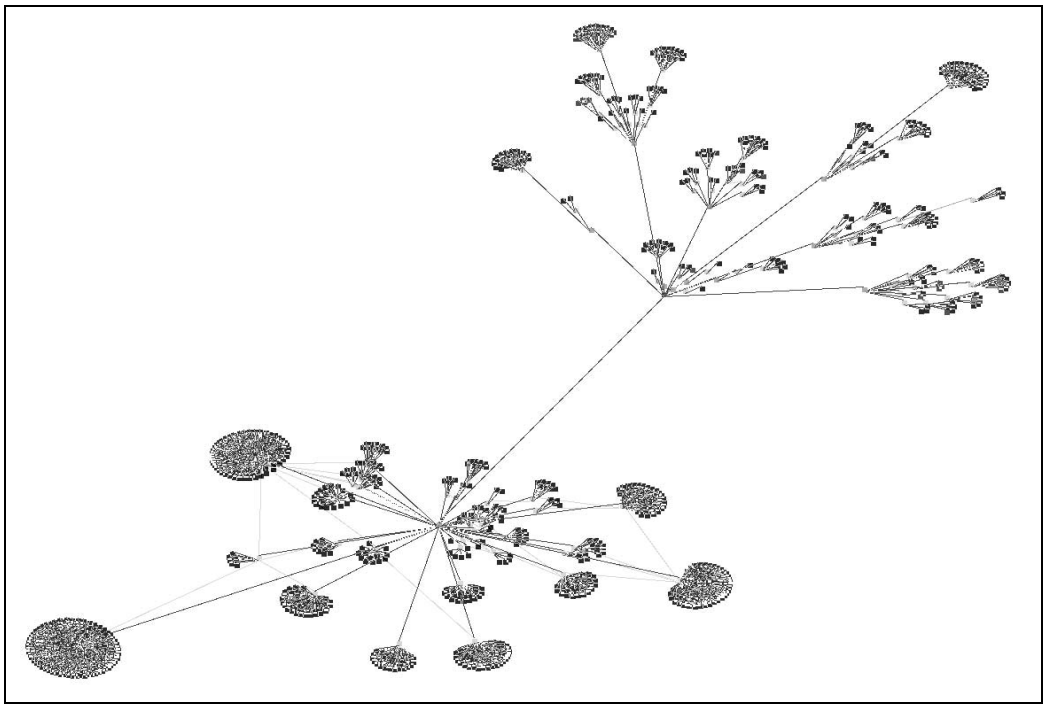

Figure 3. Collapsed smell detection graph for Chartoon (147 classes, 46 000 **LOC)**

amples are software components such as subsystems, procedures, variables, and the dependencies between them, such as composition, calls, and control- and data-flow. For our prototype, a new domain was added that describes the structure of JAVA software systems and their code smell aspects.

Unfortunately, it is not possible in Rigi to vary the color or size of a node to indicate that it possesses a smell; all nodes of a given type have the same color and all nodes have the same size. Therefore, we choose to present smells as additional nodes that are connected to the code entities that possess them. Each smell has its own node type, which has a distinct color in the graph. An advantage of this visualization is that it is easy to see which parts of the system have the most smells, and would benefit most from refactoring.

An alternative approach is to store code smell information as attributes of a node. We did not use this method since these attributes have no visual representation in the graph editor. Generally, they are used for querying the graph and their values can be inspected in a separate window.

To present the results of smell detection, we have extended the Rigi user interface with a separate jCOSMO toolbar from which the user can browse the detected smells and invoke all dedicated jCOSMO functionality. This includes various visualizations that provide customized views of extracted data and special filtering functions that can be used to show or hide certain nodes, arcs or labels. Moreover it provides a special "pruning" function that allows the user to select a particular subgraph and hide the rest.

The first view provided shows all the packages, classes, interfaces, methods and constructors, and their attached smell nodes. This gives a basic overview of the system and the distribution of the code smells. An example of this view is shown in Figure 2.[2]

A disadvantage of the previous view is that it does not scale up well for large systems. Therefore, we provide an improved view where class members such as methods and constructors are collapsed into their enclosing classes. Smell nodes are shown attached to their containing class (if a method contains a smell, it is "inherited" by the class that contains the method). By using a spring layout algorithm, it becomes clear where the code smells are clustered in the system. An example of this view is shown in Figure 3.[3] The class nodes in this view are so-called composite nodes: to view the smell distribution within a class, the user can double-click the class and a new view is opened which shows that class with its members and their smells.

For clarity, we have decided to hide node labels in these views. They can be redisplayed at any time via the jCOSMO user interface or by using a command. All typecast smell nodes are labeled with the types they are casting to and from (when known). This information is used by a number of fil-

[2] Please note that this figure shows a greyscale screendump of a view that was developed for color displays. A color screendump is available at `http://www.cwi.nl/projects/renovate/javaQA/wcre2002/curvedraw.gif`.

[3] The color version is available at `http://www.cwi.nl/projects/renovate/javaQA/wcre2002/chartoon.gif`.

tering functions that hide certain typecasts or show only casts to a certain type. Using these functions, which are also accessible via the jCOSMO user interface, a user can quickly and easily see where each kind of typecast is concentrated.

A third view orders the program entities based on code smells instead of program structure. It uses the number of smells that were detected in an entity to determine its place in the graph layout. The nodes in these graphs are ordered in a grid: all packages are arranged on the top row, distributed along an x-axis according to the average number of smells they contain. All classes are arranged on a single row below that, distributed in the same way. Finally, all methods in the system are arranged on the bottom row according to the number of smells they contain.

To prevent cluttering of the graph, this view hides the containment and inheritance arcs. Furthermore, all smells are collapsed into their containing nodes. After selecting a node in the view, the user can filter the graph using jCOSMO's prune command. Pruning leaves only the parents and children of the selected node. Thus, by pruning after selecting a method node, one sees the class and package in which it is defined, whereas pruning after selecting a class node shows all methods of that class and the package in which it is defined, and pruning after selecting a package node shows all classes and methods in that package.

5. Concluding Remarks

5.1. Evaluation

The CHARTOON system consists of 46 000 LOC (without comments or empty lines) and 147 classes. The extraction step takes about 30 sec. on a computer with an AMD Athlon processor (1.2 Ghz) and 512 Mb main memory running linux 2.4.9-12. The extracted source model contains 33 840 facts.

Because of the number of classes and methods involved, the most useful view for examining CHARTOON is the one that collapses all members into their classes while leaving the smell nodes attached to the class. This view is shown in Figure 3. Many code smells were indicated (shown as dark nodes in the figure).

Using node filtering, we have examined the distribution of each separate smell without disturbing the layout of the graph. This immediately revealed that all but two of the instanceof nodes were clustered in one class. Opening this class node revealed that within the class the instanceofs were fairly evenly distributed between the methods. This suggested that they were not linked to a switch statement, but that this might be code that could benefit from the introduction of overloaded methods. Inspection of the source code showed that such a refactoring could be performed.

It was also clear that most of the switch statements were in the same package. This might be a hint that some of these statements may be switching on the same type and could be eliminated by the introduction of an inheritance hierarchy.

Furthermore, the majority of code smells originated from the use of typecasts. Using the predefined filtering functions, new views were created that show only the typecast to one particular type at a time. These views showed that there were clusters of identical typecasts in particular areas of the system which suggests that a small amount of refactoring could remove a large number of these smells.

Feedback from the CHARTOON maintainer was generally positive. He felt that the jCOSMO views were useful for conformance checking and refactoring support. Additionally, they provided useful information for (re)documenting the system to other developers and management. He was very interested in being able to repeat detection after major revisions, so that conformance to coding standards and changes in the software quality could be monitored.

His only concerns had to do with possible difficulties when installing the tool and learning the interface. To address these issues, we added the jCOSMO toolbar as described earlier, as well as a support web page with instructions for downloading, installing and running the tool and links to available documentation. To make installation even easier, we are working on a single packaged distribution of jCOSMO that installs all the necessary components.

5.2. Related Work

Automatic Code Inspection There are a number of tools that perform some sort of automatic code inspection. The most well-known include the C analyzer LINT [15] and its JAVA variant JLINT [1] that check for type violations, portability problems and other anomalies such as flawed pointer arithmetic, memory (de)allocation, `null` references, array bounds errors, etc. IllumaSM (formerly known as InstantQA) is a defect analysis service provided by Reasoning that identifies the location of potential crash-causing and data-corrupting errors. Besides providing a detailed description of each defect found, they report on defect metrics by measuring the software's defect density (the average number of defects found per thousand lines of source code) and its relation to standard industry norms.

Generally, these tools focus on improving code quality from a technical perspective. The fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code. This differs from our approach which focuses more on code quality as seen from a program design and programming practice perspective.

More closely related to our approach is RevJava, a JAVA analysis tool developed at the Software Engineering Research Centre in the Netherlands [11]. RevJava performs design review and architectural conformance checking. It reads JAVA byte-code from which facts are derived and metrics are collected. This information is used to apply critics to

the system that check whether particular design rules were violated. A large number (70) of these critics are predefined and users can add their own. Reporting is done by means of a class browser that shows the rule violations for each class, method, etc, or using a browser that starts from the critics and shows all entities that violate that critic. There is no support for visualization of rule violations which makes RevJava less suitable for getting an overview of large software systems.

Software Metrics Another approach to assessing the quality of software systems is based on software metrics. Typically, these metrics are computed over facts that were extracted from the system's source code, which is similar to our analysis. Chidamber and Kemerer describe a suite of software metrics for object oriented systems [6]. Systä *et al.* report on using this suite for JAVA quality analysis in their tool Shimba [20]. As in our approach, Shimba represents programs as graphs where the nodes are program entities. The computed metrics are stored as attributes of a node which can be inspected in a separate window and used for querying. The metrics are not used to determine the color or layout of nodes in the graph.

CodeCrawler is a program understanding tool that combines software metrics and graphs [7]. Again, nodes represent program entities, however CodeCrawler's distinguishing feature is that it reports on the metrics of that entity by varying the size, color and position of the nodes in the graph.

There are also a number of commercial tools that use software metrics to compute the complexity and quality of software systems and present results using colored structure charts, scatterplots, metric charts and Kiviat diagrams. These tools include, amongst others, the McCabe QA and McCabe Reengineer tools by McCabe & Associates, and the Hindsight tool by IntegriSoft.

AntiPatterns Antipatterns are an extension of the design pattern idea: where design patterns describe good solutions to frequently occurring problems, antipatterns are patterns that describe frequently observed bad solutions for a given problem. Antipatterns explain why that solution looks attractive, why it turns out to be bad, and what positive patterns are applicable instead.

Brown *et al.* describe a number of antipatterns that can be found in software development, software architecture and (software) project management [5]. The software development antipatterns are very similar to code smells in that they describe commonly seen patterns in code that could benefit from refactoring. However, antipatterns are generally at a somewhat higher level, referring to source code entities at the class level or higher.

Examples of development antipatterns include "the blob" for large classes that monopolize processing, "golden hammer" for the misapplication of a familiar solution for every possible problem, "poltergeists" for classes with limited responsibility and lifetime, and "cut and paste programming" for duplicate or near-duplicate code.

Refactoring tools Finally, our approach is related to the growing body of work on tools that support the (automatic) refactoring of software systems. The original refactoring tool is the Smalltalk Refactoring Browser that was developed by John Brant and Don Roberts [17].

Recently, several other commercial and open-source tools started to offer refactoring support. These include development environments such as the Eclipse platform, Borland's JBuilder and IDEA by IntelliJ and refactoring tools such as jFactor by Instantiations, ReTool by Chive, and XRefactory that act as add-ons to popular programming environments.

All these tools have in common that they focus at the actual *code transformation* and do not analyse *when* a certain refactoring can (or should) be applied. The smell detection described in this paper could be used to add such an analysis to a programming environment allowing for an "intelligent" refactoring assistant that signals when a given refactoring can be applied.

5.3. Future Work

Beck and Fowler describe a number of smells that we can characterize as "maintenance smells". What we mean by this is that these smells are not obvious from the code itself but that they manifest themselves during maintenance of the code. These smells include:

- Divergent Change: when different parts of a class are changed in different situations.
- Shotgun Surgery: a smell that occurs when making changes requires changing many different classes.
- Parallel Inheritance Hierarchies: This is the case when making a new subclass in one place also makes it necessary to add a new subclass in another place.

Automatic detection of these smells cannot be done by analysing the program code as was described earlier; one has to *analyse the changes* that are made to the program to find out whether the program suffers from these smells. An interesting topic of future research is to investigate if the data in a configuration management system (esp. version managers such as CVS) could be used to check for these smells. Such an approach seems feasible since analysis of this type of data has already been done, for example, in the context of software evolution research [2, 9].

5.4. Contributions

We have discussed the design considerations of a software inspection tool that is based on code smell detection. We have shown how code smells can be broken up into aspects that can be automatically detected. Furthermore, we have described how the code smell concept may be expanded to include coding standard conformance. We have investigated the feasibility of the described approach using a case study

in which a prototype tool was developed and applied on a software system.

For the development of our prototype, jCOSMO, we have implemented an extendable JAVA code smell detector which can be reused in other tools. We have extended Rigi with an additional user interface that allows code smell browsing and visualization and developed several strategies for visually representing code smells within their program context.

Since the smell detection is fully automated, it can be tied into the development cycle providing continuous quality assessment and conformance checking. The graphical overviews immediately show the maintainers *if* the system contains bad smells, *what* parts are affected, and *where* the concentration of smells is the highest. Furthermore, since the analysis does not require the complete application, subsystems can be inspected before integration. This allows for incremental checking of large software systems which is especially interesting for distributed development.

Availability

The jCOSMO distribution can be downloaded from:
`http://www.cwi.nl/projects/renovate/javaQA/`

Acknowledgments

We would like to thank Serge Barthel of Epictoid for providing us with the source code and feature requests used in the CHARTOON case study.

References

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

[2] T. Ball, J. Kim, A. Porter, and H. Siy. If your version control system could talk. In *Proc. Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[3] J. A. Bergstra, J. Heering, and P. Klint. The Algebraic Specification Formalism ASF. In *Algebraic Specification*, chapter 1, pages 1–66. ACM Press & Addison-Wesley, 1989.

[4] M. G. J. van den Brand et al. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In *Proc. Compiler Construction 2001*, volume 1827 of *LNCS*, pages 265–370, 2001.

[5] W. J. Brown, R. C. Malveau, H. W. S. McCormick III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley and Sons, 1998.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

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

[8] 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 (XP2001)*, pages 92–95, May 2001.

[9] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[10] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[11] G. Florijn. RevJava – Design critiques and architectural conformance checking for Java software. White Paper. SERC, the Netherlands, 2002. See also `http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm`.

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

[13] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[14] R. 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] L. Moonen. Generating robust parsers using island grammars. In *Proc. 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22, October 2001.

[17] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[18] G. W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, 8(1):25–31, 1991.

[19] Zs. Ruttkay, P. ten Hagen, and H. Noot. Chartoon: a system to animate 2D cartoon faces. In *Short Papers and Demos Proc. of Eurographics'99*, 1999.

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

[21] A. Tarski. On the calculus of relations. *Journal of Symbolic Logic*, 6:73–89, 1941.

[22] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, December 2001.

[23] 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.

[24] M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer, 1985.

[25] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, 1997.