Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations

Master's Thesis

Ruben Wieman

Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ruben Wieman
born in Alkmaar, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Anti-Pattern Scanner: An Approach to Detect Anti-Patterns and Design Violations

Author:	Ruben Wieman
Student id:	1150812
Email:	`Ruben.Wieman@Gmail.com`

Abstract

In this Master's Thesis Project, two Code Smells, four Anti-Patterns and four types of Design Principle violations have been examined. We developed a detection program called the Anti-Pattern Scanner. This scanner has been used in an empirical evaluation where five open-source Java projects have been examined and scanned for these 'pattern' problems. The results are that the examined problems generally do occur in software systems. The problems are estimated to be inconvenient to software development, depending on the strength of their presence. Based on the Anti-Pattern Scanner and the results from this evaluation, suggestions for follow-up studies are offered.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Pinzger, Faculty EEMCS, TU Delft
Committee Member:	Dr.ir. A.J.H. Hidders, Faculty EEMCS, TU Delft

Preface

This Thesis document is the complete research report of the Master's Thesis Project. This Thesis is meant to support the candidature for an academic degree at the Delft University of Technology. This document describes the exploratory research towards detection and evaluation of software design problems, such as Anti-Patterns and sofware Design Principle violations. This research aims to answer questions regarding whether these design problems do occur in professional software projects and to provide an indication whether the problems can be harmful to such a project. This work is based on tools being developed by the Software Engineering Research Group (SERG) at the Delft University of Technology. This research also aims to complement related work and to provide a basis for further research.

I would like to thank Dr. M. Pinzger for his supervision, assistance and expertise in the field of patterns and their detection. I would also like the thank Prof. Dr. A. van Deursen for supervision and for helping me set up this project and Dr.ir. A.J.H. Hidders for his work as committee member.

Ruben Wieman
Delft, the Netherlands
April 19, 2011

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

The quality of software is affected by many aspects of the development. Design, programming and testing are some of the most important aspects of software engineering. Also the team management, product maintenance, the availability of examples such as algorithms, known working solutions and tools is important. Many helpful tools are available for software developers. Starting from pen & paper, simple text editors to large and possibly distributed development environments. Using the proper tools and technologies is as important as proper management, good requirements communication and proper user acceptance testing. However, maintenance on software products can take more than half the resources and funds of the entire product lifecycle (Vliet [44]). Making the right decisions early can greatly reduce costs later. It is in these decisions that there is a lot to gain.

There are many known good or bad practices in software engineering. In the field of software design and source code quality, these practices are called Design Patterns for ‘good’ practices and Bad Smell or Anti-Pattern for ‘bad’ practices. Many of these practices are theoretical approaches developed by software engineers. They cannot always guarantee what they promise to do. Thus, a lot of research has already been done into finding proof of the effect of these good or bad solutions. This report summarizes some of the scientific work that describes these known practices. It contains information about which practices are known, what they promise to do and summarizes some of the scientific work that shows the actual effects of these practices.

The goal of this research project is to gain further understanding of the occurrences of bad design decisions in software projects. This research builds upon related research such as [31] and [8] as it complements earlier work and adds new ideas to this field of research. It combines existing concepts and approaches to find where Anti-Patterns occur, as well as to find violations to Design Principles in general. In this research, occurrences of these design problems are discovered and explained. Their malicious nature is evaluated and possible relations between problem types are examined. The result of this research are new ideas, information about these design problems, directions for future research as well as a usable tool for detecting these design problems.

The content of this document is as follows: This chapter explains the basic concepts of good design as well as bad design. Based on these concepts, the goals of this project are formulated. Chapter 2 goes into more detail about earlier work that has examined the properties of good and bad design. It provides an explanation of techniques that can be used to find concrete examples of design problems. Descriptions of related work are given and tools for examining design problems are evaluated. Chapter 3 explains the design of the design problem detection program that is used in this research. It explains all major design decisions and presents an overview of the implementation. Chapter 4 explains the design problems in more detail and suggests heuristics that can be used for automated detection. Heuristics that have been implemented for this project are explained in detail. Chapter 5 describes the empirical evaluation that is performed using the program developed during this project. Studied cases are explained and the results are presented. Chapter 6 evaluates the results of all studied cases. It concludes with threats to validity for this study. Chapter 7 concludes with the results of this project and suggestions for future work.

1.2 Introduction to patterns, smells and principles

Design Patterns are known solutions to common design problems in software engineering. They are known good solutions for general design problems. Design patterns are usually defined as a relations between communicating objects of a software system or the way classes are structured [17]. For example, the Decorator pattern can be used to dynamically add responsibilities to an object, rather than having the responsibilities predefined through inheritance. This can add flexibility to objects at runtime which cannot be fixed at compile time.

Using Design Patterns correctly should improve the internal reuse of code (such as less duplicate code and more reusability of code) and improve maintainability. It makes extending a product easier and reduces coupling between components so they can be modified without affecting each other. At least, these are some of the claims of various books and papers about Design Patterns, such as [17]. Although Design Patterns should intuitively improve the quality of code, there are doubts about the actual usefulness of Design Patterns in many situations. Also due to these doubts, research has already been done into the actual effects of Design Patterns. Some of these researches are described in the next chapter.

Opposed to Design Patterns are Anti-Patterns. Anti-Pattern are patterns (e.g., known strategies) that are applied in an inappropriate context [5]. Based on this definition, there are two types of Anti-Patterns. The first is Design Patterns that are used in the wrong context. The other variant is known bad patterns or Anti-Pattern that are used anywhere. Patterns that are known to be wrong are introduced by [5] in a similar way as Design Patterns were introduced by [17].

Another bad coding practice is the Bad Smell [14]. Bad Smells (or Code Smells) are code taints such as long methods, code duplication and data classes. The difference between Bad Smells and Anti-Patterns is that Bad Smells tend to be local code taints within methods or classes. Anti-Patterns are usually more structural problems, such as classes using an inappropriate hierarchy. Furthermore, Code Smells are usually implementation problems where Anti-Patterns are design problems. For this reason, Anti-Patterns are sometimes called Design Smells.[1] In many cases, it is not clear whether a problem is a Bad Smell or an Anti-Pattern. Those cases are open to interpretation.

Design Principles are good practices that complement Desing Patterns. These principles do not directly present known solutions or known bad solutions, instead they provide guidelines for how software should be designed. Violations of the principles might make code difficult to understand or maintain. Design principles have been formulated by different authors and have been discussed by many software design enthusiasts. What are nowadays seen as the core design principles have been summarized and bundled in various software design books, most notably the book Agile Software Development [26]. Additional design principles can be found on discussion fora on the Internet, such as the Anti-Pattern wiki found at *http://c2.com/cgi/wiki?OoDesignPrinciples*.

1.3 Research questions

This project aims at examining Anti-Patterns and Design Principle violations. It is an exploratory study that aims to answer the following questions:

- Can heuristics be formulated that can be used to automatically detect Anti-Patterns and Design Principle violations in software systems?
- Can a program be built that can use these heuristics to find the design problems automatically and that can tell the reasons why it has found any problem?
- Do Anti-Patterns and Design Principle violations occur in software systems? Are they common or rare?
- Are they as severe and full-featured as described in literature?
- For those occurrences, can an indication be given whether they are malicious to a project?

With this exploration of design problems, our research also aims to provide a basis for possible future research.

1.4 Contributions

The contributions of this project are as follows:

- A set of heuristics for the detection of four Anti-Patterns and multiple Design Principle violations (Chapter 4).
- A tool that can automatically detect a selection of these design problems (Chapter 3).
- Evaluation of the heuristics and the occurrences of the design problems in eight software systems (Chapter 5 and 6).

[1] In this report, Bad Smell always refers to a Code Smell and the term Design Smell is never used.

Chapter 2

Related Work

This chapter summarizes earlier work on detecting Code Smells and patterns. It explains a number of techniques for detecting smells and patterns. It also describes earlier work on examining smells and patterns. Each of the works explained in this section includes a short description of their approach and the technique used in the research. Section 2.3 explains work on fault prediction or on the use of repository mining techniques. Section 2.4 presents an overview of pattern detection tools. Some of these tools have been evaluated and considered for reuse in this research.

2.1 Approaches for detecting patterns

There are many ways for systematically finding software problems. This section presents an overview of the most common methods for detecting Code Smells and patterns. The concepts of static and dynamic analysis are explained. Some possible representations of Code Smells and patterns are given for use in such analysis approaches. This section also explains the method of repository mining which is occasionally used in combination with the analysis methods. The approaches explained in this section are used by the related works that are listed in the next chapter. These approaches and the related works are used as a basis or as inspiration for this project.

2.1.1 Static and dynamic analysis

Patterns can be found using two types of program analysis: static and dynamic analysis. Static analysis is the analysis of program code or documentation. This includes examining source code or abstract representations of source code. Inspecting documentation or code can be used to find intentional and unintentional patterns. It can easily cover all available documents. The advantage of examining source code is that no information is lost by compiling. Examining abstract representations or meta-models of the source code has the advantage that the detection approach is programming language independent. The disadvantages are that information may be missed in complex situations that are affected by the different possible runtime environments or the use of reflection. Also, if every detail available in the source code is included in the search space, the detection process can be slow. Static analysis is the most common way to find patterns and smells.

Dynamic analysis is the inspection of a running program. This is usually achieved by injecting detection routines in the program binaries, thereby creating additional output when that section of the program is invoked. The advantage of this method is that it reveals the 'true' effect of the program and the way the objects interact. It may also reveal dependencies that cannot be seen in the source code, such as dynamically loaded properties or components that are configured using separate configuration files. The disadvantages of dynamic analysis are that it may be slow due to overhead of inspection, it may interfere with the program behavior and it is very difficult if not impossible to manage the program execution in such a way that it covers all code segments and patterns.

Gathering information through dynamic analysis should always be done in an automated way. Running a program and inspecting it by hand is too complex for humans to do manually. It would also be very time consuming. Gathering information through static analysis can be done in many ways. One of the most basic techniques is examining the design documents in search for potential problems. This strategy requires complete, up-to-date design information about the system. When such documents are available, reading techniques [41] can be used to inspect them. Alternatively, code metrics can be used to automatically find smells and patterns based on heuristics or by visualising properties of the code [8]. Another alternative is to transform the information into an alternative representation such as SQL [45] so the properties can be accessed through queries. Code layout can also be represented as a graph [1] to attempt 'pattern matching' of known pattern graphs.

Bad smells are usually found using static analysis because those smells are merely present in the code. They are difficult to detect at runtime because compilers usually remove information about code layout.

Methods can be combined to reduce the amount of false positives or negatives. Furthermore, there may be information available that cannot be automatically processed, such as documentation or version control histories. Such information can be used to help validate the results.

2.1.2 Definitions for patterns and smells

Code Smells and patterns have been described many times in literature. Most of these descriptions are somewhat informal, explaining the idea behind such code problems and presenting overviews. For research, it is often useful to provide a formal definition of a code problem, especially if that definition can be used by a detection program. Most of the papers and tools presented in this document have their own way of formulating patterns and smells. This section presents an overview of the most common ways to describe patterns and smells.

Creating a definition of a pattern can be done in several ways. There are various textbook sources ([17], [5], ...) that describe patterns, a lot of information is available on the Internet and intuition can help making a proper description of a pattern. Having some sample pattern instances to test any recognition program on can help greatly. Complete definitions of patterns or smells are usually presented as a descriptive text, complemented with a formal definition such as UML Class diagrams, mathematical relation operators such as propositional logic, detection rules or a combination of the basic building blocks as described below.

Code metrics are probably the most common way to describe a code problem. They do not describe relations between objects, they simply indicate quantities of specific measurements. These measurements can be taken by compilers, many IDE's and by almost any debugging tool. There are numerous code metrics and they can be combined to create new metrics. Some of the most common metrics are Number of Methods (NOM), Lines of Code (LOC) and Cyclomatic Complexity (Cyclo). Some of the features that can be measured by metrics:

Standard quantities Counts of lines of code, methods, classes, packages, ...

Complexity Calls fan in, fan out, cyclomatic complexity, amount of coupling, number of loops, ...

Communication Number of socket endpoints, dynamic classloading options, message chain length, ...

Expectations Expected bugs, estimated difficulty, volume of program, effort required, vocabulary needed, ...

Duplication Duplicate code, reused code, naming overlap, ...

Other Number of specific statements, lines of comment, ...

For most of these metrics, the min, max, averages etc are also available.

Furthermore, a number of Object-Oriented metrics exist. These metrics apply exclusively to Object-Oriented systems. These metrics include: Weighted methods per class, depth of inheritance tree, Number of children, Coupling between object classes, Response for a class and Lack of cohesion in methods [6].

One of the main problems of using metrics is that good rules and thresholds for the rules are needed to ensure that the rulebase is able to produce any valid results. The research [29] provides a way to automatically calculate usable thresholds. The problem with determining good threshold values for metric-based rules is that a lot of false positive and false negative samples are needed as data set. Collecting and examining the samples usually translates to a lot of manual labor and thus a lot if time. Yet well-tuned detection rules might pay off in the long run. The book 'Object-Oriented Metrics in Practice' [23] describes many practical approaches for using metrics and defining software properties based on metrics. This book also discusses in more detail the use of thresholds for rules.

An alternative to metrics is the Elemental Design Pattern (EDP) [38]. EDPs are solutions to the most common design problems, such as 'Class A extends class B' as a basic inheritance rule. EDPs are the building blocks of Design Patterns. They are the solution to every day source code problems. EDPs are more suitable for describing patterns than smells because they can describe where and how classes interact. Metrics are more suitable for code smells because they describe quantities and frequencies of implementation characteristics. Although metrics can also describe how often objects relate to each other (such as the Fan In metric), they do not describe specific relations between classes in a way that Design Patterns do.

When MSR is used in the research, changes across versions can be used to formulate and detect the presence of code problems. This is particularly true for Anti-Patterns because many of their characteristics are about change-proneness, causing maintenance problems or requiring specific types of maintenance patterns. If change characteristics are used to describe a pattern, the change taxonomy [13] can be used as basis. This taxonomy described source code change types as Abstract Syntax Tree (AST) editing operations and provides a formal definition for every possible change.

2.1.3 Including software history in the detection process

To gain additional information about the detected patterns or smells, the history of the software system can be included. Examining the history can yield additional information such as the evolution of the detected item, when it was introduced into the software system and whether it affected other components of the system over time. To include the history of a software system, multiple revisions of the software from the software repository are examined. Typical approaches are to examine weekly software builds or all major releases.

The paper [20] describes the classes of questions that can be answered by Mining Software Repositories (MSR). The first class of questions is the Market-Basket Question (MBQ) which can be formulates as: “When X occurs, what else occurs on a regular basis?”. The second class is the Prevalence Question (PQ) which includes boolean and metric queries. A combination of these questions may be needed to uncover the information that is sought for and to find relations.

The basic methods for answering these questions can be categorized as follows: *Changes to properties* strategy involves computation of individual properties of the sources for multiple releases, then comparing the computed properties with their changed versions. The *changes to artifacts* strategy involves examining the evolution that took the system to the next version. This includes tracking changes, bugs and additions from one version to the next.

In the past, industrial software projects were the only available sources for MSR. Nowadays, publicly available repositories such as SourceForge[1] are often used for such researches. There are a number of pitfalls and limitations to MSR [20]. The source that can be mined often contains a lot of hidden properties, such as inline communication comments (TODO, FIXME comments) that developers use as part of their task communication. Hidden information can be used in MSR research although it should be noted that it only grants a narrow view of the actual knowledge behind the system. Similarly, commit metadata (e.g., commit comments) captures only a fraction of the knowledge about the system. It is often incomplete, poorly formulated or left out entirely. Also, the data often needs normalization before it becomes usable. For example, a CVS repository registers every committed file as a separate commit, thus a sliding window normalization should be used to capture commits that belong to the same actual commit. More details about MSR and a good overview of methods and reasons for mining repositories is given in [20]. This paper explains how MSR has evolved over time.

[1] http://sourceforge.net

The MSR taxonomy [20] also provides a good summary of the basic building blocks for MSR and can be considered an important guide to MSR. The paper also provides an overview of all researches that used MSR up to 2006.

2.2 Earlier work on detecting Code Smells and patterns

Bad Smells and Design Patterns have already been examined in detail. Anti-Patterns have also been evaluated although to a lesser extend. Many of the works on these patterns are interesting for this research as they can provide insight in feasible solutions for examining the patterns and can provide directions for this research.

2.2.1 Detecting Bad Smells

Matthew James [32] built a tool that detects Bad Smells by using static analysis. This tool analyzes the Abstract Syntax Tree (AST), calculates metrics and uses a rulebase to detect Code Smells based on the metrics. The paper presents the results of the detection tool for two of the ten design problems that were detected. Examining the approach and conclusions, it becomes clear that a metric-based rulebase can be unreliable. The rules that are needed to detect the smells are open to interpretation because the design problems themselves can be described in many ways. Also the thresholds for the rules require tinkering and may not always be reliable. The problem of finding good rules and thresholds may be a threat to validity for some works.

Van Emden et al. [43] developed the jCosmo tool. This tool parses source code into an abstract model (similar to the Famix meta-model). It uses primitive and derived smell aspects (e.g., rulebase-like) to detect the presence of smells. The meta-model is used as a basis to extract primitive smell aspects from, such as “Method M contains a switch statement”. These primitive smells are then used to generate derived smell aspects such as “Class C does not use any of the methods offered by it’s superclasses”. The jCosmo tool can visualize the code layout and smell locations. The goal of this tool is to help developers assess code quality and help in refactorings. The main difference compared with other detection tools is that jCosmo tries to visualize problems by visualizing the design. As explained in [43], most other tools visualize the problems on a much more technical level, often at a programming instruction level.

Naouel et al. [31] describe a Smell detection program named DECOR. The program can detect Smells by static analysis. Smells are defined using a textual formal description and they are automatically converted to detection algorithms. The program runs these algorithms and returns the candidate classes that are found smelly. This program is used as a basis for [21] and for the DETEX tool explained in Section 2.2.2.

The Integrated Platform for Software Modeling and Analysis (iPlasma) tool described in the book ‘Object-Oriented Metrics in Practice’ [23] can be used for Code Smell analysis. This tool can calculate metrics from C++ or Java source code and apply a rulebase to detect smells. These rules combine the metrics to and are used to find code fragments that exceed the smell thresholds. These features and others are explained in detail in the book [23], as well as the rules and much practical information for using metrics.

2.2.2 Detecting patterns

Detecting Design Patterns

Bieman et al. [4] examined intentional patterns in software systems. These are the patterns that the developers intentionally chose and documented. The detection of these patterns is done by performing textual search in the documentation and examining the program code to find pattern candidates. Because the textual search yielded some false positives, a manual inspection of the model is used to verify which of the candidate classes actually participate in detected Design Patterns. Manual inspection is a commonly used method to verify the accuracy of an automated or semi-automated detection approach.

Costagliola et al. [7] use UML Class Diagrams to detect the presence of patterns. The class diagrams can be generated from source or recovered from documentation. These diagrams are stores as a Scalable Vector Graphics (SVG) format. The diagram is parsed and converted to sentences that express the relations of object, called Extended Positional Grammar (XPG). The sentences of the resulting class diagram are transformed in such a way that they can be matched against templates of Design Patterns. The templates describe small class diagrams such as an inheritance relations between multiple objects. Patterns are detected by examining every class object in the class diagram and attempting to match it's inheritance structure with a template description. If a match is found, other rules are also tested for matches because a class can participate in multiple patterns.

In [40], Stotts et al. examine small C++ programs to find Design Patterns. The authors have created a formal definition of the Design Patterns by combining Elemental Design Patterns with relation operators. This definition captures the concept of the Design Pattern and allows for identification of instances of the patterns. This enables their tool to detect patterns that have not been strictly implemented as proposed by the GoF. The data extraction process uses the gcc parse tree and converts it into a meta-model of the source code. The pattern descriptions are matched against the meta-model to find candidate patterns.

In [42], Tsantalis et al. use a graph matching method to find Design Patterns that are similar to template Design Patterns, by finding similarities in matching edges in the graph representation of the classes. The graph matching algorithm is inexact, meaning that it can also identify graphs that 'look like' the template (cliché) graphs. This allows the identification of patterns that do not perfectly match the template patterns. The graph representation is created from UML Class diagram. One of the disadvantages of this approach is that only the class relations are captured and some contextual information is not included. Some patterns cannot be differed from each other because their class relations are the same. The relation graphs map into square matrices, these matrices capture the relations between objects. Zero-entries in the matrices can be used for optimizing the matching algorithm as they can be skipped. The authors put a lot of effort into optimizing the program to be able to quickly process large systems, such as splitting the graphs into smaller hierarchies to prevent search space explosion. The program has been used to examine the jUnit, jHotdraw and jRefactory projects.

The 'Web of Patterns' project [9] describes a Java program the can be used to detect Design Patterns. The program uses a rulebase to describe the patterns. The patterns are described using Web Ontology Language. This approach also allows 'the community' to describe custom patterns and publish them on the web. Internally, the descriptions are translated to detection rules which can be processed using a Prolog-like fact resolver. The fact base is programming language dependent because not all languages support the same fact types. References to artifacts such as classes are not supported in all languages. A reference implementation for the Java language is available and is described in this paper. The scanner uses the AST representation that is built in Eclipse as a source for facts.

DeMIMA [18] is a tool that can be used to recover design motifs and trace the presence of these motifs over multiple software releases. Design Motifs are essentially the observable effects of Design Patterns in source code. DeMIMA uses multiple layers of abstraction which the user can use to describe the model of the motif that has to be detected. These layers abstract from the source code to a meta-model to an explanation based programming language that identifies the motif. The advantage of this abstraction is that the model description can be used to identify variations of the motif and not just exact matches. This property is also demonstrated in an example where DeMIMA identifies a variation to the Singleton pattern. The detected variation was an implementation of the Identity Map pattern.

The Pattern Modeling Framework (PMF) [10] is an Eclipse plugin that functions as a Design Pattern detection tool, integrated with the popular Eclipse IDE. The pattern templates are defined as UML diagrams and the tool uses static analysis to recover the Design Patterns from source code. A graph matching algorithm is used on the AST to detect the patterns.

Lee et al. [24] provide a new classification of the GoF patterns. They categorize the patterns based on how they can be detected. Some patterns can be detected by static analysis. For example, the Visitor pattern can be recovered from source code. Other patterns are best detected using dynamic analysis such as the Decorator pattern. The behavior of such patterns are best captured at runtime. Some patterns are strictly implementation specific. For example, the Iterator pattern is highly implementation-dependent and is unlikely to be implemented as described in textbook [17]. They also built a detection tool that uses this categorization to detect the patterns. This tool uses static analysis to detect the patterns categorized as detectable using static analysis. The detection rules are generated from the XML representation of the patterns. During static analysis, candidate classes for the dynamic analysis are also sought. This preparation step is implemented to reduce the search space of the dynamic analysis thereby making it faster. For dynamic analysis, the program has to be executed by the user and call traces are recorded by the tool. Call traces that satisfy the pattern constraints result in positive pattern matches. Keyword search and code-style matches are used to find implementation specific patterns.

Nija et al. [37] present PINOT, a Design Pattern detection tool that can recognize all GoF patterns that are structure or behavior driven. As explained in this paper and to some extend in other work [24], the GoF patterns can be categorized by their implementation characteristic and thus the way they can be recognized. Language-provided patterns are included in the programming language itself, such as the Iterator in Java. Structure-driven patterns are defined by inter-class structures, e.g., the overall system architecture. Behavior-driven patterns specify how objects interact, such as the Singleton pattern. Domain-specific patterns often combine other patterns into a domain-specific language and also require domain-specific knowledge to detect, making them harder to detect by generic tools. The PINOT tool can detect the patterns in the structure- and behavior-driven categories. It does so by examining the Java AST. PINOT determines for each known pattern what would be the best characteristic to identify the pattern: delegations, associations or declarations. It prunes the search space based on that criteria and attempts to detect common (known) implementation variants of the criteria. By using known variants only, it is able to detect patterns that deviate from the textbook version but it can miss some rare implementation variants.

Niere et al. [33] present the tool FAJUBA. This pattern detection tool uses the Abstract Syntax Graph (ASG) representation rules to identify patterns in the ASG. It's search algorithm uses multiple passes to identify patterns. Every pass consists of an information gathering phase and a detection phase. The tool annotates the ASG to reduce the search space and to gather information. It attempts to find matching patterns based on their ASG representations. The core concept of this tool is that it is interactive. The software engineer can interrupt it between the phases and change the ASG representations or steer the search process. This way it aims to be more interactive, perform faster and be a better help for reverse engineering.

Prolog rules can also be used to find occurrences of Design Patterns and Anti-Patterns. Stoianov et al. [39] use Prolog predicates to describe both behavioral and structural Design Patterns as well as Anti-Patterns. Such rules use a fact database to find all possible combinations of facts as long as they match the rules. Using these rules, Java components, Apache Ant and JHotDraw have been examined for occurrences of both Design Patterns and Anti-Patterns.

CrocoPat [3] is a tool that can detect patterns. The pattern templates are written in Relation Manipulation Language (RML). It can also detect code clones. A scanned program is represented as a set of relational expressions. CrocoPat tries to find tuples that match the given predicative expressions. This approach can be used to find generic patterns including some design patterns. For example, the Composite Design Pattern can be detected using this CrocoPat.

Detecting Anti-Patterns

Similar approaches can be used to detect Anti-Patterns. At the time of writing, only few researches have investigated Anti-Patterns. Dhambri et al. [8] examined the Blob and the Functional Decomposition Anti-Patterns. These patterns were identified although no further examination was done on their effects. Anti-Patterns are also mentioned to be detected by Gall et al. [15] although the Anti-Patterns were not the main focus. Van Emden et al. [43] *mention* Anti-Patterns. These bad patterns cannot be detected by jCosmo because they are usually on a higher level than metrics or Code Smells.

The DETEX tool is a detection program built on top of the DECOR [31] Smell detection tool. Using domain-specific descriptions based on Code Smells (called Design Smells in [31]), Anti-Pattern detection rules have been formulated and their detection algorithms can be generated by DECOR. This tool has been used to detect instances of Spaghetti Code, The Blob, Functional Decomposition and Swiss Army Knife Anti-Patterns.

An alternative to DECOR is an approach using B-Splines [34]. This approach uses fuzzy thresholds instead of the crisp 'Yes or No' thresholds used by many tools. In general, this approach outperforms the crisp threshold approach. However, it is sensitive to the size of the training set, meaning that a small training set can severely reduce accuracy.

The Prolog fact resolver used by [39] can detect Design Patterns as well as some Anti-Patterns. Some of the Anti-Patterns that Stoianov et al. have included are regarded as Code Smells in this report. The Anti-Patterns that have been detected are Data Class, Call Super, Constant interface, The Blob, Refused Interface, Yoyo Problem and Poltergeist. Tests on various open-source projects show that these Anti-Patterns do occur and can be detected.

Meyer [28] proposes an approach for automatic refactoring of Anti-Patterns. This theoretic approach describes how Anti-Patterns can be detected by examining the Abstract Syntax Graph (ASG). The patterns are defined as ASG rules that describe their graph layout. These patterns can be recovered by using a graph matching algorithm. Next, transformation rules described using UML can be applied to automatically transform the Anti-Patterns into normal non-Anti-Pattern classes or even Design Patterns.

Detecting Design Principle violations

Design Principles, as explained before, are clearly defined guidelines for software design. Violations to these guidelines can be detected in ways similar to Code Smells or Anti-Patterns because all of the Design Principles describe relations between program components or contents of those components. Despite the similarities and the relevance of these guidelines, there is no known research or tool that can detect such violations. Some of the tools listed in Section 2.4.2 might be able to detect some violations implicitly as those violations are actually aspects of an Anti-Pattern, but no known program is able to tell explicitly whether a principle has been violated.

2.3 Fault prediction and analysis

Fault prediction is useful in helping a software engineer to find potential problems before they actually occur. Research has been done to find how and whether fault prediction is possible. The faults themselves are also subject to research, as well as their relations with patterns. To get hard facts about whether design problems are actually malicious to a program, the data of detected problems can be mapped to software bug databases or maintenance information. Acquiring such information is a technique called Mining Software Repositories (MSR). Without such information, only estimates can be given about possible consequences of design problems. Researches regarding fault prediction and fault analysis are presented in this section.

2.3.1 Metrics as fault predictor

Metrics can be used for general purposes such as showing class sizes, giving an overview of large components in the system and to some extend, as fault predictors. Basili et al. [2] examined the usefulness of code metrics. They did not use the metrics to detect patterns or Bad Smells, instead they used them as generic fault predictors. They tested six Object-Oriented metrics for being able to predict fault-proneness during various design phases of the product lifecycle. Five of these specific metrics were found to be useful for this task. They also suggest that their models can be used to guide development in an automated way. Using their development models, the metrics can be used to reduce fault-proneness in early as well as late stages of software development.

A number of other works support the claim that metrics can be used to indicate fault-proneness. Gyimothy et al. [19] built a tool that can automatically calculate useful data (including metrics) from C++ source code. They analyzed the Mozilla project in order to investigate whether metrics can be used as fault predictors. Although the authors consider their results as inaccurate, it does indicate that metrics can be used as fault predictors in some cases. For example, they find that the rather simple Lines of Code (LOC) metric already seems to be a good fault indicator. They conclude that multivariate models based on metrics may perform better than using a single metric as estimator. The problem with such models is that they are harder to calculate. They also show that not all of the investigated metrics can be used as fault predictors, some are utterly unreliable.

Fenton et al. [12] investigated how often faults occur in a system and if this can be predicted by metrics. They found that a small amount of modules contain the majority of faults pre-release of a software product and an even smaller amount of modules contain most of the faults post-release of the product. The modules that contain the faults pre- and post-release are often not the same. They found no support for the claim that the fault density can be predicted by metrics. Complexity measures or the LOC metrics can only indicate the absolute number of faults and not the fault density.

Emam et al. [11] indicate that the size of classes can be a threat to the validity of using metrics as fault-indicators. They show that, once calculations have been normalized for class size, some metrics can no longer be used as fault predictor. A consequence is that some of the results and conclusions from earlier works might be invalid because metrics that were found to be indicators in those researches were not always normalized for class size. It also shows that special care should be taken when drawing conclusions about measured properties from software systems as it is not always certain whether these properties (such as metrics) can be used to draw conclusions about the system.

Menzies et al. [27] examine multiple methods for using metrics as fault predictor. Their best predictor is a machine learning method combined with a filter that selects a subset of the attributes that match the dataset best. They explain that traditional approaches based on attribute subsets will not predict defects properly. The attributes that are usable for fault prediction vary per dataset and the chance that any given approach will perform well is small. A lot of approaches have to be explored before a good one can be selected. In their research, only one of the six explored approaches performs satisfactory.

2.3.2 Repository mining

Repository mining is a technique that can be used to recover the history of a software system. As examining a program or it's source can reveal patterns and smells, examining the history of a software system can reveal properties such as changes, change coupling and reasons for change. This information can be used to explain why patterns are used and changed the way they are. It can give insight into aspects of patterns that cannot be seen by looking at one snapshot of a software system. Repository mining has been used in many researches. Some of the earlier works that used MSR to find information about pattern and smell behavior are listed below. The following papers are examples where versioning data or metadata is used to find additional information about patterns and smells in software.

A simple approach to identifying change ratio of classes is downloading the versioning information and then counting the number of changes to classes. This data can then be associated with other properties of these classes, such as Design Patterns. This simple approach is used by Bieman et al. [4] and Khomh et al. [22] [21] (based on DECOR [31]).

A research that uses DeMIMA is [22] (see previous section for DeMIMA [18]). This research goes into more detail on the roles that classes play in design motifs. They examine how often classes participate in motifs and how this affects the change-proneness on the classes and on related classes. They create sets of classes that participate in zero, one or two motifs according to DeMIMA. These sample sets are tested for candidate classes that participate in motifs and a voting process among developers is used to increase the accuracy of the sets. The percentages of participating classes from the sample sets are extrapolated to get the number of classes in the tested systems that participate in one or two motifs. The results show that the percentage of classes that participate in one or two of the detected motifs varies between 4% to 30%. In many cases, the percentage of classes that participate in two motifs is much higher than that of one motif. Further analysis also reveals that classes that participate in multiple motifs have a much higher chance to cause change-proneness classes the participate in one or no motifs. Those classes that do participate in multiple patterns often have greatly increased values for their metrics and are thus more complex than classes that do not. Developers should be wary of such classes as they can greatly increase the maintenance effort of the software system.

The versioning metadata can be used to complement the tools being used to find patterns or smells. This is done by Mockus et al. in [30]. Tools can examine every software version for the presence of the pattern and then the versioning metadata can be used to get additional information about the background of that version. To extract metadata from a software versioning system, the changes must have comments about what was changed for that revision. If the changes are properly commented, domain specific keyword matching tools can be used to extract the revision metadata. For example, the words “Fix”, “Fixes” and “Fixing” can be associated with corrective maintenance. This information can then be associated with the files that were changed during that revision, thus allowing automated identification of changes that are corrective. This approach also offers an interesting way to examining application evolution. It also allows identification of components that require a lot of effort to repair or maintain. Components that require a lot of fixing can be identified and refactored if necessary.

Gall et al. [15] use CVS history to observe at what rate classes change and whether other classes are influenced by the changes. The relational analysis can reveal several interesting properties of the analyzed project. In [15], each of the three following techniques complement each other. Quantitative Analysis (QA) focuses on change and growth rates. CVS history is used to analyze change frequencies on a class level. QA can indicate which classes have abnormal growth or change rates thus revealing area's of interest. Change Sequence Analysis (CSA) uses the CVS history to find classes that commonly change. Modules of the system are compared to each other based on their change patterns. Modules that frequently change together or follow the same change commit pattern are likely to be coupled together. Relation Analysis (RA) is used to find dependencies. RA goes into more detail than CSA because it uses versioning metadata to identify which modules depend on each other. This technique can also be used to find some Anti-Patterns such as bad inheritance or Spaghetti Code. Overall, this paper describes a method for identifying change frequencies and coupling. This information can be used to assess the usefulness and effectiveness of Design Patterns used in the software system.

Livshits et al. [25] explain how revision history can be mined for *usage patterns*, such as "Method call foo() always precedes bar()". These patterns are neither Design- nor Anti-Patterns. They are application specific usage patterns, such as locking and unlocking within an application. The Dynamine tool can also detect cases where the pattern is violated. The authors describe an algorithm for mining repositories, examining method transactions within the program and generating a ranking for common transactions. The authors examine Eclipse and jEdit, both of which are large systems. Their algorithm does a lot of filtering, such as leaving out common method calls such as logging methods. Without filtering, the algorithm would produce too many candidate patterns and too much noise. Dynamic analysis is used to verify that the pattern candidates are really usage patterns. By injecting additional inspection code at runtime, traces are generated to count the actual usage of the patterns. This runtime validation ensures that the detected patterns are 'real' patterns thus greatly reducing the need to deal with false positive patterns. Because the usage count of a pattern depends on how the program is used at runtime, only the number of unique ways of invoking the pattern is counted. This also results in a more effective count of pattern violations.

Vokac [45] detects Design Patterns and associates them with defects. An automatic code parser is used to parse the code into an abstract model which is kept in a database. Database query representations are provided for the Factory, Singleton, Observer, Template Method and Decorator patterns. Patterns are extracted from the meta-model by querying the database. False positive pattern candidates are detected by manual source code inspection. Because checking an entire software project for false negatives is not possible for large software systems, a sample set of the source is used to estimate the false negative rate. The defect frequency is determined by examining the source code over multiple CVS revisions. A Defect Tracking System is used to track the defects across versions. For components that did not have direct links from this tracking system, the defect rates were extrapolated. Also, because a class can participate in multiple Design Patterns or it can be a generally big or complex class, it is not always certain whether the defect rate of that class can be associated with a design pattern.

Di Penta et al. [35] based their research on DeMIMA. In addition to detecting the Design Patterns, change sets are identified and assembled as sets of multiple *delta's*. A *delta* is a set of changes that have been committed by the same user, within a few seconds and that all share the same commit comment. The changes are defined in terms of source code lines added, changed or removed. The authors compare the results of the frequencies of specific changes with what they expected from these patterns.

A similar approach is used in [1]. Aversano et al. use the tool presented in [42] and use a similar change-detection approach as [35]. Their approach also identifies co-change. Some classes that participate in Design Patterns are shown to have a higher co-change than others. By examining the changes and co-changes, it is shown that some patterns, expecially the ones with crucial roles, often have a higher co-change than those which are less crucial.

2.4 Evaluating existing tools for reuse

2.4.1 Choosing an existing tool for future research

This section explains some of the selection criteria for tools that can be used for Anti-Pattern detection. The task of finding Anti-Patterns should be automated where possible, thus allowing a quicker and broader search for Anti-Patterns. Software should be reused where possible, thus reducing the need to design and build programs that already exist. Many general problem detection tools already exist, such as PMD[2] or FindBugs[3]. However, only few of these tools can detect Anti-Patterns. Tools that can be used for future research should satisfy the following conditions:

- The tool should be well-documented both in the detection strategy it employs and how to modify it. Preferable the source code should be available or alternatively it should have a good way of modifying the pattern detection logic.
- Preferably, the tool should show some signs of life. Recent developments or community support would be best. A tool that has not been maintained for years may not be the best option.
- If additional tools will be used for research methods such as MSR, statistical analysis or other input / output steps, then it would be helpful if the detection tool can be integrated with programs for those research methods. If that is not possible then the programs should at least provide input and output in a format that can be used in the next program in the chain.

Most programs produced for earlier researches are typical prototypes that were designed to do exactly what they were needed for and nothing more. Those programs often exactly fit the scope of the research and are hard to adapt for further use. For this reason, most of the programs described earlier are not suitable for use in this project. Other works produced tools just for the purpose of producing a generic tool that is usable in future research or for (non-)commercial purposes. These tools may be more suitable than prototype tools because they are more generic. Some of these tools are listed here:

- The program described as 'Design Pattern Detection using Similarity Scoring' [42] can be found at the author's homepage.[4] The program can be tested on examples[5] or on custom source code.

[2] http://pmd.sourceforge.net/

[3] http://findbugs.sourceforge.net/

[4] http://java.uom.gr/ nikos/pattern-detection.html

[5] http://www.vincehuston.org/dp/

- Another Design Pattern detection tool.[6]
- The CrocoPat project[7] offers a tool [3] that can detect patterns.
- The Web of Patterns project [9] can be found on their homepage.[8]

There is no off-the-shelf program that can be used to detect Anti-Patterns. There are various candidates that might be able to do the job if some changes can be made to the program. Those tools are listed in the next section.

2.4.2 Summary of pattern detection tools

Below is a table that summarizes the detectable patterns of the tools discussed earlier. The information in this table is based on what is described in the papers or official documents provided with the tool. Some papers explicitly state that the tool can be adapted and in those cases, a 'Y' is given for *custom patterns*. Using custom patterns should also give some room for detecting Anti-Patterns although that can only be said with certainty after the Anti-Patterns have been defined using a formal definition that can be implemented in such a program.

From Table 2.1 it becomes clear that there is no single tool that can detect all GOF Design Patterns. Combining strategies is promising as is done by [24] where static and dynamic analysis have been combined. For complete coverage of all Design Patterns, a new strategy might need to be designed or existing tools have to be combined. Anti-Patterns have the same problem as Design Patterns. To cover all possible Anti-Patterns that can be found in a program, combined detection approaches are probably necessary. Existing tools that are adaptable can probably detect a subset of the Anti-Patterns, depending on their chosen pattern detection strategy.

Where possible, these tools have been examined and tested for the criteria listed in Section 2.4.1. The conclusion of this test is that there is no tool that satisfies all criteria (e.g., good documentation, recent development, adaptability and possibly integration with other tools). The lack of any directly usable tool for detection of custom Anti-Patterns and Design Principle violations is the main motivation for building a detection program from scratch. The program that has been developed for this purpose is described in Chapter 3.

[6] http://www.sesa.dmi.unisa.it/dpr/

[7] http://www.sosy-lab.org/ dbeyer/CrocoPat/

[8] http://www-ist.massey.ac.nz/wop/

Source	Abstract Factory	Active Object	Adapter	Bridge	Builder	Chain of Responsibility	Command	Composite	Decorator	Factory Method	Façade	Flyweight	Interpreter	Iterator	Mediator	Memento	Observer	Prototype	Proxy	Singleton	State	Strategy	Template Method	Visitor	Custom or Anti-Pattern
Graph matching [1]	N	**Y**	N	N	N	N	**Y**	**Y**	**Y**	**Y**	N	N	N	N	N	N	**Y**	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	?
Manually [4]	N	N	**Y**	N	**Y**	N	**Y**	N	N	**Y**	N	N	N	**Y**	N	N	N	N	**Y**	**Y**	**Y**	**Y**	N	**Y**	**Y**
Visual language parsing [7]	N	N	**Y**	**Y**	N	N	N	**Y**	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	**Y**
CrocoPat	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	**Y**
(Unknown)[a]	N	N	**Y**	**Y**	N	N	N	**Y**	**Y**	N	**Y**	N	N	N	N	N	N	N	**Y**	N	N	N	N	N	N
Similarity scoring [42]	**Y**	N	**Y**	N	N	N	**Y**	**Y**	**Y**	N	N	N	N	N	N	N	**Y**	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	?
Web of Patterns [9]	**Y**	N	**Y**	**Y**	N	N	N	**Y**	N	N	N	N	N	N	N	N	N	N	**Y**	**Y**	N	N	**Y**	**Y**	**Y**
DeMIMA [18], [22], [35]	**Y**	N	**Y**	N	N	N	**Y**	**Y**	**Y**	**Y**	N	N	N	N	N	N	**Y**	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	?
PMD	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	?
DECOR [31]	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	**Y**
PMF [10]	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	**Y**
SPQR [40]	N	N	N	N	N	N	N	N	**Y**	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	**Y**
Static + Dynamic [24]	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	N	**Y**	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	N	**Y**	**Y**
PINOT [37]	**Y**	N	**Y**	**Y**	N	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	N	N	**Y**	N	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**
FAJUBA [33][b]	**Y**	N	**Y**	**Y**	N	**Y**	N	**Y**	**Y**	**Y**	**Y**	**Y**	N	**Y**	**Y**	**Y**	**Y**	N	N	**Y**	N	**Y**	**Y**	N	**Y**
Prolog [39]	N	N	**Y**	N	N	N	N	N	**Y**	N	N	N	N	N	N	N	**Y**	N	N	**Y**	N	**Y**	N	N	**Y**
Queries [45]	**Y**	N	N	N	N	N	N	N	**Y**	N	N	N	N	N	N	N	**Y**	N	N	**Y**	N	N	**Y**	N	?

Table 2.1: Detection tools and their out-of-the-box detectable patterns.

[a] http://www.sesa.dmi.unisa.it/dpr/

[b] Neither the paper [33] nor the FAJUBA web site specify any of the patterns that it can detect although [37] compares against it and it does list many patterns

Chapter 3

The Anti-Pattern Scanner

Because this thesis aims to explore Anti-Patterns, Design Principles and similar properties that manifest in a program's source, it is necessary to have a proper tool that can detect these properties. Such a tool has been developed in this Thesis project.

The first 'proof of concept' goal of this tool was to demonstrate that a simple heuristic can be used to detect Anti-Patterns. Additional patterns were implemented over time and discussion about the results suggested that it can do more, better and with proper result reporting. For this reason, the original prototype 'Anti-Pattern Scanner' has been designed incrementally to also include Code Smells and Design Principles. This chapter contains the latest design information for this Anti-Pattern Scanner.

3.1 Global design

3.1.1 Requirements

As explained in Section 2.4, the three requirements for reusing a pattern detection tool are:

- Good documentation about how it works and about the detection approach. In addition to that, some way to edit the tool or to add custom patterns. The tool must be capable of detecting the patterns because not all detection approaches work well for patterns.

- Signs of life. Compatible with recent versions of the Eclipse IDE, active development or active support.

- A proper interface to process input and output, also with respect to compatibility with an approach for Mining Software Repositories (MSR).

Evaluation of existing tools has shown that there are no available tools that support all requirements. There are available tools that can be downloaded from the Internet. Some of these tools can detect patterns when used on a Java project. Usually these tools also have some documentation such as a research paper describing their detection approach. However, there are no tools that satisfy all of the requirements for an Anti-Pattern detection tool.

In addition to the requirements stated earlier, we added requirements that should make it convenient to use and to open up possibilities for future use. The functional requirements are:

- Be able to detect as accurate as possible instances of chosen Anti-Patterns. It's decisions about what is and what is not an Anti-Pattern should be near-equivalent of what a programmer might decide after examining a program.
- The program must be able to tell how and why it found a problem and why it did not. Reporting should be clear and precise. The user should not need to delve through documents, stacktraces or chunks of code to review the scan results.

In addition to the functional requirements, there are a number of non-functional requirements:

- The program is allowed to be a prototype program for research. It is not meant to be developed in a large team or to see everyday use by non-developers. To increase development speed, it does not need to be fully documented in every detail or to have professional versioning and release management. However, it should have an overall clear structure and proper use of Object-Oriented programming principles. It should not be necessary to completely redesign and rebuild it before it can be used outside of the current scope.
- Runtime should be fast. A scan should be able to complete in the order of seconds to minutes thus allowing quick adjustments and re-scanning of sources. This requirement is for convenience. It is not a problem to leave a computer running for an overnight scan however fast scans greatly increase usability.
- A scan must be able to run in the background. Most users have other things to do than staring at a 'scan in progress' dialog.
- Because the current version is still a prototype and is meant to be used in research, it is acceptable that it will not be suitable for a wide audience. It will require some domain knowledge from the user, such as a basic understanding of Design Patterns, Anti-Patterns and Eclipse IDE usage. It is not meant to be a programming assistant for everyday use. Assumption is that the user does know the limitations of his or her hardware.
- A normal desktop computer or workstation must be able to run the program properly. Assumptions are that a reasonable amount of memory is available to the Java VM, such as more than one gigabyte. This is expected to be reasonable for development machines.
- Because modern computers often have more than one processing core, multicore processing should be available unless there is proof that the chosen detection approach is not suitable for parallel processing. It will not be optimized for high-performance computing.

Future versions of the program might also be used for an additional purpose: to detect Anti-Patterns while the system engineer builds or refactors a software system. It might be a helpful tool to detect some design problems and it can be used to complement existing Code Smell and problem detection tools.

3.1.2 Basis and core components

The input for the program is a Famix[1] Eclipse Modeling Framework (EMF[2]) meta-model as included in Evolizer[3] [16]. There are three reasons for this decision. First, this model is an existing, working and proven abstract model that contains virtually all of the information necessary to perform static analysis. Because it offers a convenient file-based database of source-code entities (e.g., an .emf file), it also opens up possibilities for storing input data for repeated analysis. Second, it has an existing approach for MSR. A repository can be analyzed and releases can be stored as EMF files. This feature will not be used in this project although it opens up possibilities for further research. Last but not least, active support is available for this product. Figure 3.1 shows where the Famix parser fits in the flow of information. A program's source code is parsed and converted to the abstract model. This model containing all the Famix EMF entities can either be stored as a database file or fed directly to the Anti-Pattern Scanner as shown in Figure 3.2.

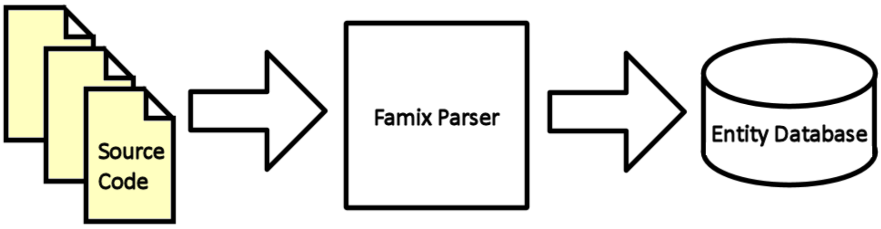


Figure 3.1: Source code to model conversion.

Given a Famix EMF model, the program will scan the entities of the model using all of the pattern scanners. The intermediate and final results are cached to allow quick access to them. The entire cache can be passed to an output generator, such as an Eclipse View component. For an example of what this process looks like, see Section 3.2.1.

The current version of the Anti-Pattern Scanner does not use any frameworks except for the provided Famix EMF and the Eclipse IDE. Smell detection, metric generation etcetera is done manually by any component that needs it. The reason for this is that the only source of information is the provided EMF model, meaning that any library should be able to use this. Furthermore, each of the heuristics developed so far are rather simple and do not need third-party programs to run. A manual implementation using the available system might also be faster in many cases.

[1] http://www.moosetechnology.org/docs/famix

[2] http://eclipse.org/modeling/emf/

[3] https://www.evolizer.org/

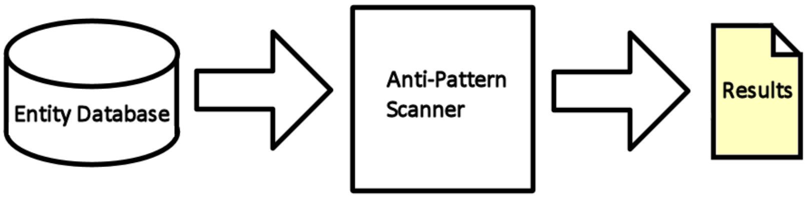


Figure 3.2: The model is the input of the Anti-Pattern Scanner.

There are two important aspects of this program. The first is the rating system, the second are the scanners. The program works by using a tree structure of scanners to examine all the aspects of an Anti-Pattern. The root scanner of the Anti-Pattern accepts a Famix EMF model as input and provides a rating as output. The rating gives an indication of how much and how certain the model contains instances of the Anti-Pattern. The scanner can construct this rating by delegating subtasks to other scanners and combining the results.

3.1.3 Rating

A Rating is a container for information that describes how strong a Famix EMF entity resembles an aspect of an Anti-Pattern. It contains a rating value that identifies the strength of the indication and it can contain a reason for that value. For example, a scanner might identify how many system calls a class has. Assuming that many calls would be bad, the scanner can give a 'strong' rating when it identifies a class that performs many calls or a 'negative' rating if the class does not.

The ratings have some advantages over other approaches presented in related works. The values are fuzzy, meaning that the program is not limited to thresholds that result in a crisp Yes or No. This should provide more freedom in the implementation of heuristics. The fuzzy values are not completely free either. The choice of results is limited to a small number of ratings, each having a meaning instead of being just a variable for a formula. The available values should make it clear whether a design problem aspect evaluates to true or false while maintaining some of the benefits of fractional values. It is meant to simplify the heuristics where possible while keeping the freedom and imagination of the user.

The main purpose of the rating is that it allows passing of judgements between scanners. The second purpose of the rating is that the Rating container can help to combine ratings. Every rating contains a value that follows certain guidelines. The container offers functions for combining ratings on various scales. The default scale uses integers to set an indication level:

Strong Negative A value of -10 indicates that the given entity is unable to possess the scanned aspect of the Anti-Pattern. It indicates a high certainty that the pattern will not be found in the given structure. This rating should be used with care because it will severely reduce the likelihood of a positive indication when combined with other ratings.

Negative A value of -2 indicates that the given entity does not indicate the presence of the aspect at all. In fact, it might be a textbook example proper design.

Neutral A value of 0 means that the scanner cannot provide a judgement or that it's value is neither positive nor negative. It is often better to provide a positive or negative judgement just to ensure that the final judgement goes the direction determined by the heuristic. Returning a neutral value makes the scanner look indecisive and could make the final calculation less confident.

Weak A value of +1 indicates that the entity shows some hints of the scanned aspect. Only weak indications should not be enough to accuse the entity of participating in an Anti-Pattern.

Strong A value of +3 indicates that the entity has exceeded thresholds for the heuristics and that the scanned aspect is most likely violated.

Very Strong The entity exceeds threshold values by a high amount and certainly possesses the scanned property. This indication has a value of +6.

The reason for not choosing a linear scale is that this scale makes the Strong Negative and Very Strong outliers have a heavier impact on the final calculation.

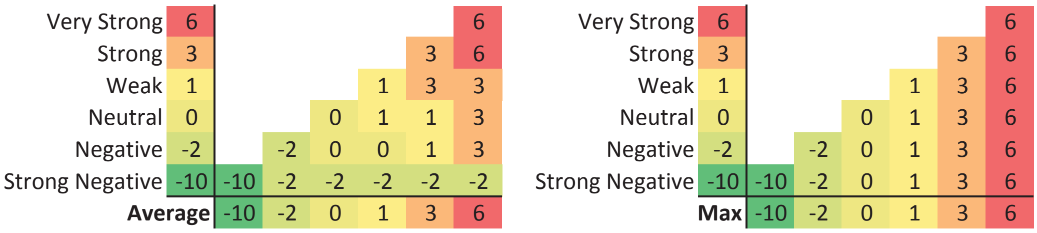
The Rating container also provides functions for using the given indication levels on a linear scale from Strong Negative (-2) to Very Strong (+3). It also provides utility functions for fractional calculations, such as providing a weighted average for multiple ratings and float rounding to one of the indication levels.

The NestedRating is a container for bundling multiple ratings and other nested ratings into a final judgement rating. It is used by complex scanners that generate their judgement based on the results of other scanners. These nested ratings are meant to create a one-to-one mapping of ratings to their scanners. A tree structure of ratings uses nested ratings as intermediate nodes and single ratings as leaf nodes.

Many of the scanner implementations combine rating values to create new values. Combining the values is usually implemented as either a fuzzy 'and' condition or a fuzzy 'or' condition. The 'and' condition consists of averaging the input values, the 'or' condition takes the strongest of the values. The resulting value is usually rounded to one of the crisp values listed above although scanners do not strictly need to round. Rounding can be skipped during intermediate steps where scanners need to continue working with fuzzy values. Figure 3.3 shows the resulting values for the combination of two input values (the X and Y axis). The table on the left shows the combination values for an *average* ('and') condition, the table on the right shows the combination values for the *maximal* ('or') condition. Scanners are not restricted to using only two input values and they can add weights to input values if needed.

3.1.4 Scanners

Every pattern consists of multiple aspects. A combination of problems of design decisions together makes a pattern, with each individual problem or decision an aspect. The approach of the Anti-Pattern Scanner is to examine a class and to answer the question whether that class participates in an Anti-Pattern or not. This question can be answered by splitting the problem into the different aspects of the Anti-Pattern. Each aspect can be examined



Very Strong	6						6
Strong	3					3	6
Weak	1				1	3	3
Neutral	0			0	1	1	3
Negative	-2		-2	0	0	1	3
Strong Negative	-10	-10	-2	-2	-2	-2	-2
Average		-10	-2	0	1	3	6

Very Strong	6						6
Strong	3					3	6
Weak	1				1	3	6
Neutral	0			0	1	3	6
Negative	-2		-2	0	1	3	6
Strong Negative	-10	-10	-2	0	1	3	6
Max		-10	-2	0	1	3	6

Figure 3.3: Rating mappings for combining two scanner results.

and if the class possesses all (or any) aspects of the Anti-Pattern then a heuristic can be used to determine how strongly the class participates in the pattern. Each of the aspects can be treated as a new question. Many of the aspects can be split into smaller questions again. This can be repeated until only very basic questions remain. These are usually metric questions that can be retrieved from the model without complicated if-then-else routines. In general, an Anti-Pattern can be represented as a tree structure of properties, with each of those properties being an intermediate or leaf node in the tree structure. The root node of the tree returns a single rating which tells the user how strongly the Anti-Pattern is present in the model.

The program layout is similar to the tree structure of questions. This enables the question implementation to be simple, either using a heuristic to answer a simple question or to delegate and combine questions. This mapping of a pattern to questions also enables some re-use of questions between different Anti-Patterns, it allows for parallel processing and generally makes the program easy to design and understand. The execution of the heuristics are the tasks for ‘scanner’ components. There are three types of scanners: Pattern scanners, simple scanners and complex scanners.

A pattern scanner is tasked with performing all scan operations for a single Anti-Pattern. The input is the Famix EMF model and the output is a rating that describes the pattern presence indication for the model. Most pattern scanners delegate the task to a complex scanner that determines the pattern strength for a single class. The pattern scanner can return the average rating for all classes or use a heuristic if the average would not be relevant. Parallelism is usually implemented by the pattern scanner. Running a scan per class can be done in parallel. Although every step in a decision tree can be done asynchronously, it is easier to manage the scheduling of parallel tasks at a single and high level. Merging parallel threads is necessary before any result can be returned to the parent of a scanner.

Simple scanners are used to answer a single query, or multiple closely related queries. A simple scanner usually inspects a part of a single aspect of an Anti-Pattern. For example, in The Blob Anti-Pattern, the *ManyMethodsScanner* is part of the implementation of the class size aspect. Within that aspect, it rates the number of methods.

Complex scanners also answer queries, but do so by splitting the query into subqueries. The tasks are delegated to other complex scanners and eventually to simple scanners that provide the answer for the most basic queries. The complex scanners combine the ratings using their own heuristics.

There are some guidelines for scanners. The pattern scanner should be named after the pattern that it detects. All complex and simple scanners should be named after the aspect that they process. Also, a pattern scanner should always be the root scanner of the decision tree. Except for the task of performing parallel scheduling, it is similar to a complex scanner. ‘Complex scanner’ should not mean ‘difficult scanner’. A complex scanner that uses if-statements in a loop or vice versa can probably be split into two complex scanners: one for the loop and the other for the decisions. Tasks can be delegated that way. Most complex scanners either delegate a task to multiple different scanners and compute a weighted average of the results, or run one other scanner in a loop and return an averaged rating. Other heuristics are allowed although often not necessary. Simple scanners are not allowed to delegate tasks. They answer a single question or multiple closely related questions, which must be as small and as simple as convenient.

The tree structure of scanners has a number of advantages over a ‘flat’ rulebase. It provides derived answers by combining properties. Because the derived answers are constructed using a tree structure of queries, the rules are easy to understand. The tree structure allows for easy identification of which scanner provided what answer. The resulting tree structure of ratings is useful for feedback and for tuning the heuristics. The concept is that scanners are small, modular and that the technique can be mapped to any step in an Anti-Pattern detection approach. Anti-Patterns that share aspects might even be able to reuse some of the scanners. Another theoretical advantage of the scanner implementation is that almost every step can be done in parallel.

The scanner architecture also offers caching opportunities. When a scanner is asked for it’s judgement twice, the second time it can return a cached result. A simple caching layer can optimize runtime speed for scanner-to-scanner communication. This ensures that complex scanners can be simplified by not having to worry about invoking other scanners too often. Furthermore, when a model scan is complete, a complete memory with all results including intermediate results is available.

In some cases, the search tree can be pruned. For example, when a scanner high in the search tree decides that an entity cannot participate in the pattern (e.g., a ‘Strong Negative’ rating), it could skip scanning all other aspects of the pattern for that entity. Search tree pruning is optional. In a few cases it will grant a performance boost. The disadvantage is that not all aspects are examined, possibly leading to a biased result. Furthermore, one of the features of the search tree approach is that even when the final result is that there is no pattern, intermediate ratings might indicate that there is *some* problem if an entity does possess some pattern aspects. This might be missed if the search tree is pruned.

3.1.5 Testing and tuning

There are two types of tests. The first is to actually test the scanners and ensure that they work as designed, the second is to tune the heuristics and thresholds.

Testing the scanners is done by feeding them specially prepared Anti-Pattern samples as well as non-Anti-Pattern samples. The Anti-Pattern samples should contain at least one implementation of every major case of the pattern. Manually selected entities can be fed to separate scanners and their output can be verified manually.

Tuning the scanners should be done in steps. The initial thresholds and heuristics are designed by examining the pattern, listing target values[4] for the metrics and using intuition. Beginning with small models (such as the samples), some thresholds can be verified. When the rate of true and false positives and negatives is satisfactory, larger models can be examined. This process can be repeated multiple times, each time stepping up a model size.

3.2 Implementation

This section contains a demonstration of the Anti-Pattern Scanner. The design described in the previous section has been implemented and the heuristics described in the next chapter have been implemented in this program. The example in this section shows how the program is used and what the output looks like.

3.2.1 Anti-Pattern Scanner user experience

The Anti-Pattern Scanner is an Eclipse IDE plugin that can work with any Java project. Starting the program is simple but analyzing the results is far from trivial. The figures 3.4(a) to 3.4(c) show the process of scanning a project. An example result overview is shown in Figure 3.5.

Analyzing a project requires some specific knowledge about the reasoning behind the patterns. The detected occurrences can give a basic explanation about why they are detected but this information is rather limited. This document or even a simple web search should yield much more information than what the plugin can tell.

Analyzing the reasoning tree (see next section) will also require some conceptual knowledge about the layout of the scanners. All information can be found in this document. Due to the limited information and short explanations, the program is good enough for research but it is not user-friendly enough to publicate it as is.

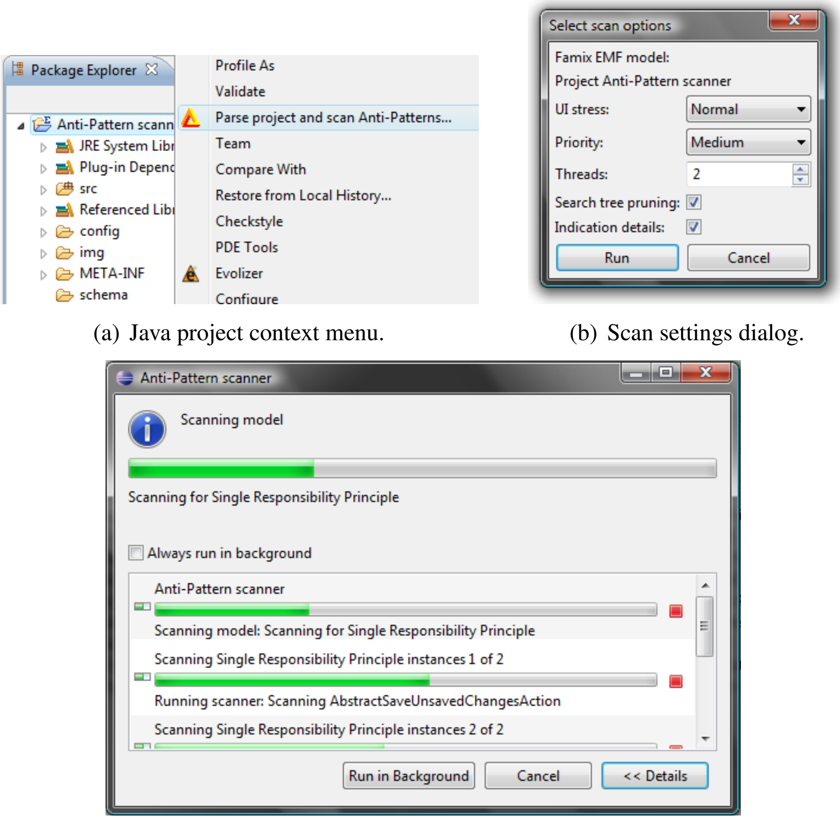
3.2.2 Evaluation of Anti-Pattern Scanner design

Some of the heuristics described in Chapter 4 have been implemented and evaluated. They have been tested and used in the empirical evaluation (Chapter 5) and this section describes some of the impressions of their performance.

The Anti-Pattern Scanner was originally designed for scanning Anti-Patterns, hence the name. The patterns can be split into multiple, smaller fragment scanners that each determine values for aspects of a pattern. An example result of combined scanner effort is shown in Figure 3.6. The output contains values ranging from very strong positive to negative indications. Leaf nodes are more towards the bottom and the combined result value is displayed on top. A mix of many positive and negative indications is typical for patterns where many aspect scanners are used.

Combining scanner heuristics works very well. It does require careful evaluation of output values for the lower-level scanners and combination formula for higher level scanners

[4] http://www.aivosto.com/project/help/pm-list.html



Famix EMF model:
Project Anti-Pattern scanner

UI stress: Normal

Priority: Medium

Threads: 2

Search tree pruning: ☑

Indication details: ☑

Scanning model

Scanning for Single Responsibility Principle

☐ Always run in background

Anti-Pattern scanner

Scanning model: Scanning for Single Responsibility Principle

Scanning Single Responsibility Principle instances 1 of 2

Running scanner: Scanning AbstractSaveUnsavedChangesAction

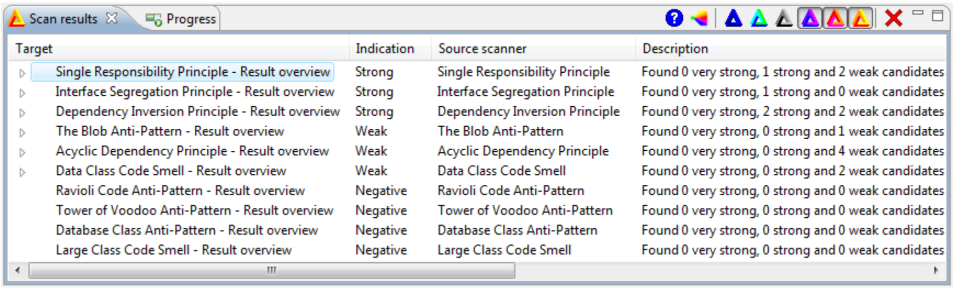
Scanning Single Responsibility Principle instances 2 of 2

(a) Java project context menu.

(b) Scan settings dialog.

(c) Scan in progress. . .

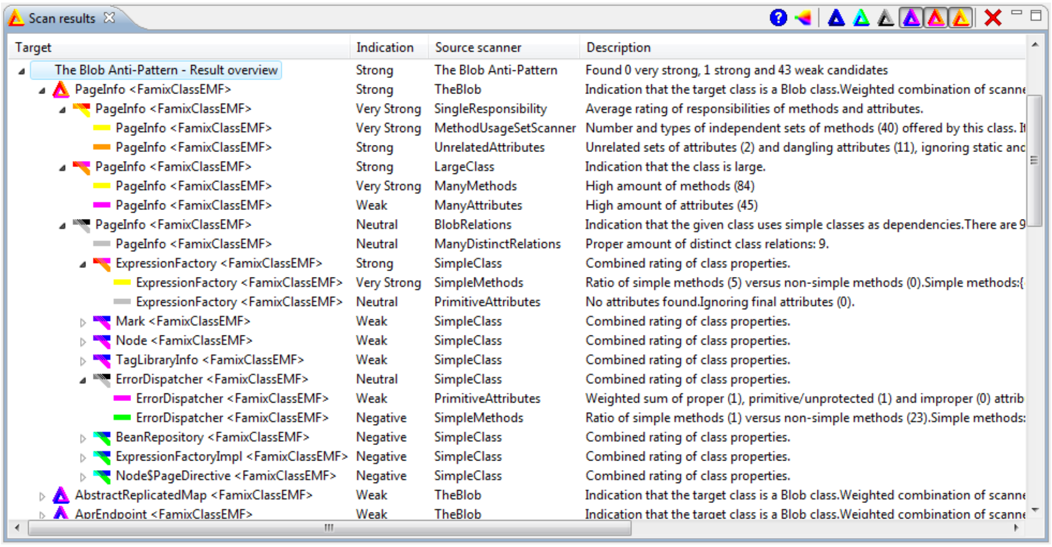
Figure 3.4: Anti-Pattern Scanner run procedure.



Target	Indication	Source scanner	Description
Single Responsibility Principle - Result overview	Strong	Single Responsibility Principle	Found 0 very strong, 1 strong and 2 weak candidates
Interface Segregation Principle - Result overview	Strong	Interface Segregation Principle	Found 0 very strong, 1 strong and 0 weak candidates
Dependency Inversion Principle - Result overview	Strong	Dependency Inversion Principle	Found 0 very strong, 2 strong and 2 weak candidates
The Blob Anti-Pattern - Result overview	Weak	The Blob Anti-Pattern	Found 0 very strong, 0 strong and 1 weak candidates
Acyclic Dependency Principle - Result overview	Weak	Acyclic Dependency Principle	Found 0 very strong, 0 strong and 4 weak candidates
Data Class Code Smell - Result overview	Weak	Data Class Code Smell	Found 0 very strong, 0 strong and 2 weak candidates
Ravioli Code Anti-Pattern - Result overview	Negative	Ravioli Code Anti-Pattern	Found 0 very strong, 0 strong and 0 weak candidates
Tower of Voodoo Anti-Pattern - Result overview	Negative	Tower of Voodoo Anti-Pattern	Found 0 very strong, 0 strong and 0 weak candidates
Database Class Anti-Pattern - Result overview	Negative	Database Class Anti-Pattern	Found 0 very strong, 0 strong and 0 weak candidates
Large Class Code Smell - Result overview	Negative	Large Class Code Smell	Found 0 very strong, 0 strong and 0 weak candidates

Figure 3.5: A scan result overview.

but it has the advantage that it can keep the scanners small and modular. Some of the principles cannot be implemented using modular scanners because the principles are elemental aspects themselves. Some cannot be split into smaller aspects and will need to be treated as

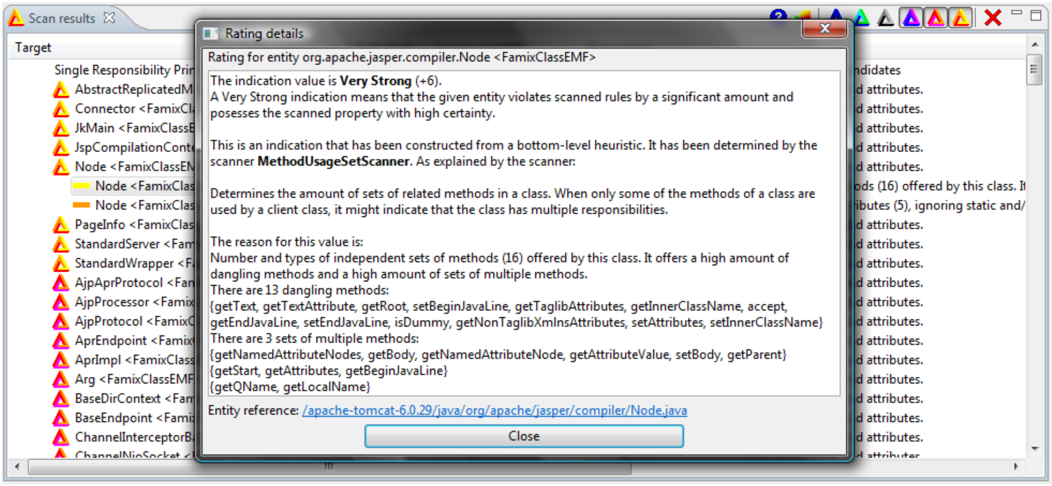

Scan results

Target	Indication	Source scanner	Description
The Blob Anti-Pattern - Result overview	Strong	The Blob Anti-Pattern	Found 0 very strong, 1 strong and 43 weak candidates
PageInfo <FamixClassEMF>	Strong	TheBlob	Indication that the target class is a Blob class.Weighted combination of scann
PageInfo <FamixClassEMF>	Very Strong	SingleResponsibility	Average rating of responsibilities of methods and attributes.
PageInfo <FamixClassEMF>	Very Strong	MethodUsageSetScanner	Number and types of independent sets of methods (40) offered by this class. It
PageInfo <FamixClassEMF>	Strong	UnrelatedAttributes	Unrelated sets of attributes (2) and dangling attributes (11), ignoring static anc
PageInfo <FamixClassEMF>	Strong	LargeClass	Indication that the class is large.
PageInfo <FamixClassEMF>	Very Strong	ManyMethods	High amount of methods (84)
PageInfo <FamixClassEMF>	Weak	ManyAttributes	High amount of attributes (45)
PageInfo <FamixClassEMF>	Neutral	BlobRelations	Indication that the given class uses simple classes as dependencies.There are 9
PageInfo <FamixClassEMF>	Neutral	ManyDistinctRelations	Proper amount of distinct class relations: 9.
ExpressionFactory <FamixClassEMF>	Strong	SimpleClass	Combined rating of class properties.
ExpressionFactory <FamixClassEMF>	Very Strong	SimpleMethods	Ratio of simple methods (5) versus non-simple methods (0).Simple methods:{
ExpressionFactory <FamixClassEMF>	Neutral	PrimitiveAttributes	No attributes found.Ignoring final attributes (0).
Mark <FamixClassEMF>	Weak	SimpleClass	Combined rating of class properties.
Node <FamixClassEMF>	Weak	SimpleClass	Combined rating of class properties.
TagLibraryInfo <FamixClassEMF>	Weak	SimpleClass	Combined rating of class properties.
ErrorDispatcher <FamixClassEMF>	Neutral	SimpleClass	Combined rating of class properties.
ErrorDispatcher <FamixClassEMF>	Weak	PrimitiveAttributes	Weighted sum of proper (1), primitive/unprotected (1) and improper (0) attrib
ErrorDispatcher <FamixClassEMF>	Negative	SimpleMethods	Ratio of simple methods (1) versus non-simple methods (23).Simple methods:
BeanRepository <FamixClassEMF>	Negative	SimpleClass	Combined rating of class properties.
ExpressionFactoryImpl <FamixClassEMF>	Negative	SimpleClass	Combined rating of class properties.
Node$PageDirective <FamixClassEMF>	Negative	SimpleClass	Combined rating of class properties.
AbstractReplicatedMap <FamixClassEMF>	Weak	TheBlob	Indication that the target class is a Blob class.Weighted combination of scann
AprEndpoint <FamixClassEMF>	Weak	TheBlob	Indication that the target class is a Blob class.Weighted combination of scann

Figure 3.6: A heuristic decision tree for The Blob Anti-Pattern.

a bottom-level scanner.

Even though not all problem types can benefit from modularity of the system, they can still give a clear indication of their reasoning. An example reasoning a subcomponent of the Single Responsibility Principle is shown in Figure 3.7. All entries from any scanner can produce such a dialog, meaning that all scanners can provide an explicit explanation in addition to the reasoning tree structure.


Rating details

Rating for entity org.apache.jasper.compiler.Node <FamixClassEMF>

The indication value is **Very Strong** (+6).
A Very Strong indication means that the given entity violates scanned rules by a significant amount and posesses the scanned property with high certainty.

This is an indication that has been constructed from a bottom-level heuristic. It has been determined by the scanner **MethodUsageSetScanner**. As explained by the scanner:

Determines the amount of sets of related methods in a class. When only some of the methods of a class are used by a client class, it might indicate that the class has multiple responsibilities.

The reason for this value is:
Number and types of independent sets of methods (16) offered by this class. It offers a high amount of dangling methods and a high amount of sets of multiple methods.
There are 13 dangling methods:
{getText, getTextAttribute, getRoot, setBeginJavaLine, getTaglibAttributes, getInnerClassName, accept, getEndJavaLine, setEndJavaLine, isDummy, getNonTaglibXmlnsAttributes, setAttributes, setInnerClassName}
There are 3 sets of multiple methods:
{getNamedAttributeNodes, getBody, getNamedAttributeNode, getAttributeValue, setBody, getParent}
{getStart, getAttributes, getBeginJavaLine}
{getQName, getLocalName}

Entity reference: /apache-tomcat-6.0.29/java/org/apache/jasper/compiler/Node.java

Figure 3.7: An explanation dialog of an aspect of the SRP.

The Anti-Pattern Scanner and the heuristics are as effective as expected and as accurate as hoped for. The requirements stated in Section 3.1 have been met. The current version should be seen as a prototype for research purposes and does have some severe shortcomings, even though the results so far are satisfactory. Potential improvements for this program are reduction of the amounts of false-positive indications (see Section 5.2), improvements to user experience (previous section) and possible tweaks to the heuristics such as solutions to the detected false-positives. In general, should development continue on this program, virtually every aspect of the program could be improved.

Possibly the most significant design decision for the Anti-Pattern Scanner is that this program uses Java classes to implement the heuristics. The heuristics are scripted inside the scanner classes. Other, similar programs (see Section 2.4) use alternative approaches, such as providing template graphs of patterns and matching the graphs against source code, or by providing a library of operators and allowing the user to specify formula using these operators. The main advantages of the approach of the Anti-Pattern Scanner is that any Java programmer can contribute to scanners, without having to learn alternative representations or languages. A design of a pattern can be coded directly into scanner classes. The disadvantages are that this approach does not allow for simple externalization or distribution of the detection logic. Instead of using templates or formula sheets, the Java approach requires hard-coded heuristics. The decision for choosing this approach is probably the best, given the availability of an abstract model (the Famix model) and the ease of programming. This choice has affected the entire approach of this project as all heuristics have been designed to match this approach.

Chapter 4

Detection strategies

This chapter contains an overview of the Code Smells and Anti-Patterns that have been included in the Anti-Pattern Scanner. It also contains an overview of Design Principles even though not all of them have been implemented. Most of the principles could be implemented in the scanner as it is, some would require the addition of Mining Software Repositories (MSR). In some cases, no heuristic is available that could work with the implementation of the Anti-Pattern Scanner. All of the principles and patterns contain an explanation, a description of the suggested heuristic and implementation information. If the heuristic has been implemented, that information describes the scanners and ratings that resemble the heuristic.

All suggested heuristics are designed to work with the abstract meta-model that is used for scanning. This abstract model was chosen as a basis for the scanner design because it makes the scan procedures considerably easier. The limitations of this model are that some exact numbers are no longer available, such as the amount of lines of code in a class, the number of words etc.

Many of the heuristics mention 'relations', 'references' or 'dependencies'. The relations of a class are the other classes that it depends on. For example, in the Java programming language, 'String' depends on 'Object'. The relations are found by examining the way entities of the meta-model relate to each other. The relations of a class are defined as the sum of the following:

- The types of the attributes of the class.
- (Anonymous) inner classes.
- The relations of (anonymous) inner classes.
- The types of attributes accessed by the methods of the class.
- The types of parameters of the methods.
- The class types in instance-of checks in the methods.
- The types of local variables in the methods.
- The types of type-casts in the methods.
- The types of any class or object on which methods are invoked.
- The return types of the methods.

The resulting list of class types are the relations or dependencies of a class. The set of unique relations is the same list without any duplicate types. A ‘custom class’ relation is a relation to any class that is defined in the project source, meaning that does not originate from the programming language or a library package.

4.1 Code Smells

For use in Anti-Pattern detection heuristics, two Code Smells have been included to the list of heuristics. This section contains an explanation of the suggested heuristics and of the reasoning behind the suggested strategy. Only ‘used’ smells are explained, more smells can be found in the book ‘Refactoring. Improving the Design of Existing Code’[14].

Code Smells are often very easy to implement because many of them can be based on metrics and thresholds. Example metric heuristics can be found in the book ‘Object-Oriented Metrics in Practice’[23]. Most of the listed heuristic parts can be converted to concrete implementations for the Anti-Pattern Scanner. Heuristic parts can be combined using fuzzy ‘and’ conditions (typically averaging them) and ‘or’ conditions (taking the worst rating). For more information on the implemented ‘and’ and ‘or’ conditions, see Section 3.1.3.

4.1.1 Data Class

The Data Class Code Smell [14] is an indication that a class is only used to store data. The class has attributes which contain the data. The attributes are either not kept private or there are getter and setter methods that allow access to the data. The class itself does not do anything with the data. Processing and checking of data is done by other classes that control the data class.

Heuristic

To find Data Classes, the following properties are checked:

- A class has attributes.
- The class does not appear to have responsibilities, such as calculations or complex operations on the data.

In addition to checking those conditions, the number of false-positive and false-negative cases can be reduced by using additional constraints. To find less and more probable cases, the following properties can also be checked: Classes that have only very few attributes, such as only two, are less likely to be Data Classes because they cannot have much data. Also, classes that do not protect their attributes (e.g., public attributes) are more likely to be Data Classes because they cannot bear responsibility for the values of the attributes. When a class only has primitive attributes, such as integer or boolean, it has a higher probability of being a Data Class. In such a case, the class might look like a universal data container.

Implementation

The initial version of The Blob Anti-Pattern included a heuristic for determining whether a class is simple. Simple classes are easily used as 'slave' classes, classes that are just used for storing data or for some minor task. The heuristic for determining whether a class is simple also doubles as a heuristic for detecting classes with the Data Class Code Smell. Figure 4.1 shows the implementation of this smell. This implementation consists of two parts. The first part is a filter that removes any obvious false-positives from the results. The second part contains a heuristic for finding suspicious 'simplistic' classes.

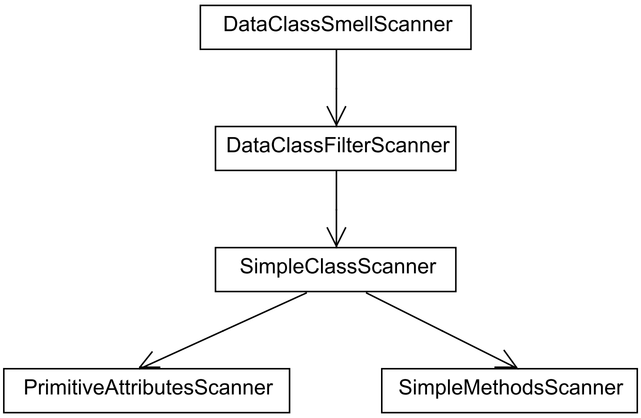


Figure 4.1: Data Class scanner layout.

Simple classes

Simple 'slave' classes can typically be found near a Blob. The classes are expected to possess all properties of a Data Class, except for those cases for which a filter has been applied. Determining whether classes are 'simple' is done in three steps.

The first step is to determine, whether a class uses simple attributes. The *PrimitiveAttributesScanner* produces a rating for this step. The following attribute types are considered simple: String, Object, byte, short, int, long, float, double, boolean and char. These types are either primitives or universal objects that are commonly used in applications. Types that extend Object are not simple. This scanner also determines whether the attributes are properly protected. If attributes are not private, it might suggest that the class does not bear responsibility for them. Slave classes for The Blob typically have no responsibilities. The *PrimitiveAttributesScanner* classifies the attributes in a class:

- Attributes declared 'final' are ignored because they are less suspectable to abuse by a Blob and they probably have a special purpose in the class.
- Non-simple, well-protected types are considered 'good'. These attributes indicate some deeper responsibility for a class.

- Types that are either simple or that are not private are considered ‘bad’. They increase the probability that a class can be used as slave.
- Types that are both simple and not private are ‘very bad’. This indicates that the class is used as a simple storage structure.

A ‘balance’ calculation determines how strongly an entity tilts towards ‘good’ or ‘bad’. This balance calculation is formulated as follows: $Balance = |Bad| + 3 * |Verybad| - |Good|$.

The scanner produces a rating based on the following heuristic:

Strong Negative N/A.
Negative Either of these:

- Only ‘good’ attributes in the class: $|Bad| + |Verybad| = 0$ and $|Good| > 0$.
- Balance of ‘good’ versus ‘bad’: Strongly towards ‘good’ ($<= -3$).

Neutral No attributes or only final attributes.
Weak Positive Balance of ‘good’ versus ‘bad’: Slightly bad or better balance ($<= 3$).
Strong Positive Either of these:

- No ‘good’ attributes in the class and no ‘very bad’ attributes: $|Bad| > 0$ and $|Verybad| + |Good| = 0$.
- Balance of ‘good’ versus ‘bad’: Strong favor for ‘bad’ attributes ($<= 10$).

Very Strong Positive Either of these:

- No ‘good’ attributes in the class and some ‘very bad’ attributes: $|Verybad| > 0$ and $|Good| = 0$.
- Balance of ‘good’ versus ‘bad’: Very strong favor for ‘bad’ attributes (> 10).

The next step is to determine ratings for each of the classes for how simple their methods are. This operation is performed by *SimpleMethodsScanner*. A method is considered simple when each of the following conditions are met:

- It does not invoke any other methods.
- It does not use more than two local variables.
- It does not use more than one class attribute.

Given the amount of simple and non-simple methods, the following heuristic is used to produce a rating:

Strong Negative N/A.
Negative Either of these:

- Only non-simple methods.
- The ratio of simple versus non-simple methods is better than 2 : 1 (on average, there is more than one non-simple method for every two simple methods).

Neutral No methods.
Weak Positive The ratio of simple versus non-simple methods is better than 3 : 1.
Strong Positive The ratio of simple versus non-simple methods is 3 : 1 or worse ($X : 1$ where $X >= 3$).

Very Strong Positive Only simple methods.

The last step is to combine the first two steps into a final judgement about the simplicity of all of the related classes. This judgement is calculated by *SimpleClassScanner*. This class controls the first two scanners and provides input for them. The rating is weighted with a factor 1 for simple attributes and 1.5 for simple methods. Methods are considered to be more important because they reflect the simplicity of the class better. The weighted average is calculated as the sum of ratings each of which are multiplied with their respective weights. The sum of all weighted ratings is divided by the sum of weights.

Filtering

The top scanner *DataClassFilterScanner* scans for candidates that either have no attributes or only 'final' attributes. Such entities can never be Data Class entities because they cannot be used as data classes. They can only be used for definitions (final attributes) or to execute logic. All other cases fall through the filter and are scanned by the *SimpleClassScanner*.

4.1.2 Large Class

The Large Class Code Smell [14] applies to classes that have grown very long. This smell is similar to the God Class Code Smell [23] although a Large Class does not necessarily try to control other classes like a God Class does.

Heuristic

Class size can be measured in multiple ways, such as by using the metric 'number of lines of code' or 'number of methods'. The suggested heuristic for determining class size is by counting the number of methods in a class and the number of attributes. The class size can be determined by evaluating both properties using an 'and' condition. This means that a class is considered large with strong confidence when both attributes exceed threshold values and the confidence decreases when either attribute does not exceed threshold values. Vice versa, there is strong confidence that the class is not a large class when both attributes are far below the thresholds. For more details about this averaging, see Section 3.1.3. This representation is a very rough estimate of class size because it does not take into account the size of the methods, complexity of the methods or other characteristics. If such properties are also needed for a pattern or combined smell then a threshold function can be added for that metric and added to the 'and' function.

Threshold values need to be determined for the threshold of any chosen metric. Techniques are available for finding good threshold values [29], although examples of average and outlier metric values can also be obtained in other ways. This matter has also been discussed in [23]. Baseline values have been obtained by using examples provided in literature and by numbers found on web sites about metrics.[1] Keeping in mind that the Anti-Pattern Scanner can determine weak, strong and very strong cases, the baseline values are used to find neutral to weak cases (e.g., those just above average) and multitudes of the baseline

[1] For example http://www.aivosto.com/project/help/pm-list.html

values are used to find stronger cases. The suggested strategy for finding large classes is to find classes that have both many attributes and many methods. Baseline values can be 20 attributes and 15 methods.

Implementation

The Large Class Code Smell is a basic threshold-based heuristic. Figure 4.2 shows the layout of two threshold scanners and their combination.

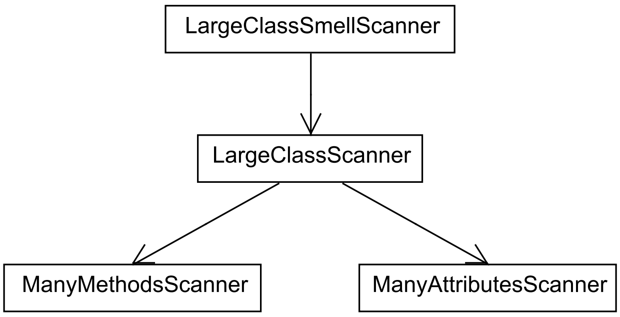


Figure 4.2: Large Class scanner layout.

Thresholds

The class size thresholds are determined by the *ManyMethodsScanner* for the number of methods threshold and the *ManyAttributesScanner* for the number of attributes. The returned ratings are determined as follows for methods:

Strong Negative There are no methods.
Negative All other cases (less than 15 methods).
Neutral There are 15 or more methods.
Weak Positive There are 30 or more methods.
Strong Positive There are 60 or more methods.
Very Strong Positive There are 75 or more methods.

Ratings for number of attributes:

Strong Negative There are no attributes.
Negative All other cases (less than 20 attributes).
Neutral There are 20 or more attributes.
Weak Positive There are 40 or more attributes.
Strong Positive There are 60 or more attributes.
Very Strong Positive There are 80 or more attributes.

Combining the thresholds

The results of the two scanners described above is averaged and the result is returned by the *LargeClassScanner*. Hence, the implementation of this Code Smell is very simple: Two scanners base their judgement on a metric threshold, a third scanner combines the result into a single answer. This scanner structure is among the simplest implementations possible yet it works surprisingly well. It can also be easily adjusted, for example by modifying the threshold values or by adding weights to the averaging process.

4.2 Anti-Patterns

Four Anti-Patterns have been chosen from a long list of patterns. These patterns are implemented in the Anti-Pattern Scanner. Suggestions for more Anti-Patterns can be found on the Internet as well as in literature [5]. The chosen patterns are diverse and are expected to have little overlap. This diversity increases the chance of finding cases of an Anti-Pattern in the software projects that are examined in the empirical evaluation. Also, the chosen patterns should cover a wide range of potential design problems. This allows more problem area's to be discovered that could be examined later, such as in a more focused study. The chosen Anti-Patterns are 'Database Class', 'Ravioli Code', 'The Blob' and 'Tower of Voodoo'. Each of those are explained in the next sections.

4.2.1 Database Class

The Database Class Anti-Pattern[2] is about poor abstraction of a database. It occurs when the database layer or connector classes offer methods that use a query string as input and returns the table contents as output. This is not a problem if used inside a database abstraction layer although it is a problem if it is used to let client classes access a database. It is good practice to create an abstraction of a database, such as by providing a structure of custom objects that do match the data model inside the application, keeping the underlying database table structure hidden from clients.

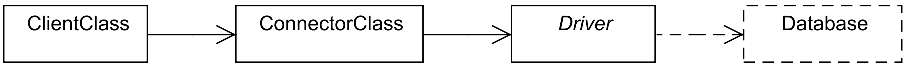


Figure 4.3: Database Class expected structure.

The layout in Figure 4.3 provides an overview of how a Database Class might fit in an application. Every block in the diagram could represent a software entity or a layer in the application. From left to right: Application logic layer, Database abstraction layer, Driver layer, Database layer. The client class(es) contain the application logic and need to tell the lower-level database connector class which queries need to be executed, which operations need to be performed etc. The connector class controls the database driver, such as a driver

[2] http://c2.com/cgi/wiki?DbClass

provided by the database manufacturer. The driver connects to the database itself. In this figure, it is good practice if the connector class would provide an abstract model of the database to ensure that the client class does not directly know and manage the database layout. It would be bad practice if the connector class provides generic access methods to the database, causing the client to do query and fault handling. In the latter case, the connector class is a Database Class according to this Anti-Pattern. A class that directly connects to a database by using a driver will be called a ‘connector class’ in the context of this heuristic.

Heuristic

The concept of the strategy is to find methods that expose a database or allow interaction with a database on a low level. Classes that expose the database without providing an abstraction are marked as ‘Database Classes’. The strategy essentially consists of two parts: finding classes that connect to a database and finding methods that directly expose the database.

Finding classes that connect to a database is context sensitive. Such classes can be recognized by analyzing their dependencies. Connector classes usually depend on a database driver provided as a library. The context is the programming language and the type of database used at bottom level. For example, a Java program might connect to a database by using the java.sql.* package. At this stage, an implementation of the detection strategy may have to be customized for a specific target language.

Finding methods that expose a database without providing an abstraction is more difficult than finding a database connector class. To find such methods, examine all methods of connector classes. Methods that return primitive types, textual strings, arrays of primitive types or language-specific lists are suspicious. The same holds for methods that accept such types as parameters. Methods that either return or have parameters consisting of custom objects are not suspicious because they probably return a specialized structure such as an abstraction. Connector classes that provide suspicious methods are Database Classes.

To reduce false-positive cases in the detection of Database Classes, methods that are only used internally by a connector class can be ignored. Because database abstraction layers or connection managers typically consist of multiple classes, the dependencies should only matter on a package level. To achieve this, packages can be marked as ‘packages that contain connector classes’ and ‘packages that do not contain connector classes’. Packages with connector classes manage the database and can legally use low-level database statements. Packages that do not directly connect to the database should not do so, they should only use abstractions. This rule also provides a good separation between the database layer and the logic layer of an application. The final heuristic is simple: when any class from a non-database connector package invokes any suspicious method from a class in a database connector package, the latter class can be considered a Database Class.

Implementation

The Database Class is the most complicated Anti-Pattern of all implemented patterns. It attempts to automatically identify which packages can legally use direct database communication by examining which packages use the java.sql package. This approach works best if the database connector packages are properly located in a few packages. All classes that do not directly access the database (e.g., logic or UI classes) should not be placed in those packages and they should not directly catch any SQLExceptions. If package allocation is not done properly, there might be more false negative candidates for this Anti-Pattern.

The layout of the scanners is displayed in figure 4.4. The scanners are explained in the sections below.

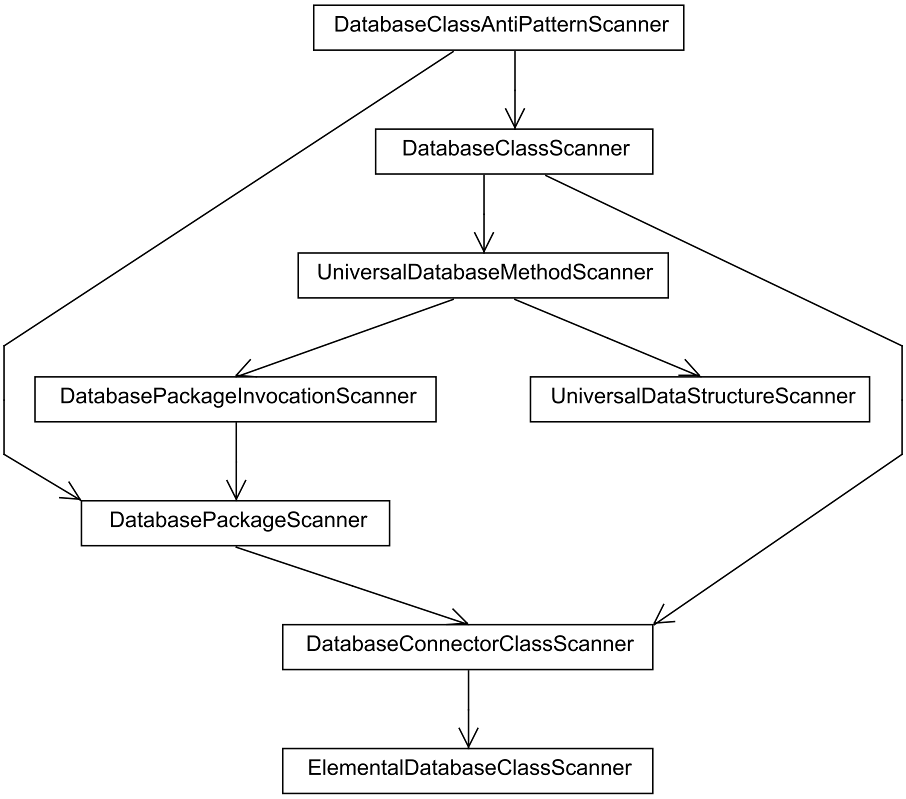

DatabaseClassAntiPatternScanner
DatabaseClassScanner
UniversalDatabaseMethodScanner
DatabasePackageInvocationScanner
UniversalDataStructureScanner
DatabasePackageScanner
DatabaseConnectorClassScanner
ElementalDatabaseClassScanner

Figure 4.4: Database Class scanner layout.

Database connector classes

The *ElementalDatabaseClassScanner* determines whether a given class is a core java.sql.** class. It identifies classes by matching the name of the class against a list of known sql classes that are unique by name (e.g., the name is only found in the java.sql package). This way, it should not confuse sql classes with non-sql classes although it might miss a few cases because of naming conflicts. This scanner has a binary output: either ‘no’ or ‘certainly yes’. The rating will be returned as listed below:

Strong Negative N/A.
Negative The given class is not an elemental database class.
Neutral N/A.
Weak Positive N/A.
Strong Positive The given class is an elemental database class.
Very Strong Positive N/A.

The *DatabaseConnectorClassScanner* examines a class and returns a positive rating if the class uses java.sql components. It invokes the *ElementalDatabaseClassScanner* on each of the class’ relations and returns a rating as listed below:

Strong Negative N/A.
Negative None of the relations is an elemental database class.
Neutral N/A.
Weak Positive One of the relations is an elemental database class.
Strong Positive Two of the relations are elemental database classes.
Very Strong Positive Three or more of the relations are elemental database classes.

The cases of ‘weak’ and ‘strong’ ratings might trigger on classes that handle only exceptions. Some classes handle the generic SQLException, causing them to be rated as ‘weak’. Classes that are more focused on database interaction usually need more components from the java.sql package and automatically classify as ‘strongest’.

A package that contains a class that has direct database communication (e.g., a connector class) is considered a database package. All classes in that package should be seen as database connector classes because all of the classes in the package either use direct database access or are helper classes for doing so. The *DatabasePackageScanner* examines a package and rates the package according to the strongest database connector class rating found in that package.

Invoking a database connector class

Whenever a non-database-connector class invokes a method that belongs to a database connector class, the method should return an abstracted result. This prevents non-database-connector classes from having to deal directly with the database layout or problems. The *NonDatabasePackageInvocationScanner* can only be invoked on methods that belong to a database connector class or a class in a database connector package. This scanner returns a rating that describes how inappropriate the invocations on the scanned method are. To do so, it examines invocations from all other methods in the model. Depending on the package in which the method resides, a *score* is calculated. The rating obtained from the *DatabasePackageScanner* is used to increment the score with the following value:

Strong Negative 5.
Negative 5. This means that a non-database-connector package invokes a database connector method.

Neutral 4.
Weak Positive 3.
Strong Positive 1.
Very Strong Positive -1. This means that a database connector package invokes a database connector method, which is completely safe.

Using this score, the *NonDatabasePackageInvocationScanner* calculates the rating for the method:

Strong Negative N/A.
Negative All other cases.
Neutral N/A.
Weak Positive The score is greater or equal to 2.
Strong Positive The score is greater or equal to 5.
Very Strong Positive The score is greater or equal to 9.

A positive indication means that invocations on the scanned method are inappropriate. This occurs when the method is in a database connector class and it is being invoked from a non-database-connector class.

The *UniversalDataStructureScanner* examines the return type of a method. It returns a rating that indicates how 'bad' it is to use a given data structure as database abstraction.

Strong Negative N/A.
Negative All other cases.
Neutral Return type 'void'.
Weak Positive Return type is a simple type according to *PrimitiveAttributesScanner*.
Strong Positive Return type is one of the Java List variations.
Very Strong Positive Return type is a java.sql.ResultSet.

For any given method, the *UniversalDatabaseMethodScanner* returns the average rating of the *NonDatabasePackageInvocationScanner* and the *UniversalDataStructureScanner*. A strong rating indicates that the method is being invoked by methods in non-database-connector packages and that it returns a generic result type.

Database Class candidates

The *DatabaseClassScanner* scans a given class for the Database Class Anti-Pattern. Any class that is not a database connector class is automatically rejected. If a class is a database connector class, this scanner returns the average rating of all of the *NonDatabasePackageInvocationScanner* invocations on each of class' methods.

The *DatabaseClassAntiPatternScanner* returns the strongest rating of all of the classes, as rated by the *DatabaseClassScanner*. There is a known problem at this point: if database connector classes do not include exception handling then invoking classes are required to do so. This means that classes that are not database connector classes still might use SQLException types. In that case, the *DatabaseConnectorClassScanner* identifies the classes as connector classes because they use classes from the java.sql package. This automatically makes their package a database connector package (decided by *DatabasePackageScanner*) and it causes the *NonDatabasePackageInvocationScanner* to qualify methods are 'good' when invoked from those packages. If the specific SQLExceptions are handled at every package in a program, every package is marked as a database connector package, thus the separation of database connector classes and non-database-connector classes is never seen. There is a simple solution for this problem. When this problem occurs, too many of the packages are rated as database packages. This can be detected and the rating can be adjusted. The *DatabaseClassAntiPatternScanner* the following rating: either the strongest candidate of all classes rated by *DatabaseClassScanner*, or one of the following (whichever is stronger):

Strong Negative N/A.
Negative All other cases.
Neutral N/A.
Weak Positive 20% or more of the packages is a database connector package.
Strong Positive 30% or more of the packages is a database connector package.
Very Strong Positive 40% or more of the packages is a database connector package.

4.2.2 Ravioli Code

Ravioli Code[3] is the Object-Oriented alternative of Spaghetti Code. Ravioli Code occurs where many little classes are related to each other, use many abstractions and delegation. It is not necessarily an Anti-Pattern because it has many of the good aspects of proper Object-Oriented programming: abstraction, delegation, small manageable classes. However, it might be a problem if it is overdone: too many small classes that do virtually nothing or where the user cannot find where an operation actually occurs due to vague abstractions. The problem with Ravioli Code is that the program source can be difficult to comprehend, which may cause problems to a programmer who is new to the program's source. It is unclear whether this pattern is an Anti-Pattern or good practice thus it might be interesting to examine any detected cases.

Heuristic

The question that drives this heuristic is "Is it clear what the given class is doing?". When it is not clear, it might be Ravioli Code. The focus is on the usage of abstract classes and interfaces. Using many of these will make the code loosely coupled, using many concrete classes makes it tightly coupled. To get an indication of how loosely coupled a class is, count the number of loosely coupled relations. If there are many loosely coupled relations, the class might be vague. If the related classes are also vague then the entire structure might be unclear about what it is doing.

Unlike The Blob Anti-Pattern (see below), this pattern is not focused on a central class. The focus is on the related classes because the entire group of relations could be the problem, not just the class in the center of attention. When there are many dependencies, there

[3] http://c2.com/cgi/wiki?RavioliCode

is an increased probability that the given class is not focused and not clear about what it is doing. If those relations are doing rather abstract things, such as by being dependent on many abstract classes or interfaces, then the probability increases. The suggested heuristic for determining whether a class is or participates in Ravioli Code is to focus on the classes it depends on. If those classes are rather abstract, such as by having half or more abstract class or interface dependencies, then they might be vague. Then if there are many of such vague classes that the given class depends on, typically 7 or more unique custom defined types, then the indication for this Anti-Pattern should be positive. If the examined class itself is also vague then this indication can be even stronger. Weighting should be such that if there are few and/or mostly concrete dependencies, then the implementation is focused and clear. If there are many and/or vague dependencies, then the implementation is unfocussed and maybe vague.

Implementation

The Ravioli Code Anti-Pattern scanner implementation examines a given class and determines whether it and directly related classes resemble Ravioli Code. The scanner layout is displayed in Figure 4.5. Details of these scanners are explained in the following sections.

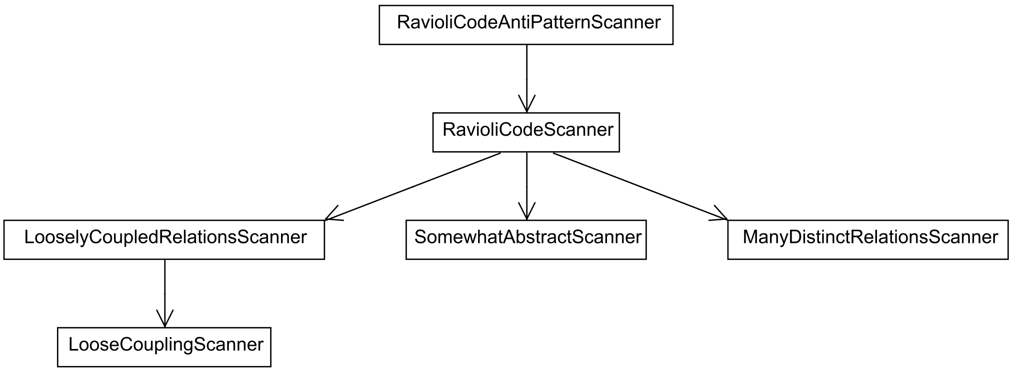

RavioliCodeAntiPatternScanner
RavioliCodeScanner
LooselyCoupledRelationsScanner
SomewhatAbstractScanner
ManyDistinctRelationsScanner
LooseCouplingScanner

Figure 4.5: Ravioli Code scanner layout.

Ravioli classes

The *LooseCouplingScanner* identifies whether a class is loosely coupled. Loosely coupled classes depend on abstract classes or interfaces. In the context of Ravioli Code, loosely coupled also means that a class does do something although it does so in a vague way: it uses abstracts or interfaces to acquire information, delegate work etc. Therefore, this scanner determines how vague the class is. It ignores relations with primitive types or Java system classes because those are not vague, even though they are loosely coupled. This scanner examines the relation with custom classes and gives a strong rating when most of those classes are vague. Using concrete custom classes reduces vagueness.

Strong Negative N/A.
Negative All other cases.
Neutral There are no relations to other classes.
Weak Positive Either of these:

- There are no custom class relations and three or less loose relations.
- More than three loose relation for every custom class relation.

Strong Positive Either of these:

- There are no custom class relations and four or five loose relations.
- More than five loose relation for every custom class relation.

Very Strong Positive Either of these:

- There are no custom class relations and more than five loose relations.
- More than ten loose relation for every custom class relation.

The class that is examined for presence of this Anti-Pattern needs to have it's relations examined for vagueness. The *LooseCouplingScanner* scans each of the relations for this property. These operations are coordinated by the *LooselyCoupledRelationsScanner* which iterates over all the relations of the given class. This scanner returns the average of all ratings. Having many vague classes as relations could mean that there are many Ravioli classes as relations.

Center class

The class that is examined for this Anti-Pattern can also be vague itself. In addition to having Ravioli classes as relations, being somewhat vague by itself can increase the confidence that a class participates in the Anti-Pattern. This property is not essential and it should not weight heavily compared to the ratings of relations. The scanner *SomewhatAbstractScanner* determines how vague a class is. This scanner is less restrictive than the *LooseCouplingScanner* because it does not determine whether the class is loosely coupled. It just determines whether it is somewhat vague, in addition to all the other properties.

Strong Negative N/A.
Negative N/A.
Neutral All other cases.
Weak Positive Number of abstract or interface dependencies equal or greater than 3.
Strong Positive Number of abstract or interface dependencies equal or greater than 7.
Very Strong Positive Number of abstract or interface dependencies equal or greater than 10.

If the class has many relations, there is an increased chance that the structure is not properly designed. The *ManyDistinctRelationsScanner* uses a threshold heuristic to determine whether a class has many unique dependencies.

Strong Negative N/A.
Negative All other cases.

Neutral Number of dependencies equal or greater than 7.
Weak Positive Number of dependencies equal or greater than 10.
Strong Positive Number of dependencies equal or greater than 14.
Very Strong Positive Number of dependencies equal or greater than 20.

Rating the Ravioli Code

For any given class, the *RavioliCodeScanner* determines how heavily it participates in the Anti-Pattern. It's rating is based on a weighted average of all of the results. The number of distinct referenced types has a weight of 1, the rating of the current class being somewhat abstract has a value of 0.8 and the outcome of the number of somewhat abstract relations (*LooselyCoupledRelationsScanner*) is weighted by the number of relations. This means that the weight of the relations increases when there are many. The relations are considered a very important measurement to decide how 'Ravioli' they are. The weight is calculated as $1+0.2*N$ where N is the number of relations. This value is capped at 5.

The final rating is equal to the strongest rating found in all results of class scans. This result is determined by the *RavioliCodeAntiPatternScanner*. De design of this Ravioli Code scanner is such that at a low number of dependencies, the result is rated down by the *ManyDistinctRelationsScanner*. At a high number of dependencies, the result is mainly determined by the *LooselyCoupledRelationsScanner*, meaning that the rating is strongly influenced by how vague the relations are.

4.2.3 The Blob

The Blob[5] is probably the most well-known Anti-Pattern of all. A large class contains most of the logic of the application and uses small 'slave' classes to store data. This classic Anti-Pattern is expected to be present in many software projects, either in lesser or full form.

Heuristic

A Blob class contains a large portion of the responsibilities of a system. Other classes have far less responsibilities and are simply used as data containers. To get a positive indication of a class being a Blob class, a number of properties have to be checked. The first is to verify that the class has multiple responsibilities. A heuristic for estimating the number of responsibilities is explained in the design principles section later in this chapter. In addition to checking for multiple responsibilities, it can be verified that the class is very large. Large classes are more suspectable to becoming a Blob. Next, all dependencies should be examined. Many distinct dependencies can be interpreted as the class having multiple responsibilities and even worse, the class having a central role or controller role in the system. If the dependencies themselves do not have many responsibilities then there is a strong indication of unbalanced class roles within the system. In that case, the Blob class has all of the responsibilities and the dependencies having none. Dependencies that only contain data, such as classes with the Data Class Code Smell also contribute to this.

Optionally it can also be checked whether the Blob class is actually a controller for the other, smaller classes. This property however is not crucial because the earlier checks have already shown that class size may be unbalanced, which will cover the most significant aspects of the Blob Anti-Pattern. The proposed strategy for detecting whether a class is a controller of another class is to check that the controller class uses most of the functions or attributes offered by the class being controlled. If the controller class is the only user of those functions or attributes, the indication will be even stronger.

If all of the criteria evaluate positively then there is a strong indication of the class being a Blob. If only some of the criteria evaluate positively then there may be a lesser form of a Blob such as a cluster of classes having unbalanced class size and responsibilities.

Implementation

Figure 4.6 shows the layout of the scanner structure for The Blob detection. The top level scanner is the *TheBlobAntiPatternScanner* which is a pattern scanner. Intermediate and leaf nodes scan aspects of The Blob and those scanners will be explained in this section.

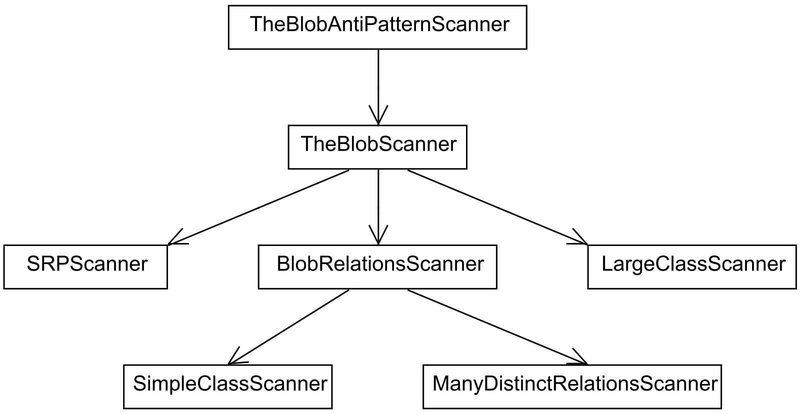


Figure 4.6: The Blob scanner layout.

The Blob class size

The candidate Blob class is scanned for class size and for unbalanced role assignments. The size and behavior are determined by two scanners: the *SRPScanner* which determines whether the class has too many responsibilities, and the *LargeClassScanner* which determines whether the class has the Large Class Code Smell. For more information about these scanners, see the Single Responsibility Principle further in this chapter and the Large Class Code Smell earlier in this chapter.

Slave classes

If the role assignments are not properly balanced in the application, classes related to the potential Blob class may have too few roles. The *SimpleClassScanner* determines whether related classes are being used as slave classes by examining how much logic they possess.

This scanner is part of the Data Class Code Smell presented earlier. The main difference with this smell is that in the context of The Blob, slave classes are not limited to Data Classes. Instead, any simplistic class will do.

To *BlobRelationsScanner* averages the simplicity rating for all classes that the candidate class depends on. It returns a positive indication if many of the related classes could be slave classes. This scanner filters out library and system classes because those classes would cause false-positives. Such classes are always simple because their member attributes and functions are not included in the model.

If there are very few related classes and those few are considered simple, then averaging the ratings would result in a biased rating. To prevent the *BlobRelationsScanner* from reporting a positive result if only a few relations are found, the *ManyDistinctRelationsScanner* is added to the averaging process. This is the same scanner as included in the Ravioli Code Anti-Pattern and it is added for the same reason: to limit positive indications if the amount of sampled classes is small. The

Rating a Blob

The final Blob rating is a combination of each of the previous scanners. The scanner *TheBlobScanner* controls each of the other scanners. The simplicity of related classes weights heavily compared to the size of the Blob class itself. The *TheBlobScanner* weights the rating of the relations 150% of that of the class size. These relations are considered the most important aspect of the Anti-Pattern because without slave classes, a Blob would just be a Large Class.

4.2.4 Tower of Voodoo

A Tower of Voodoo[4] is an inheritance structure where multiple, possibly badly designed wrapping or extension layers are built upon a well-designed foundation. Unnecessary and poorly defined extensions are supposed to complement the base class. The problems with such extensions are that it might lose the intended meaning of the structure, it might perform poorly and require more extension to suit new goals.

There are two variations of the Tower of Voodoo. The first is encapsulation, where a class has another class as attribute and maps it's own functions to those of the encapsulated class, possibly modifying, removing or adding to the functions offered by the encapsulated class. The second variation is inheritance, where a class extends another class, adding to or modifying the functions of the base class. Only a heuristic for the second version is presented in this document.

Heuristic

The goal of this heuristic is to find extension structures that appear to not be well-defined. Structures that contain multiple levels of inheritance are considered suspicious. Inheritance is a core feature of Object-Oriented programming and the heuristic should not punish proper

[4] http://c2.com/cgi/wiki?TowerOfVoodoo

usage. Improper usage, thus badly defined structures typically occur when the designer or programmer has not taken the time to properly evaluate the added value and design of the new level of inheritance. Also, it might occur if the designer has not read the design information of the base class or not understood it's working. In such a case, the added level can be a quick fix to get the base class working for a specific purpose. Due to the lack of a heuristic to find 'quick fixes' or 'dirty solutions', the suggested heuristic is limited to evaluating depth of inheritance, taking into consideration the use of abstract levels and implemented interfaces.

To find poor inheritance structures, number of inheritance levels is counted. Extending a base 'Object' can be ignored. Three levels of inheritance is already poor because there is rarely a reason to create three levels. Four levels of inheritance is considered to be bad. These numbers are deducted from levels of inheritance found on web sites with metric overviews. There are a number of important exceptions. Extending an abstract class, thus an abstract level, should not count towards the threshold. Abstract classes are usually well-defined and explicitly meant to be extended. The same holds for interfaces. Any extending class that implements one or more interfaces should not count towards the threshold. The interface clearly defines the added value of the class thus it is a sign of good practice instead of Tower of Voodoo. Remaining candidates typically have unnecessarily high extension levels or they are not explicitly defined using abstract or interface definitions.

Implementation

The current implementation of the Tower of Voodoo Anti-Pattern identifies long inheritance structures. This might not cover all cases of Tower of Voodoo as creating a 'quick and dirty' extension to a Java core component could also be considered a Tower of Voodoo. There is currently no heuristic for 'quick and dirty'. Such cases might be related to the Anti-Pattern 'Junk Yard Coding'.

The layout of the Tower of Voodoo scanners is shown in Figure 4.7.

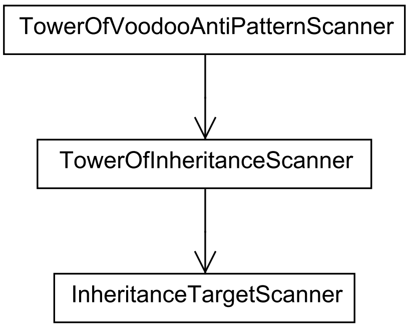


Figure 4.7: Tower of Voodoo scanner layout.

Inheritance severity

The *InheritanceTargetScanner* examines the type of class of the entity it receives as input. Based on that type, it returns a rating of how 'bad' it is to inherit from that type. It is good practice to implement an interface because interfaces are meant to be implemented. Extending an abstract is neutral behavior because they are meant to be extended although it does add a level of complexity. Other cases are weak, meaning that extending such classes results in adding complexity to a structure, using classes that are not explicitly meant to be extended.

Strong Negative N/A.
Negative N/A.
Neutral Class is an abstract or it is 'Object'.
Weak Positive All other cases.
Strong Positive N/A.
Very Strong Positive N/A.

Inherited severity

Given a class, the *TowerOfInheritanceScanner* scans all inheritance structures of a class. It does so by recursively invoking itself on the supertype of the class. It takes that result and adjusts it by the rating as given by the *InheritanceTargetScanner* for that superclass as explained above. This means that the base class is evaluated using the *InheritanceTargetScanner*, that rating is adjusted by the rating for the child and these adjustments are repeated up to the parent of the given class. The adjustments stack, meaning that multiple levels of 'weak' ratings from the *InheritanceTargetScanner* will eventually result in a 'very strong' rating. One exception is that, if a class does implement interfaces, it's adjustment value is lowered by one. This is to encourage the use of interfaces in inheritance structures since interfaces can provide clear definitions of what the implementing class adds to the structure. The adjustment values are:

Strong Negative -2.
Negative -1.
Neutral 0.
Weak Positive 1.
Strong Positive 2.
Very Strong Positive 3.

The resulting rating is the worst rating for all inheritance structures. If no inheritance structure is found, the rating will be negative. This ensures that all inheritance structures start at negative, allowing to the first level of inheritance to be 'free'. Extending a class once is considered good practice, hence the first level results in a negative indication.

The final result is returned by the *TowerOfVoodooAntiPatternScanner*. This scanner returns the worst result of all invocations of the *TowerOfInheritanceScanner*, meaning that it will return the rating of the worst Tower of Voodo instance found.

4.3 Class Design Principles

There are five class design principles, also called the S.O.L.I.D. principles. These principles suggest how classes should be designed, how they should interact with each other and which conditions should hold for good reusability. Suggested heuristics for these principles are explained in the following sections. Details and reasoning behind the heuristics are not included in these sections, more information about the principles can be found in the textbook [26].

4.3.1 Single Responsibility Principle

The Single Responsibility Principle (SRP) states that every class should only have one reason for change. Every responsibility of a class is an axis of change. Classes that have multiple responsibilities can be split into multiple classes, each having only one reason for change [26].

Heuristic

To find axis of change, software can be analyzed over multiple versions to see which pieces change together. When a single change to the design would cause a number of small changes to the implementation, those changes can be grouped as a change set or axis of change. Finding change sets requires MSR. Even more difficult is to find actual reasons for change, which is needed for determining the responsibilities of a class. Instead of using MSR, an alternative heuristic can be used to estimate the responsibilities of a class. The strategy is to examine how a class is used. Every way of using a class can be associated with one responsibility for that class.

A strategy without MSR is to determine sets of responsibilities by determining which components of a class are used together. There are two indicators for class usage that can be placed in sets: the attributes and the methods. The concept is the same for both however the methods are slightly more difficult to implement because they can be used in more different ways than the attributes. The strategy is to determine which items are used together and to place them in the same set. For example, attribute A and B are used in the same method, attribute C is not. That would place attribute A and B in one set and C in another. That means that A and B probably belong to the same responsibility because they are related by being necessary within one method body. Attribute C stands alone, which indicates that it is not directly related to A and B.

Attributes or methods that ‘stand alone’ are expected to occur frequently. Many operations need only one of the methods or operations offered by a class and the class might offer overloaded versions of which only one is needed in any given situation. Having multiple stand-alone attributes or operations still might be an indication that the class is offering multiple services and thus having multiple responsibilities. Multiple sets of multiple attributes or methods is a better indication because it provides a stronger evidence that a given service consisting of multiple elements, is different from another service that consists of multiple elements. For example, a set of [A, B, C] and [D, E, F] would indicate that A+B+C are together one service and D+E+F another.

The heuristic boils down to identifying the sets of attributes and methods. One set per type can be considered good practice because the class would have exactly one responsibility. Two sets is in violation with this principle because that would mean the attributes or method sets are not directly related. More than two sets is bad and the severity of this principle violation increases as more sets are identified.

Implementation

The Single Responsibility Principle scanners can give an approximation of the number of responsibilities of a class. Figure 4.8 shows the layout of the scanners for this principle. The heuristic for finding violations of this principle works by finding independent sets of attributes and of methods in a class.

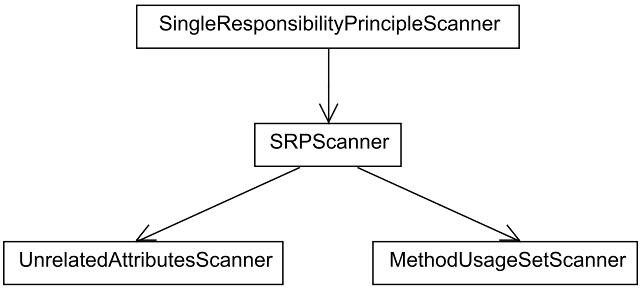

SingleResponsibilityPrincipleScanner
SRPScanner
UnrelatedAttributesScanner
MethodUsageSetScanner

Figure 4.8: Single Responsibility Principle scanner layout.

Responsibilities

The responsibilities of a class are determined by identifying independent sets of attributes and methods. The approach is to find attributes that are used inside methods. Attributes that are used *together* in the same method are placed in the same set. This yields sets of attributes that can indicate the amount of responsibilities. The *UnrelatedAttributesScanner* finds sets of attributes.

Strong Negative There are no attributes.

Negative There is exactly one set of attributes (e.g., all attributes are used in a combination with at least one other).

Neutral There are up to five sets of attributes, one of which can be of any size, the others are no larger than one attribute.

Weak Positive More than five sets of attributes, one of which can be of any size, the others are no larger than one attribute.

Strong Positive There are three or less sets that contain more than one attribute.

Very Strong Positive Any number of sets with more than one attribute.

As can be seen from this list, cases where multiple large sets of attributes are found have a stronger indication towards a violation of the principle. Stand-alone ‘dangling’ attributes are considered less severe. The scanner ignores static or final attributes because they are expected to have a special meaning in the class, such as defining constants. It may not be appropriate to consider them to be a responsibility, especially because tracking of system-wide constants may be inaccurate.

The *MethodUsageSetScanner* applies a similar tactic to methods. This scanner identifies which program components could use the methods, taking into account package placement and method visibility according to the Java Language Specification. Then, it groups usable methods into sets similar to those of attributes.

Strong Negative N/A.
Negative All other cases.
Neutral N/A.
Weak Positive Either of these:

- Some classes use all of the methods offered by the given class, some use only a subset of the methods.
- There are no classes that use all of the methods offered by the given class although there are not many sets of methods (see points below).

Strong Positive There are no classes that use all of the methods offered by the given class. Furthermore, there are either 12 or more sets of 1 ‘dangling’ methods or there are 3 or more sets of multiple methods.
Very Strong Positive There are no classes that use all of the methods offered by the given class. Furthermore, there are both 12 or more sets of 1 ‘dangling’ methods and there are 3 or more sets of multiple methods.

Combined rating

The scanner *SRPScanner* calculates the average of the two previous scanners. This average is the resulting judgement of the given class. An exception is that this scanner returns Strong Negative if the given class is not directly placed in a package. This is typically the case for anonymous inner classes or system classes. This exception is implemented because the underlying scanner *MethodUsageSetScanner* expects that classes are directly inside a package. This is necessary to determine method visibility for client classes.

4.3.2 Open-Closed Principle

The Open-Closed Principle (OCP) states that a reusable class should be open for extension but closed for modification. Code that is working should not be changed. Functionality can be added by extending the code instead of changing it. This principle prevents the introduction of new bugs in existing code. It also prevents fixing bugs in existing code. Mainly for this reason, true closure may not be fully achieved although it is a good class design guideline.

Heuristic

There are two strategies for detecting violations of the OCP. The first is by finding symptoms of violations such as changes to code caused by the implementation to dependent code. The second is to find bad design decisions that could potentially cause closure to fail in the future.

The first strategy is to apply MSR to find violations. A class or package that has been introduced in an early generation of the project should remain unchanged after it has been completed. If another class or package is introduced to the system, and the earlier package needs to be updated to ensure proper working, then the closure principle of the first class or package has been violated.

The second strategy is to look inside the program's code to find potential problems that could cause the closure principle to fail. Variables that are not kept private, instance-of checks that do not refer to abstract types or dependencies on concrete classes outside the current package are potential troublemakers. Such concrete dependencies can cause the constant and 'closed' behavior of the implementation to fail. Instead, it should depend on abstract types to allow for open modifications without disturbing existing functionality. The heuristic is simple: the more of such problems are found, the higher is the chance to cause the closure principle to fail. A suggested relaxation of this heuristic is to ignore dependencies within the package as packages are often seen as one software entity.

Implementation

No implementation of this heuristic exists. The second version could be implemented in the Anti-Pattern Scanner as it is now, the first version would require MSR.

4.3.3 Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that an instance of a derived class should be able to replace any instance of its superclass. To comply to this principle, functions that use references to a base class must be able to use objects of a derived class without knowing it. Furthermore, derived functions can have equal or weaker preconditions and equal or stronger postconditions.

Heuristic

The first part of this principle is to ensure derived classes do not harm the behavior of base classes. Within the context of this project, taking into consideration the use of the abstract Famix model and the limitations of static analysis, a reliable heuristic is not possible. Thorough analysis of class behavior and the effects in inheritance structures is very complex and out of the scope of this project.

The second part of this principle is to ensure preconditions and postconditions are compatible and not more restricting in overridden methods or extended classes. Like class behavior analysis, pre- and postcondition analysis is out of the scope of this project. Thus, no heuristic is available.

Implementation

Most compilers have checks to enforce overriding methods is safe with respect to signature equivalence. More extensive checks would require good heuristics which are not available.

4.3.4 Interface Segregation Principle

According to the Interface Segregation Principle (ISP), client classes should not be forced to depend upon interfaces that they do not use. Some clients might be enforced to implement interfaces or parts of interfaces that they do not need but that other clients do need. In such a case, the implementations become more tightly coupled than would be necessary and changes to one of the classes might require changes to the other classes even though they themselves have no reason for it.

Heuristic

Violations of the ISP occur when an interface is ‘fat’, causing it to define multiple groups of methods that are intended for different clients. Interfaces with different groups of methods should have been different interfaces, each with one group of methods. To find different groups of methods, the usage of the interface and implementing classes has to be analyzed.

Classes may depend on an interface or on classes that implement the interface. For this reason, for every method defined by an interface, all signature equivalent implementations of that method in both implementing and extending classes should be treated as the same method. Client classes that use one of those interface methods (or overriding methods) should use all of the other methods defined by the interface. If they do then the interface is not ‘fat’ because all clients use it in the same way. When any of the clients do not use all of the methods, that is an indication that the interface contains some redundant methods or that some methods are not strictly needed in all situations. Even worse, if there are disjunct sets of methods with one type of client using one set and another type of client using another set, then it is a clear signal that the interface provides different services. Those disjunct sets should be defined in different interfaces.

This heuristic yields four levels of severity of ISP violations. The first is non-violation: client classes use all of the methods defined by the interface. The second case is a neutral case where client classes depend on the interface but do not use any of the methods. The third case is a weak indication of the interface containing redundant methods. This occurs if some clients do not use all of the methods. The last case is that of the interface having disjunct sets of methods. This occurs when different clients use the interface for different purposes. The indication of this case grows stronger as more disjunct sets are identified.

Implementation

The Interface Segregation Principle scanners attempt to identify ‘fat’ interfaces. Such interfaces define more methods than some of the interface clients use thus the interface is forcing implementing classes to have methods that are not used. Figure 4.9 shows the layout of this scanner.

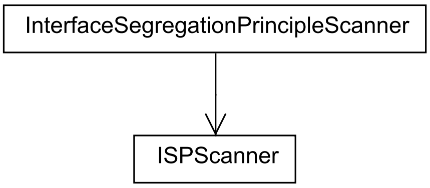


Figure 4.9: Interface Segregation Principle scanner layout.

Interface scanning

The *ISPScanner* examines how interfaces are used. This scanner examines all classes in the model to find which classes extend or implement the methods of the interface. All overridden or implemented versions of any method are treated as the same method for this heuristic. All classes that depend on the interface (and implementing classes) are examined to find which of these methods they use. The result of this scanner is as follows:

Strong Negative Either of these:

- The given class is not an interface.
- The interface does not define any methods (ignoring blacklisted or private methods).

Negative All other cases (there are no cases where only a subset of the methods are used).
Neutral N/A.
Weak Positive None of the dependent classes use any of the methods.
Strong Positive Some dependent classes use all of the methods, some use only a subset.
Very Strong Positive None of the dependent classes use all of the methods, some use only a subset.

4.3.5 Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that the modules that implement a high level policy should not depend on the modules that implement the low level policies, but rather, they should depend on some well-defined interfaces. Both classes and packages can be treated as modules for this heuristic as packages are often used to separate high level policies from low level policies. Policies typically translate to the separation of framework and application logic as well as other components of an application.

Heuristic

This principle can be formulated as three different rules. The first rule is that high level components should not depend on low level components. The second rule is that abstractions should not depend upon details. The last rule is that details should depend upon abstractions. In fact, both high and low level components should depend upon abstractions whenever dependencies are necessary.

To examine the dependencies between high and low level components, a heuristic is needed to identify high and low level components. High and low levels typically map from framework to application logic layers. This means that framework levels should not have dependencies in the direction of application logic and the logic can have dependencies upwards towards framework levels, keeping in mind that these dependencies should be on abstractions. Using this concept, high and low levels can be visualized, either on paper on in computer memory. The suggested visualization is that all classes of an application could be considered nodes in a graph and the dependencies between the classes can be modelled as directional edges. When drawing this graph, nodes that are at a higher levels should be drawn above nodes in a lower level. This height metric can be determined by examining the dependencies: when a class depends on another class, the first should be *below* the latter. This should place logic below framework levels. This relation should be transitive, placing classes that depend on other classes through multiple node hops lower in the graph.

In the ideal case, such a graph should be easy to draw, all low levels should appear below high levels with only dependencies pointing up. In that case, the component design adheres to the DIP. If the graph cannot be perfectly drawn for some classes, those classes might violate the Dependency Inversion Principle. When summing up all the dependencies of a class, some might point upwards and some might point downwards. In that case, the DIP is violated with the strength of the violation dependent on the amount of levels that the dependency edge spans across the graph. If drawing this map is necessary then the height metric must be made mode robust so it can be used even when there are DIP violations. This could be achieved by averaging the dependencies or by drawing the violating dependencies towards dummy nodes. One special case of dependencies is a threat to this heuristic, that case is any violation of the Acyclic Dependency Principle (ASP). The problem with cyclic dependencies is that it makes the graph hard to draw because the height metric cannot be calculated. For more details on this principle, see the ASP section below.

An alternative, very simple although very unreliable heuristic to determine high and low level components is to look at their package naming. Many software designers put high level components in packages named ‘my.application.component.**’ and low level components in packages named ‘my.application.component.subcomponent.category.**’. The heuristic would simply be to count the number of package separator dots, with more dots yielding lower level components. The heuristic for detecting DIP violations would simply be that any class that depends on another class with more dots is in violation with this principle. The reason why this heuristic is unreliable is because package naming does not necessarily reflect how system components relate to each other. Furthermore, package names may not have a meaning, such as in the Java programming language where package names do not have any meaning[5] except to provide namespace separation. Even though the language specification and convention do not add meaning to these package names, many system designers do. For this reason, the heuristic might work just as a rough indication.

The second rule is much easier to check. The rule is that abstractions should not depend upon details. The suggested heuristic is to count the number of concrete dependencies of any interface or abstract class. Dependencies on concrete classes like as language specific

[5] See Java Language Specification (JLS) http://java.sun.com/docs/books/jls/

classes do not count as they do not add (significant) coupling to the application. Dependencies on ‘final’ classes should not weight as heavily as other concrete classes because those constant classes often reflect a special meaning in the application, such as containing constant definitions or serving as an application-wide utility class. Any remaining concrete classes are bad and the violation gets worse as the abstraction is more tightly coupled with concrete classes.

The third rule is that details should depend upon abstractions. Obviously, the heuristic for this rule could simply be the opposite of that of the second rule. By determining how many dependencies a concrete class has on other concrete classes, the severity of any violation can be indicated. To reduce the reporting of non-severe cases of concrete dependencies, this heuristic could be restricted to just examining dependencies between classes in different packages, thus ignoring dependencies within a package. Furthermore, in combination with the first rule, the severity might be increased if dependencies span many levels of high and low components.

Implementation

As this principle consists of three different rules, three implementations are necessary. The first rule has not been implemented because the heuristic is expected to be very unreliable, especially in the current empirical evaluation where many Acyclic Dependency Principle violations are found. Furthermore, time limitations prevent implementation of all heuristics. This heuristic could be implemented as suggested in the Anti-Pattern Scanner.

The implementation of the Dependency Inversion Principle is very limited as it only implements one of the three rules of DIP. Only the rule ‘Abstractions should not depend upon concrete classes’ has been implemented. Figure 4.10 shows the layout of this scanner.

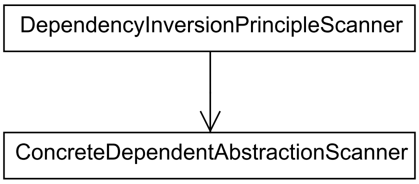

DependencyInversionPrincipleScanner
ConcreteDependentAbstractionScanner

Figure 4.10: Dependency Inversion Principle scanner layout.

Concrete dependencies

The *ConcreteDependentAbstractionScanner* examines abstract classes and interfaces. The process is very simple: the scanner counts all unique dependencies. From those dependencies, abstract classes, interfaces, system classes and concrete classes within the current package are removed. The rating is then determined by the number of remaining dependencies:

Strong Negative The class is not an abstract class and it is not an interface.

Negative There are no concrete dependencies.
Neutral N/A.
Weak Positive There are three or less concrete dependencies.
Strong Positive There are four or more concrete dependencies.
Very Strong Positive There are ten or more concrete dependencies.

The removal of concrete classes within the package of the examined class serves to ignore excessive positive matches for dependencies within the package. These dependencies are usually not problematic. Dependencies on other packages are considered a violation of this principle.

4.4 Package Cohesion Principles

The category Package Cohesion Principles contains three principles that suggest what should or should not put in the same package. These principles are meant to help design the packages optimally for reuse and reduce the impact of changes.

4.4.1 Reuse-Release Equivalence Principle

The Reuse-Release Equivalence Principle (REP) states that the granule of reuse is that same as the granule of release. In other words, for packages to be reused, they should be released as a library and not as separate classes.

Heuristic

This principle focuses on how software components should be packaged, released and reused as one entity. Packages that are released to be used as a library should by imported as a whole and updates should occur to the package as a whole. To check this, libraries and their package or class structures may have to be analyzed, which is not possible with the current setup of the abstract Famix model. Furthermore, MSR may or may not be enough to examine if packages are updated and reused because libraries are often not stored in the same repository as the code that uses the library. For these reasons, there is no heuristic for determining whether packages are properly distributed and used as a library.

Implementation

No implementation possible.

4.4.2 Common Closure Principle

The Common Closure Principle (CCP) states that all classes in a package should be closed against the same kinds of changes. When a change occurs, the change should only affect classes within the package where the change is relevant and it should not affect classes across the program.

Heuristic

To find violations of the CCP, changes can be backtracked and examined. Changes can be associated with each other, such as when a class field changes and methods that depend on that field change to be compatible with the field. Such backtracking is probably the easiest and safest way to identify sets of change. With such information, it can be verified that changes only occur within packages and that a change to one package does not affect other packages. If it does, it would be a violation of the CCP. Also, the change should affect all classes in the package because all of them are closed for the same kinds of changes.

An alternative solution to finding violations would be to scan for change sets before a change actually occurs. However, no reliable heuristic is available for this solution because searching for potential changes or change sets may yield a near infinite search space. Alternatively, it would be possible to verify that changes cannot ripple through to other packages such as by ensuring that no concrete dependencies exist between packages. However, any heuristic for this has a strong overlap with heuristics for DIP and various Package Coupling Principles.

Implementation

Using MSR could yield a good indication of packages adhering to the CCP. Alternative solutions such as scanning the code for potential violations are not implemented although the implementation of DIP can find some violations of this principle.

4.4.3 Common Reuse Principle

According to the Common Reuse Principle (CRP), classes in a package are always reused together. If one of the classes is reused, all are reused.

Heuristic

The word ‘reuse’ may be associated with exporting a component and using it in another application as though it were a code library. Or, it could be associated with having implemented a component as some point for some purpose and in a later version of the software, using the same component again at another place in the program for nearly the same purpose. However, the problem with those interpretations of reuse is that they would require some way to analyze libraries or to use MSR, neither of which are included in the current design of the Anti-Pattern Scanner. Instead, reuse will be defined as ‘every time the same component is used in the current system snapshot’, where usage is defined as another class depending on the component.

Using this approach of reuse, the principle will not be violated if and only if, at every time a class depends on some class of a package, it directly or indirectly uses all other classes of that package. In that case, it reuses all classes of the package. This reasoning can be translated to a heuristic: Look at every class in a package. If any class outside the current package is depending on that class, mark that class. Otherwise, ignore that class (because it is not directly reused). Next, check that any marked class depends on all other classes in the same package, either directly or indirectly. Any class that is not reused from the viewpoint of the marked class is in violation with this principle. That means that the class is not in the reuse set of the package and it should have been in a different package.

Violations of this principle *do not* indicate bad design. Instead, they indicate that a package is not meant to be released because some of the classes are not in every reuse set. It could be in the package for different reasons which cannot be detected by this heuristic. Therefore, this heuristic only applies to packages that are meant to be released or that are meant to be reused within the same application. Furthermore, dependencies between packages are not automatically taken into account. Any dependency from the current package on another package should trigger the heuristic to run on that package too, ensuring that all released packages on their own comply to this principle.

Implementation

The suggested heuristic can be implemented in the Anti-Pattern Scanner. In the current version, it has not been implemented.

4.5 Package Coupling Principles

The category of Package Coupling Principles contains principles that can be used as guidelines to improve package stability and to determine dependencies. This section contains three principles.

4.5.1 Acyclic Dependencies Principle

The Acyclic Dependencies Principle (ADP) states that the dependency structure between packages must not contain cyclic dependencies. This means that the dependency structure between packages can be drawn as an acyclic directed graph.

Heuristic

The description already suggests the solution for detection of violations. The package dependency graph should be an acyclic directed graph. When constructing this graph, if the dependency structure would attempt to add an edge to the graph that would create a cycle, all packages participating in that cycle violate this principle.

Implementation

The Acyclic Dependencies Principle states that the dependencies between packages should be an acyclic directed graph. To find violations of this principle, a scanner has been constructed. Figure 4.11 shows the layout of the scanners.

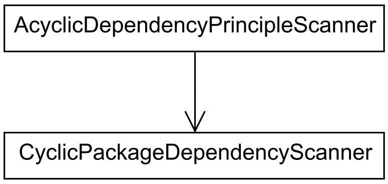

AcyclicDependencyPrincipleScanner
CyclicPackageDependencyScanner

Figure 4.11: Acyclic Dependencies Principle scanner layout.

Cycle detection

The scanner *CyclicPackageDependencyScanner* constructs a tree of package dependencies. The root node of the tree is the current package and the nodes directly connected to the root are the packages that the root package depends on. For every node, this process is repeated until all of the indirect package dependencies have unfolded. For each node, packages that are already in the branch up to the root node are not included thus every package occurs only once in every branch. There is one exception: when the root node is found in the list of dependencies, it is added to the branch and thus terminates that branch.

The tree structure contains all unique dependency structures. By counting the number of possible cycles and the length of the longest cycle, the strength of any indication can be determined. The underlying idea is that a long cycle could impact many more levels of an application compared to just two packages having a cyclic references.

Strong Negative N/A.
Negative There are no cyclic package dependencies.
Neutral N/A.
Weak Positive The longest cycle is of length 2, such as ‘self’ depends on ‘other’ depends on ‘self’ and there are 3 or less such cycles.
Strong Positive Either of these:

- The longest cycle is of length 3 or more.
- There are 4 or more unique cycles.

Very Strong Positive Both the longest cycle is of length 3 or more and there are 4 or more unique cycles.

4.5.2 Stable Dependencies Principle

The Stable Dependencies Principle (SDP) states that dependencies between packages should be such that a package can only depend on packages that are more stable than it is. This means that packages should only depend on packages that are changed less frequently than itself.

Heuristic

A solution without MSR is that speculations about where, which and how often components would need to be modified can be made. Such speculation might be very hard to design because there are numerous reasons and ways components can change. However, the Engineering Notebook[6] that is also used as a basis for [26] offers a metric to estimate *stability*. This metric is:

Ca : **Afferent Couplings** The number of classes outside the package that depend on classes inside the package.

Ce : **Efferent Couplings** The number of classes inside the package that depend on classes outside the package.

I : **Instability** A metric in the range of $[0,1]$ where 0 indicates maximally stable and 1 indicates maximally instable.

The formula for is $I = Ce/(Ce+Ca)$. With this (in)stability metric, it is easy to determine whether package dependencies violate this principle.

An alternative heuristic is to measure the frequency of changes to packages over a period of time, such as every major software release. This change frequency could also be used as an indicator for I. Furthermore, MSR information can be used to extrapolate the change frequency to determine whether the design adheres to this principle in further software releases.

If a map of all dependencies and stability metrics in the system must be drawn then the graph could be similar to that of the DIP heuristic explained earlier. Instead of using the height metric, a stability metric could be used, such as the amount of changes per time period.

Implementation

No implementation exists.

4.5.3 Stable Abstractions Principle

According to the Stable Abstractions Principle (SAP), an architecture should contain as many stable abstractions as possible, and these abstractions should be isolated from the ones that are more likely to change. This means that stable packages should be as abstract as possible, causing them to preserve stability while being able to be extended.

Heuristic

This principle can be rewritten to three rules. The first is that packages that are maximally stable should be maximally abstract. The second is the instable packages should be concrete. The last rule is that package abstraction should be proportional to it's stability. The concept of (in)stability and the corresponding metric I was introduced in the heuristic for SDP as presented earlier. In addition to this metric, the author of the Engineering Notebook suggests that the abstractness of a package can be formulated as the amount of abstract classes in the package divided by the amount of total classes in the package. Obviously, using these metrics, an ordering of abstractness and instability can be created. The three rules can be easily checked using this ordering.

[6] http://objectmentor.com/resources/publishedArticles.html, category Object-Oriented (Robert C. Martin).

Implementation

This strategy has not been implemented.

4.6 Other Principles

The Law of Dementer (LOD[7]) is an interesting principle that does not fit in the categories mentioned above. The next section contains a description of the suggested heuristic for finding violations for this principle. Open discussion on web sites like the Anti-Pattern Catalog Wiki[8] yields the categorization used in this chapter. Suggestions for other principles can also be found there.

4.6.1 The Law of Dementer

The Law of Dementer (LOD) states that classes should only talk to their closest friends, who share it's concerns. This means that units should only have limited knowledge about other units such as only information about the units that are directly related.

Heuristic

The LOD can be applied to classes. In that case, it can be formulated as follows: Inside operation *O* of class *C*, we should only call operations of the following classes:

- Classes of immediate subparts, either computed or stored, of the current object.
- Classes of the argument objects of *O*, including *C*.
- Classes of the object created by *O*.

The heuristic for the LOD is trivial as all of these properties can be checked. All violations count towards violations of the LOD.

Implementation

Checks for LOD have not been implemented.

[7] http://www.ccs.neu.edu/home/lieber/LoD.html

[8] http://www.c2.com/cgi/wiki?PrinciplesOfObjectOrientedDesign

Chapter 5

Empirical evaluation

This chapter describes the empirical evaluation that has been performed using the Anti-Pattern Scanner. First, the approach of this evaluation is explained. Next, a description of the cases is given. Last, we discuss the positive occurrences of the Anti-Patterns and Design Principle violations as reported by the program.

5.1 Case study setup

5.1.1 Goals

As explained in Chapter 1, the goal of this Thesis Project is to examine Anti-Patterns and Design Principles. The Anti-Pattern Scanner has been developed for this purpose. Using this program, case studies are performed. The goal of this evaluation is to explore the occurrence of Anti-Patterns and Design Principle violations. In concrete terms, the goal of this evaluation is to answer the following questions:

- Do Anti-Patterns and Design Principle violations occur in software systems? Are they common or rare?
- Are they severe and full-featured as described in literature or are their occurrences minor problems?
- For those occurrences, can an indication be given whether they are malicious to a project?
- Is the approach of modular ‘scanner’ components and the use of fuzzy ratings effective?

During this study, the performance of the heuristics will be evaluated. The effectiveness of the heuristics is determined, especially with respect to the recall of implemented patterns and the amount of reported false-positive cases. Effective heuristics can be reused in future research and ineffective heuristics can either be abandoned, redesigned or replaced by other approaches.

The study is set up to explore multiple problem aspects in different types of projects. The types of problems are diverse: the selected patterns and principles cover class design, class relations, package cohesion and the occurrence of smells. The results are expected to differ from type to type and some may be more significant than others. Should one type stand out, it may be interesting to focus future studies on that type.

5.1.2 Approach

The evaluation starts out small with specially prepared samples being examined. Stepwise, the examined projects are scaled up from projects that the author has co-developed to larger open-source projects that are developed by other, larger teams. With each iteration, the effectiveness of the heuristics is evaluated and project statistics are being collected. Heuristic designs and implementations are updated between every iteration. After each iteration, the Anti-Pattern Scanner is re-run on the projects of the earlier iterations to see the effect of the updated heuristics. Every case study involves one software system being examined. Cases within each case study, such as entities with positive or negative indications, should provide the sought information during this evaluation.

Detected cases are manually examined. For those projects and problem types where only a few cases have been found, each of the cases is examined. If many cases are found, only ten are selected, both randomly and some selectively. Cases being examined include both positive and negative indications. Affected entities are examined by looking at the source, reviewing the explanation offered by the Anti-Pattern Scanner and examining the properties and relations of the entity. The source is browsed using the Eclipse IDE and relations are visualized using Dependency Analysis for Java (DA4Java) [36]. Examining every case, the following questions are asked:

- By looking at the source, what is the first impression? Is the code clear and self-explanatory? Is the code understood and is the purpose clear?
- Looking at the indication of the Anti-Pattern Scanner, is this case a positive, negative or neutral case? Is this indication correct or incorrect?
- Compare the case reasoning with the source code of the candidate. Is the reasoning correct or does the reasoning mismatch the purpose and characteristics of the source?
- Having compared both the reasoning, the code layout and entity purpose in details, is the indication correct? Is the entity a positive, negative or neutral case? Is the indication strong or weak?
- If the case is positive, false-positive or false-negative, visualize the source. Does it have the properties of the sought pattern? Is there a characteristic that could be added to the heuristic?
- Look at the purpose of the project, the general layout and try to get a feeling for the quality of the implementation. Is a detected case expected or unexpected? Is it a unique case?

In addition to evaluating the heuristics and the projects, the Anti-Pattern Scanner is being evaluated too. The requirements in Chapter 3 are used as a checklist for determining the usability of the tool. Important aspects of the scanner are the processing speed requirements, the effect of having weak and strong cases and the correctness of fuzzy reasoning. The latter is one of the main features of the program and it requires careful evaluation to work correctly. The implementations and use of fuzzy and- and or-conditions, the choice of rating values and weights is carefully being evaluated after every iteration. The values are verified against expected and occurring cases. These features must function properly to ensure that the modular implementations together realize the proposed heuristics.

5.1.3 Studied projects

Eight Java projects have been selected to be examined and scanned for patterns. Sample Anti-Patterns have also been examined for testing purposes although they are not part of the evaluation. Most projects have a practical motivation for examining them, some even have a personal motivation. The following projects are chosen to be studied:

The Anti-Pattern Scanner The first pick is the scanner itself. The reason is that this is the project in which the author has most up-to-date build knowledge. Examining this tool will quickly allow verification of true and false detection candidates.

Race Creation Tool A small Content Management System (CMS) Java Applet that was built to administrate a simple web-based game. While no longer in use, this program was one of the first carefully designed programs that the author wrote. The author has extensive knowledge of it's code. There are a few known problems with this program and it is interesting to see these problems translated to design principles.

The NewNomads[1] Desktop A server-side application framework. This platform has been under development from 2005 to 2010. It has been developed by the author and his colleagues and at the moment of writing, it is still in use although it is no longer being further developed. The reasons for choosing this program are that the author has extensive knowledge about it's structure, it is the work of multiple people and there are some known design problems with it.

Apache Tomcat This application platform is an open-source implementation of the Java Servlet and JavaServer Pages technologies. The author has used this project for many years and has used it's API. It is interesting to examine this project more closely. The version being examined is V6.0.29. For more information about this project, visit
http://tomcat.apache.org/

JForum This open-source forum is a popular discussion board application. It is being used by many hosters, both for small and large companies. This product is receiving active development. The version being examined is V3.0 beta. For more information, visit
http://jforum.net/

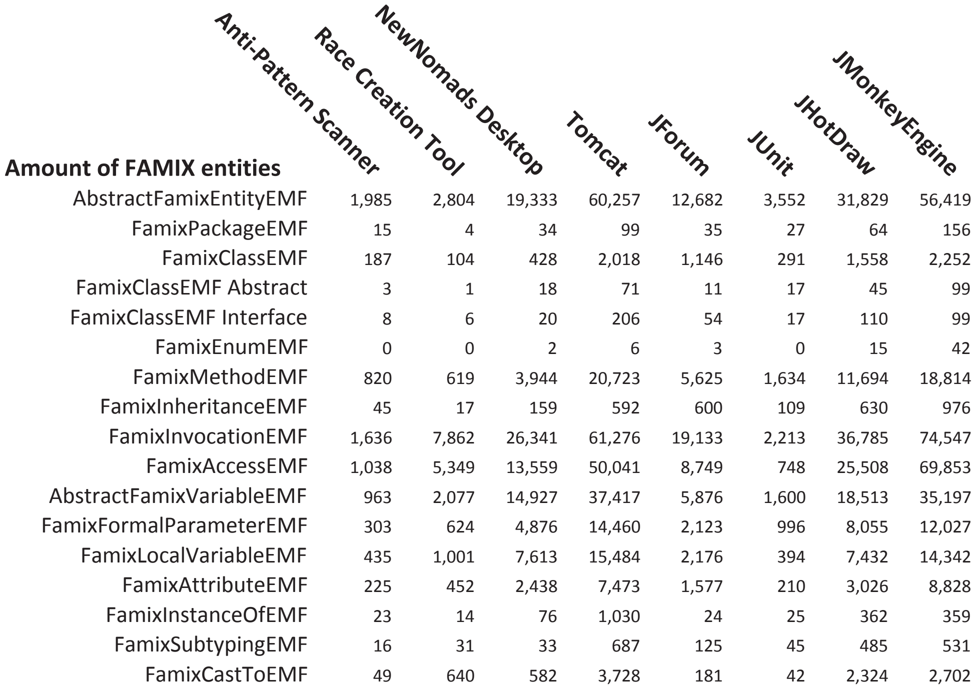
JUnit JUnit is a testing framework for Java applications. It is being used all over the globe for software projects of all sizes. It is also being used for the Anti-Pattern Scanner. The version being examined is V1.8.2. For more information, go to
http://www.junit.org/

JHotDraw This is a GUI framework for drawing technical and structured graphics. The main reason why this project is included in the empirical evaluation is because it has a strong focus on Design Patterns. It was originally being developed for trying out Design Patterns and building an application that could utilize Design Patterns as effectively as possible. The version being examined is V7.5.1. For more information, visit
http://sourceforge.net/projects/jhotdraw/

JMonkeyEngine JME3 is an open-source game engine for 3D games. It is receiving active development and it is being used by various game companies. The version being examined is V3 beta. For more information, see
http://jmonkeyengine.org/

5.1.4 Project size

Project sizes range from a few classes to thousands of classes. The numbers per project are shown in figure 5.1. Project size is determined by the number of Famix entities. Based on this metric, the Race Creation Tool can probably be considered the smallest of all projects and JMonkeyEngine the largest.



Amount of FAMIX entities	Anti-Pattern Scanner	Race Creation Tool	NewNomads Desktop	Tomcat	JForum	JUnit	JHotDraw	JMonkeyEngine
AbstractFamixEntityEMF	1,985	2,804	19,333	60,257	12,682	3,552	31,829	56,419
FamixPackageEMF	15	4	34	99	35	27	64	156
FamixClassEMF	187	104	428	2,018	1,146	291	1,558	2,252
FamixClassEMF Abstract	3	1	18	71	11	17	45	99
FamixClassEMF Interface	8	6	20	206	54	17	110	99
FamixEnumEMF	0	0	2	6	3	0	15	42
FamixMethodEMF	820	619	3,944	20,723	5,625	1,634	11,694	18,814
FamixInheritanceEMF	45	17	159	592	600	109	630	976
FamixInvocationEMF	1,636	7,862	26,341	61,276	19,133	2,213	36,785	74,547
FamixAccessEMF	1,038	5,349	13,559	50,041	8,749	748	25,508	69,853
AbstractFamixVariableEMF	963	2,077	14,927	37,417	5,876	1,600	18,513	35,197
FamixFormalParameterEMF	303	624	4,876	14,460	2,123	996	8,055	12,027
FamixLocalVariableEMF	435	1,001	7,613	15,484	2,176	394	7,432	14,342
FamixAttributeEMF	225	452	2,438	7,473	1,577	210	3,026	8,828
FamixInstanceOfEMF	23	14	76	1,030	24	25	362	359
FamixSubtypingEMF	16	31	33	687	125	45	485	531
FamixCastToEMF	49	640	582	3,728	181	42	2,324	2,702

Figure 5.1: Amounts of Famix entities found in each project.

The list in Figure 5.1 contains the class names of the Famix entities. More information about the entity types can be found at the Famix Importer WIKI:
https://www.evolizer.org/wiki/bin/view/Evolizer/Features/Famix. The AbstractFamixEntityEMF is the superclass of all source entity types: packages, variables, classes and methods. Within this abstraction, the AbstractFamixVariableEMF serves as a superclass for the local variables, formal method parameters and class attributes. The FamixClassEMF is also a superclass for the Enum type and the Interface type.

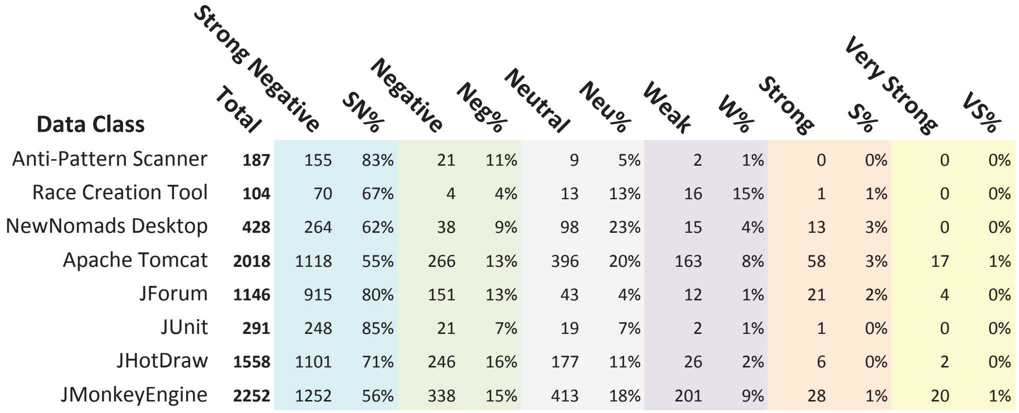
5.2 Detected cases

The implemented patterns listed in Chapter 3 have been used to scan each of the selected Java projects. The results have been evaluated and a selection of the results are presented in this section. Every section that follows contains a quick overview of some of the entities that are found positive by the Anti-Pattern Scanner. Where relevant, false-positive cases are also described. These are the cases that are still being marked as 'positive' by the scanner but that are known to be false. Visualizations of the patterns are included to illustrate what the heuristics are meant to detect as positive cases. The results presented in this section are the results after all the iterations of improving the scanner implementation.

For the projects 'Race Creation Tool' and 'NewNomads Desktop', the source code is not publicly available. Instead of providing the exact names of detected entities, the components are described.

5.2.1 Data Class

Figure 5.2 shows the scan results for this Code Smell.



Data Class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	155	83%	21	11%	9	5%	2	1%	0	0%	0	0%
Race Creation Tool	**104**	70	67%	4	4%	13	13%	16	15%	1	1%	0	0%
NewNomads Desktop	**428**	264	62%	38	9%	98	23%	15	4%	13	3%	0	0%
Apache Tomcat	**2018**	1118	55%	266	13%	396	20%	163	8%	58	3%	17	1%
JForum	**1146**	915	80%	151	13%	43	4%	12	1%	21	2%	4	0%
JUnit	**291**	248	85%	21	7%	19	7%	2	1%	1	0%	0	0%
JHotDraw	**1558**	1101	71%	246	16%	177	11%	26	2%	6	0%	2	0%
JMonkeyEngine	**2252**	1252	56%	338	15%	413	18%	201	9%	28	1%	20	1%

Figure 5.2: Scan results for the Data Class Code Smell.

Positive cases

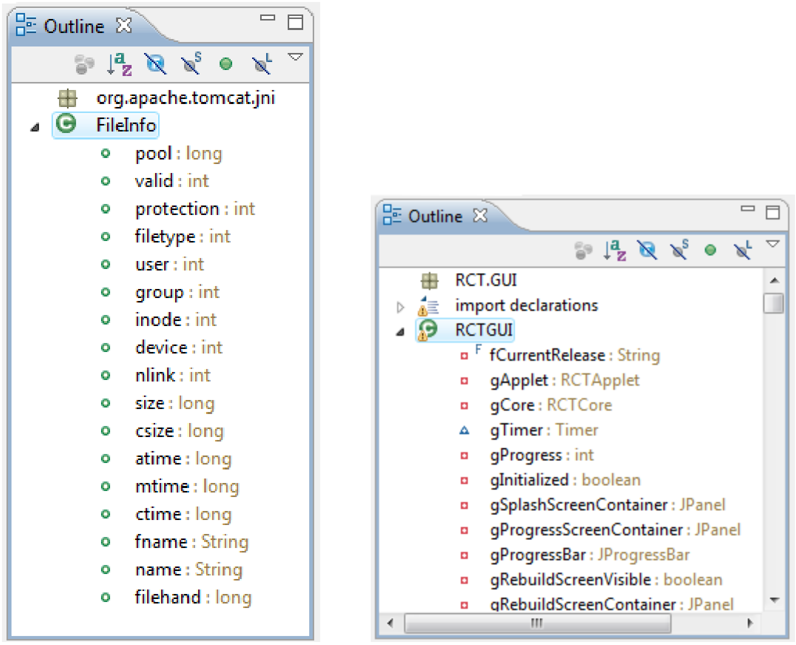
Some of the detected cases:

The Anti-Pattern Scanner ScanSettings.
Race Creation Tool Login data container, all GUI listeners.
NewNomads Desktop Many concrete CRM entry classes, various framework configuration containers.
Apache Tomcat FileInfo.
JForum ForumStats, Theme.

JUnit InexactComparisonCriteria.
JHotDraw ODGStylesReader$Style, StdXMLReader$StackedReader.
JMonkeyEngine AnimationProperties, StateRecord.

Observations

The Data Class Code Smell is a smell that is present in many systems. This can be seen by the number of ‘Strong’ and ‘Very Strong’ cases as shown in Figure 5.2. It is found in all of the systems examined in this empirical evaluation and it will probably be present in many others. Figure 5.3(a) shows what a Data Class might look like.



org.apache.tomcat.jni
FileInfo
pool : long
valid : int
protection : int
filetype : int
user : int
group : int
inode : int
device : int
nlink : int
size : long
csize : long
atime : long
mtime : long
ctime : long
fname : String
name : String
filehand : long

(a) An extreme case of a Data Class.

RCT.GUI
import declarations
RCTGUI
F fCurrentRelease : String
gApplet : RCTApplet
gCore : RCTCore
gTimer : Timer
gProgress : int
gInitialized : boolean
gSplashScreenContainer : JPanel
gProgressScreenContainer : JPanel
gProgressBar : JProgressBar
gRebuildScreenVisible : boolean
gRebuildScreenContainer : JPanel

(b) A small part of a large class.

Figure 5.3: An example Data Class and a Large Class.

The case from Figure 5.3(a) was found in Tomcat. It is a bad type of Data Class for the following reasons: It has many attribute fields, in this case more than ten. It has no methods. All of the attributes are public thus the class cannot bear any responsibility for their values. Other cases that have been found are less severe. Many Data Class instances do have some methods, often for setting and getting the attribute values.

In JHotDraw, Data Classes occur as inner classes of another class. This happens when a class needs some custom data object for local use. For example, the ODGStylesReader$Style class extends HashMap and has a few attributes added to the class. The Style class is an answer for the need of a simple properties container.

False positives

The current implementation of this Code Smell suffers from some false positive indications. First of all, some methods are marked as ‘trivial’ when they should not even exist. The methods <oinit> and <clinit> are internal constructors and helper methods that are inserted by the compiler. They are never entirely empty although they are not implemented by the application programmer. In some cases, they should be filtered. Filtering of these methods is not implemented and most of the times the methods push the candidate class towards a positive indication.

Interfaces are filtered because they are typically simple. They define no attributes and their methods are empty, thus ‘trivial’. In the Tomcat project, some classes define native methods that provide an interface to other programs. The problem is that such classes are not actual interfaces, thus they are not filtered. They do have the properties of interfaces since they define empty methods. These classes are rated as positive cases, which is not correct.

Abstract classes can have all the properties of Data Classes. In some cases, abstract classes are marked as positive cases. When another class extends the abstract class, it can add responsibilities. That extending class is then not a Data Class. Because abstract classes are explicitly meant to have logic added by extending classes, they might be seen as an exception to the Data Class Code Smell. The current implementation treats concrete classes and abstract classes the same way, meaning that it only looks at the given class and it ignores any superclasses or subclasses.

Some classes have few attributes, such as one or two. For example, the ‘InexactComparisonCriteria’ class listed above. This class triggers a positive indication because it does not have any logic, meaning that it is a simple class. Although it only has one attribute, it still matches the criteria for a Data Class. The solution would be to rate classes with few attributes down to a more neutral indication.

In JForum, some classes are annotated with Hibernate[2] annotations, meaning that the class is an interface to the database. In that case, the class does have some responsibilities and deeper meaning in the program although the heuristic identifies this class as a Data Class. It may be hard to identify classes that have such a hidden role in the system. The current implementation does not recognize them.

5.2.2 Large Class

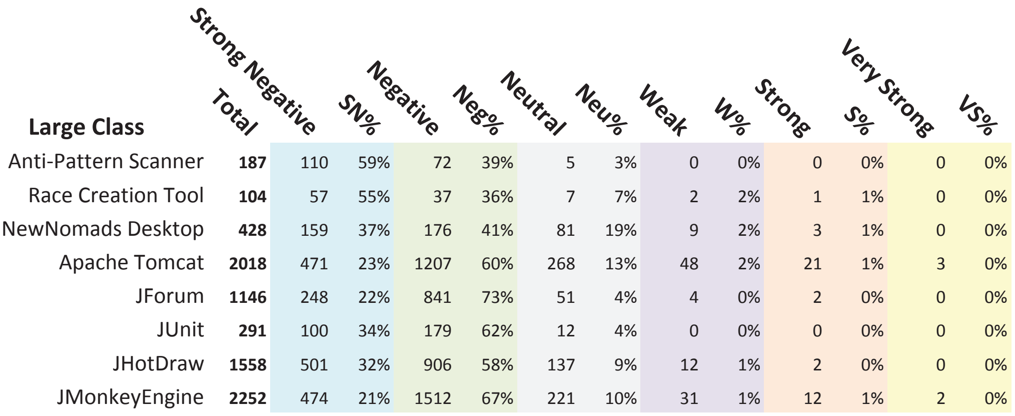
Figure 5.4 shows the scan results for this Code Smell.

Positive cases

Some of the detected cases:

Race Creation Tool The GUI.
NewNomads Desktop Various one-size-fits-all concrete CRM entries, such as a ‘Product’.
Apache Tomcat NioEndpoint, StandardContext.

[2] http://www.hibernate.org/



Large Class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	110	59%	72	39%	5	3%	0	0%	0	0%	0	0%
Race Creation Tool	**104**	57	55%	37	36%	7	7%	2	2%	1	1%	0	0%
NewNomads Desktop	**428**	159	37%	176	41%	81	19%	9	2%	3	1%	0	0%
Apache Tomcat	**2018**	471	23%	1207	60%	268	13%	48	2%	21	1%	3	0%
JForum	**1146**	248	22%	841	73%	51	4%	4	0%	2	0%	0	0%
JUnit	**291**	100	34%	179	62%	12	4%	0	0%	0	0%	0	0%
JHotDraw	**1558**	501	32%	906	58%	137	9%	12	1%	2	0%	0	0%
JMonkeyEngine	**2252**	474	21%	1512	67%	221	10%	31	1%	12	1%	2	0%

Figure 5.4: Scan results for the Large Class Code Smell.

JForum ConfigKeys, User.
JHotDraw DefaultDrawingView, HandleAttributeKeys.
JMonkeyEngine ParticleSystem, WaterRenderPass.

Observations

Large classes occur in many systems, like the Data Class Code Smell. The class shown in Figure 5.3(b) is a small part of the overview of the Graphical User Interface (GUI) component from the RCT. With over 100 attributes, 40 methods and around 10 internal classes, this class is definitely large. Poor GUI class design results in many attributes in one class that are used to manage the many containers, screens and listeners. In this case, the class is made even larger because it's methods contain long scripts that create, check and fill the GUI components. With all this control over those containers, the class certainly also possess the God Class Code Smell and it is probably a Blob too.

Some classes have hundreds of attributes and few methods, some vice versa. In either case, such classes are large. In all of the examined systems, large classes make up only a small part of the system. Large classes often occur where multiple components are bundled together. It appears that the classes often serve as an access point to many subsystems.

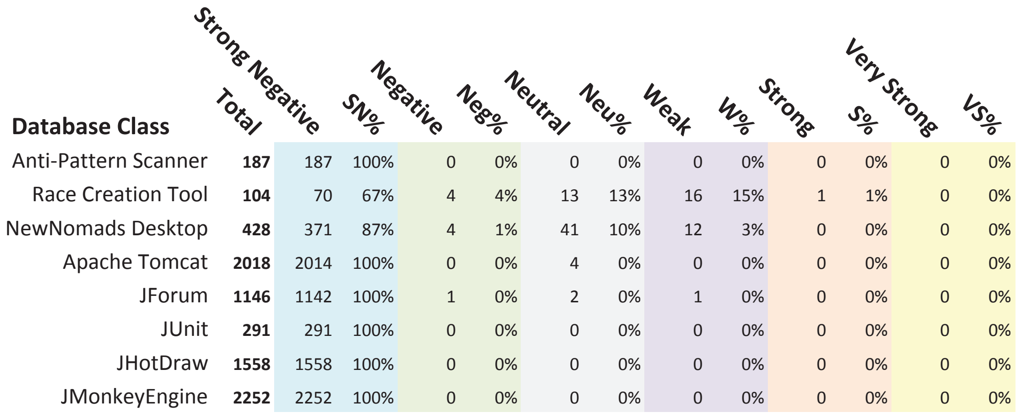
False positives

Some interfaces define many methods. Even though an interface never defines attributes, it might still be marked as a Large Class if it has many methods. This occasionally occurs for interfaces from the Tomcat project. The question is, are they actually large? It may depend on personal taste. On the other hand, even if the interface would not be large, the class that implements the interface typically is. For this reason, it might be useful to scan interfaces too. Positive cases of large interfaces might indicate that other classes are forced to be large. For this reason, large interfaces should not be false positive.

Similar to that, some classes define many attributes. For example, the ‘HandleAttributeKeys’ class defines many attributes that are both static and final, meaning that they are system-wide constants. Classes that are only used for constant definition could be ignored, since they do not suffer the disadvantages of maintenance of large classes.

5.2.3 Database Class

Figure 5.5 shows the scan results for this Anti-Pattern.



Database Class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	187	100%	0	0%	0	0%	0	0%	0	0%	0	0%
Race Creation Tool	**104**	70	67%	4	4%	13	13%	16	15%	1	1%	0	0%
NewNomads Desktop	**428**	371	87%	4	1%	41	10%	12	3%	0	0%	0	0%
Apache Tomcat	**2018**	2014	100%	0	0%	4	0%	0	0%	0	0%	0	0%
JForum	**1146**	1142	100%	1	0%	2	0%	1	0%	0	0%	0	0%
JUnit	**291**	291	100%	0	0%	0	0%	0	0%	0	0%	0	0%
JHotDraw	**1558**	1558	100%	0	0%	0	0%	0	0%	0	0%	0	0%
JMonkeyEngine	**2252**	2252	100%	0	0%	0	0%	0	0%	0	0%	0	0%

Figure 5.5: Scan results for the Database Class Anti-Pattern.

Positive cases

Some of the detected cases:

Race Creation Tool All classes that represent a database table manage their own queries. That is exactly 50% of all classes.

NewNomads Desktop Due to many application module classes that manage their application-specific database in their own way, nearly 40% of all of the classes interact with a database.

JForum JDBCLoader.

Observations

The Database Class Anti-Pattern does not occur often. Figure 5.5 shows that there are few positive cases. Projects that have 100% Strong Negative cases often do not use database interaction at all.

The Database Class Anti-Pattern shown in Figure 5.6 was taken from the NewNomads Desktop. The classes on the bottom of the image belong to the database connection pool and offer generic query functions. A caching layer is added to the application to speed up reading operations. The topmost classes contain some basic logic for checking input and output. They read and write to the database. Other components (not shown) can invoke the topmost layers and set or read values, they can retrieve lists of strings containing information from tables and they might need to handle SQLExceptions. The problem is, none of the layers provide any abstraction. The application logic needs to initialize the connection, set and read scalar values and handle exceptions. The topmost layer is not an abstraction layer, it is just there to help manage the connection. This means that both the cache and the helper classes are Database Classes. The classes on the bottom are the basic connection layer which does not consist of Database Classes. This behavior also explains why around 40% of the program uses classes from the java.sql package: most of the logic is also responsible for the persistence of the data it uses.

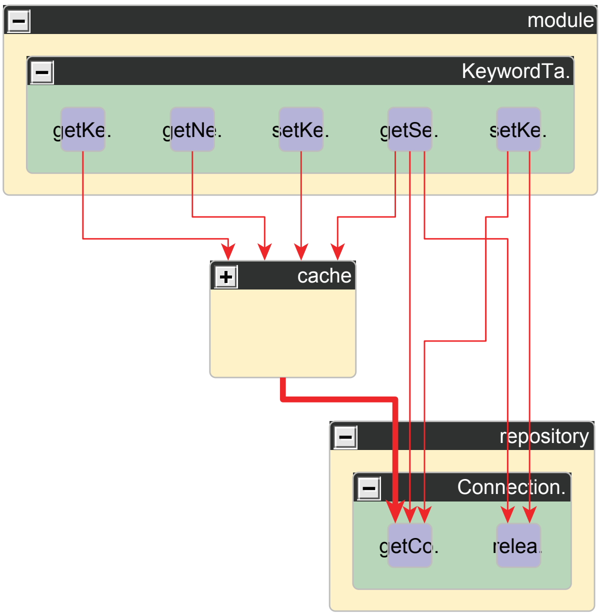
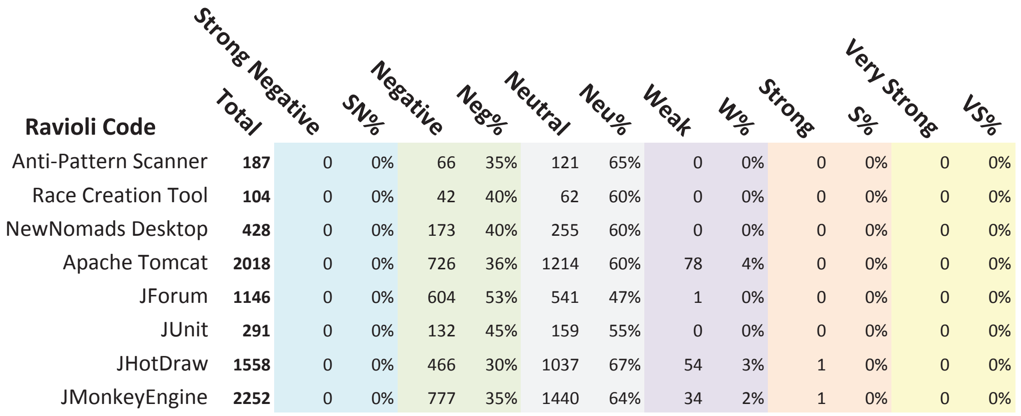


Figure 5.6: Part of a logic module in the NewNomads Desktop.

In the JForum project, all data is being managed by annotated ‘data classes’ that use the Hibernate data persistence framework. This means that the data is properly abstracted. For the test cases, a database dump is used to set up the test environment. The ‘JDBCLoader’ is a helper class for this setup. It has all the properties for a Database Class because it does not use an abstraction to manage the database, instead it just receives input statements and returns the result. While it is a real case of a Database Class Anti-Pattern, it is obviously not malicious in this context.

5.2.4 Ravioli Code

Figure 5.7 shows the scan results for this Anti-Pattern.



Ravioli Code	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	0	0%	66	35%	121	65%	0	0%	0	0%	0	0%
Race Creation Tool	**104**	0	0%	42	40%	62	60%	0	0%	0	0%	0	0%
NewNomads Desktop	**428**	0	0%	173	40%	255	60%	0	0%	0	0%	0	0%
Apache Tomcat	**2018**	0	0%	726	36%	1214	60%	78	4%	0	0%	0	0%
JForum	**1146**	0	0%	604	53%	541	47%	1	0%	0	0%	0	0%
JUnit	**291**	0	0%	132	45%	159	55%	0	0%	0	0%	0	0%
JHotDraw	**1558**	0	0%	466	30%	1037	67%	54	3%	1	0%	0	0%
JMonkeyEngine	**2252**	0	0%	777	35%	1440	64%	34	2%	1	0%	0	0%

Figure 5.7: Scan results for the Ravioli Code Anti-Pattern.

Positive cases

Some of the detected cases:

Apache Tomcat AbstractReplicatedMap, ApplicationHttpRequest.
JHotDraw SelectionTool.
JMonkeyEngine BinaryClassLoader.

Observations

Figure 5.8 shows the detected Anti-Pattern instance from the JHotDraw project. Purple nodes are interfaces, green nodes are concrete implementations or abstract classes. From this figure it can be seen that the SelectionTool class has some concrete dependencies and many abstract or interface dependencies. That class is somewhat large and it has a high number of dependencies. Those dependencies themselves have mostly abstract or interface dependencies. Not all dependencies are included in this figure.

As shown in Figure 5.7, few positive cases have been found. The cases of Ravioli Code that have been found are most often weak indications. The SelectionTool is somewhat stronger although there are no severe cases found in any system. In general, all of the Ravioli Code instances are somewhat large classes with many dependencies. In all cases, their neighbourhood appears to be a cluster of mostly abstract classes or interfaces. Those clusters are much more abstract than other structures found throughout the projects.

5.2.5 The Blob

Figure 5.9 shows the scan results for this Anti-Pattern.

Positive cases

Some of the detected cases:

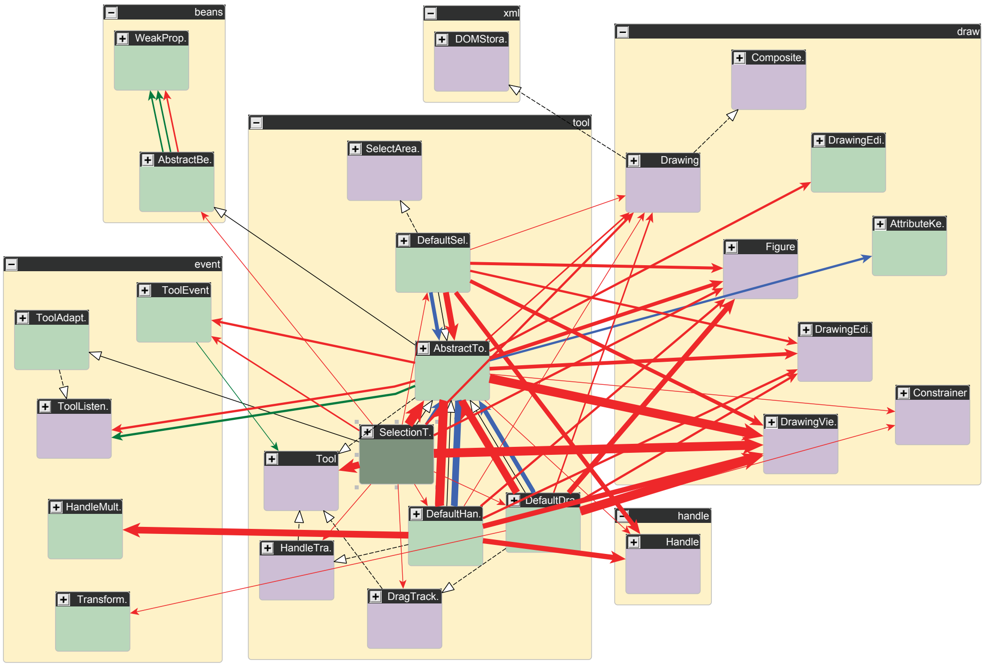
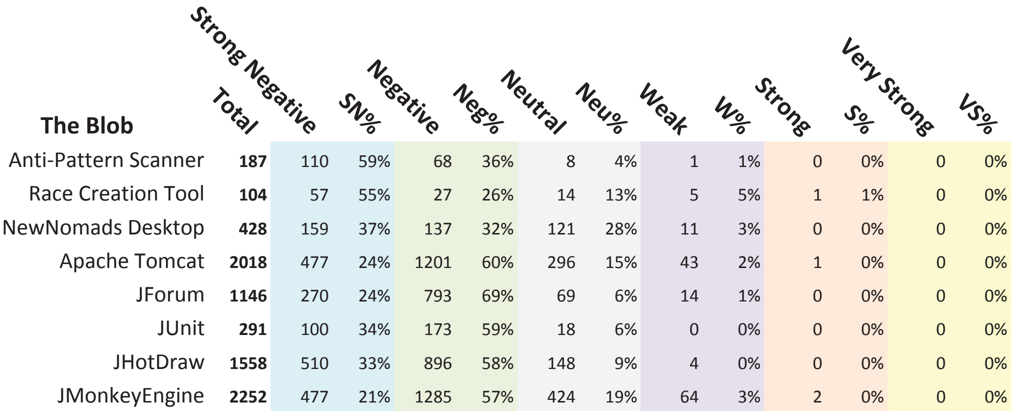


Figure 5.8: The SelectionTool from the JHotDraw project.



The Blob	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	110	59%	68	36%	8	4%	1	1%	0	0%	0	0%
Race Creation Tool	**104**	57	55%	27	26%	14	13%	5	5%	1	1%	0	0%
NewNomads Desktop	**428**	159	37%	137	32%	121	28%	11	3%	0	0%	0	0%
Apache Tomcat	**2018**	477	24%	1201	60%	296	15%	43	2%	1	0%	0	0%
JForum	**1146**	270	24%	793	69%	69	6%	14	1%	0	0%	0	0%
JUnit	**291**	100	34%	173	59%	18	6%	0	0%	0	0%	0	0%
JHotDraw	**1558**	510	33%	896	58%	148	9%	4	0%	0	0%	0	0%
JMonkeyEngine	**2252**	477	21%	1285	57%	424	19%	64	3%	2	0%	0	0%

Figure 5.9: Scan results for The Blob Anti-Pattern.

Race Creation Tool A class with the appropriate name ‘GUI’.
Apache Tomcat PageInfo.
JHotDraw DefaultDrawingView, PaletteToolbarUI (weak indications).
JMonkeyEngine ParticleSystem, WaterRenderPass.

Observations

The Blob often occurs in a lesser form. Figure 5.9 shows that weak cases have been found in most projects. Of all Anti-Patterns that were scanned, The Blob is the most common. Figure 5.10 shows a birdeye view of a Blob. In this case, the JMonkeyEngine 'WaterRenderPass' has 35 different dependencies, some of them having proper responsibilities, others just being simplistic. The WaterRenderPass class is where it all comes together, having many different responsibilities and being large. Due to the reasonable balance in class responsibilities, it is not a severe case of a Blob.

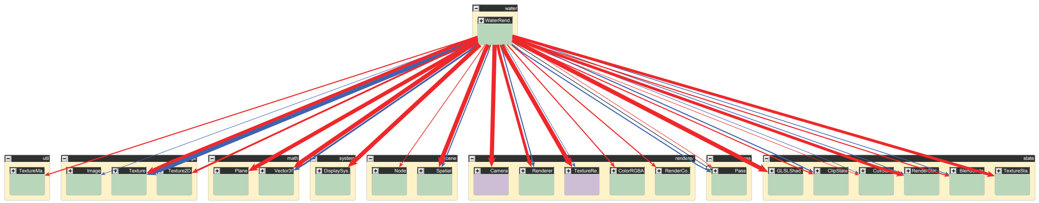


Figure 5.10: An illustrative example of a potential Blob.

In all cases, a Blob also has the God Class Code Smell as determined by the InCode Eclipse Plugin. The classes are always somewhat large if not very large. In the particular case of the RCT GUI, the Blob is the class that manages the entire GUI and it has control over lots of simple 'listener' classes and 'container' classes that contain GUI elements. In systems that are better designed, blobs still occur as classes that depend on many other classes, typically to bundle or manage those classes.

Some of the weaker cases of The Blob are caused by large classes that have many dependencies although the dependencies are not 'simple'. A large cluster of 'good' classes that are bundled by one fat class. Such cases do suffer from Single Responsibility violations or the God Class Code Smell although the related classes are not being used as simple slave classes.

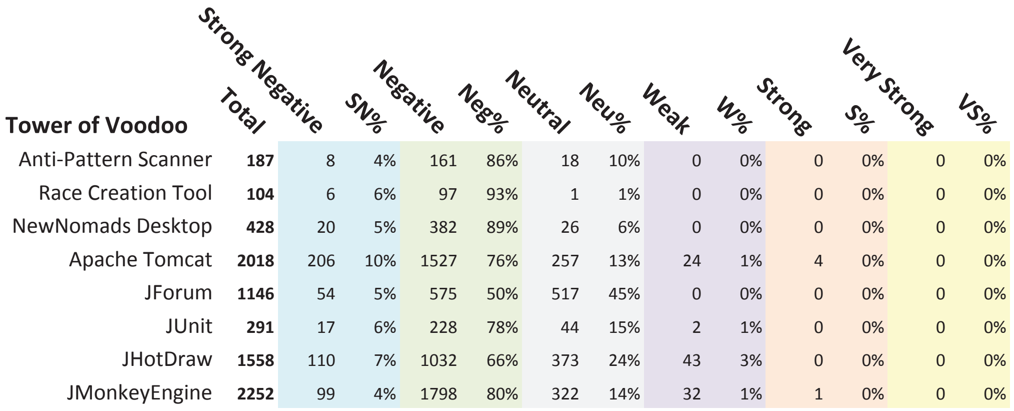
5.2.6 Tower of Voodoo

Figure 5.11 shows the scan results for this Anti-Pattern.

Positive cases

Some of the detected cases:

Apache Tomcat JspMethodNotFoundException.
JUnit AllTests.
JHotDraw ActionsToolbar, DependencyFigure (weak indications).
JMonkeyEngine TestDnD$MyGameState, AbsoluteMouse.



Tower of Voodoo	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	8	4%	161	86%	18	10%	0	0%	0	0%	0	0%
Race Creation Tool	**104**	6	6%	97	93%	1	1%	0	0%	0	0%	0	0%
NewNomads Desktop	**428**	20	5%	382	89%	26	6%	0	0%	0	0%	0	0%
Apache Tomcat	**2018**	206	10%	1527	76%	257	13%	24	1%	4	0%	0	0%
JForum	**1146**	54	5%	575	50%	517	45%	0	0%	0	0%	0	0%
JUnit	**291**	17	6%	228	78%	44	15%	2	1%	0	0%	0	0%
JHotDraw	**1558**	110	7%	1032	66%	373	24%	43	3%	0	0%	0	0%
JMonkeyEngine	**2252**	99	4%	1798	80%	322	14%	32	1%	1	0%	0	0%

Figure 5.11: Scan results for the Tower of Voodoo Anti-Pattern.

Observations

The Tower of Voodoo Anti-Pattern is present in some of the projects. As shown in Figure 5.11, few Strong cases have been found. There are no Very Strong cases.

An example Tower of Voodoo is shown in Figure 5.12. The topmost structure has the most significant levels and is the only structure that counts towards the final rating. This example features a game that is built upon a debug engine, built upon a text based game, based on a generic engine, built upon a state controller.

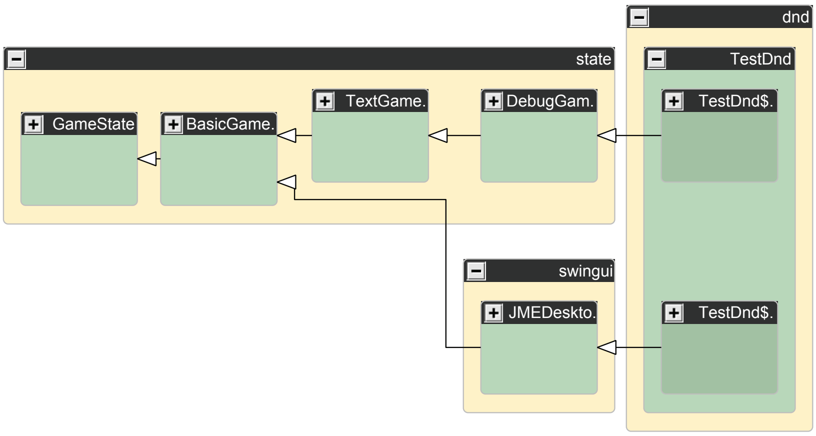


Figure 5.12: A Tower of Voodoo example from the JMonkeyEngine test games.

Some of the cases that were found seem to express some laziness. Empty comments blocks, auto-generated comments, virtually empty methods have been found. Another example is a ‘quick’ experimental or stub extension. One was found with the comment “Not ready yet”. It was last maintained in JDK 1.4. Such cases could also be considered cases of the Lava Flow Anti-Pattern [5].

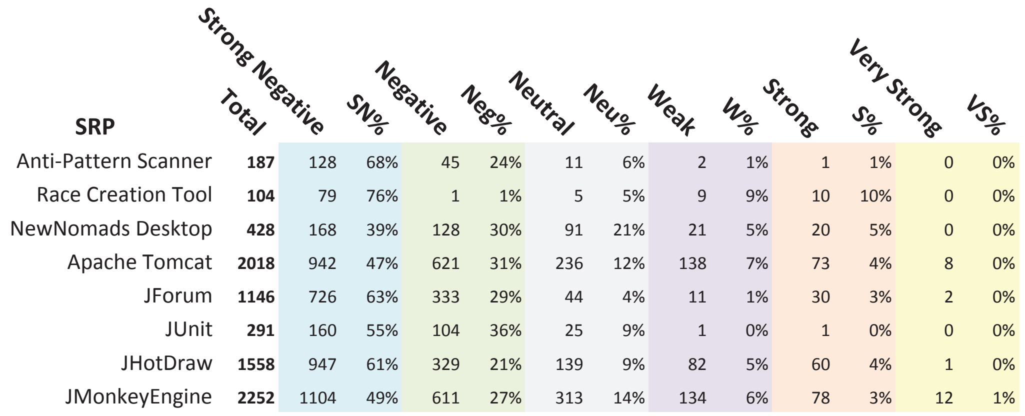
The cases picked up by this heuristic are those where no explicit definitions are supplied for the added levels. Manual examination also revealed cases that were longer than the positive cases but that did have explicit information about their structure. Those cases had interfaces implemented by their intermediate levels, meaning that the interface defines the added value of such an intermediate level.

False positives

Exception inheritance structures are the most common cases of this Anti-Pattern. Long trees of exception types that specify exceptions for more specific cases can be found in many applications, including the Java language itself. These cases are often simple extensions that add more meaning than functionality and they may not be malicious to the application. Still, they are cases of the Tower of Voodoo and in some cases, rethinking their meaning might make them obsolete. Those exception structures are easily picked up by this heuristic and it might depend on personal taste to whether they are or are not real cases of this Anti-Pattern.

5.2.7 Single Responsibility Principle

Figure 5.13 shows the scan results for this Design Principle.



SRP	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	128	68%	45	24%	11	6%	2	1%	1	1%	0	0%
Race Creation Tool	**104**	79	76%	1	1%	5	5%	9	9%	10	10%	0	0%
NewNomads Desktop	**428**	168	39%	128	30%	91	21%	21	5%	20	5%	0	0%
Apache Tomcat	**2018**	942	47%	621	31%	236	12%	138	7%	73	4%	8	0%
JForum	**1146**	726	63%	333	29%	44	4%	11	1%	30	3%	2	0%
JUnit	**291**	160	55%	104	36%	25	9%	1	0%	1	0%	0	0%
JHotDraw	**1558**	947	61%	329	21%	139	9%	82	5%	60	4%	1	0%
JMonkeyEngine	**2252**	1104	49%	611	27%	313	14%	134	6%	78	3%	12	1%

Figure 5.13: Scan results for the Single Responsibility Principle.

Positive cases

Some of the detected cases:

The Anti-Pattern Scanner ScanCache.

Race Creation Tool Buildings, units, upgrades, …
NewNomads Desktop Core framework management class, the cache of one of the application modules, many of the concrete CRM entry classes.
Apache Tomcat AbstractReplicatedMap, Node, StandardWrapper.
JForum Topic, User.
JUnit ResultPrinter.
JHotDraw DefaultDrawingView.
JMonkeyEngine AbstractCamera, AudioTrack.

Observations

Violations to this principle are common. As shown in Figure 5.13, both Strong and Very Strong cases occur frequently. Cases have been found in all systems and often in large amounts. Figure 5.14 shows the SRP violations in the SearchParams class from the JForum project. To begin with, some of the attributes of the class (round nodes) are independent from one another. This causes them to be seen as multiple different related sets of attributes. Furthermore, the methods are being invoked in different sets from different classes. For example, the ‘getQuery’, ‘getUser’ and ‘setMaxResults’ methods are one set, the ‘getMaxResults’ and ‘getStart’ methods are in another set.

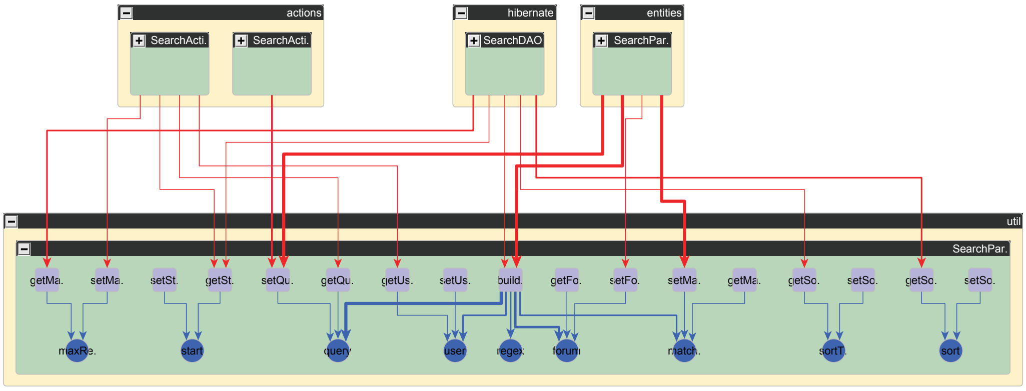


Figure 5.14: A class that violates the SRP principle.

Classes can have control methods, used to manipulate the state or the data of the objects. Some of these methods are exposed, some are not. Control methods often appear to be reserved for specific clients or simply not properly protected. In general, all methods that are not properly protected become exposed responsibilities. In many cases, control methods are one set of responsibilities and the class offers additional methods for other purposes.

Classes that control a system state typically have methods that set their flags, switch states on and off etc. They also have a set of methods that do the actual controlling such as determining whether something is allowed, triggering or even executing logic based on the state.

Manager-style classes typically have sets of getter and setter methods for core components. They might also offer persistent storage entry points, application logic entry points, user-interface hooks etc.

Objects that contain the logic sometimes also contain the data for executing the logic or contain the result data. Both the input and output data can be different sets, sometimes there are even multiple sets for either. The latter typically occurs when the data is modeled using different primitives, each with their own methods for getting and setting. If data is modeled using custom objects to contain the data, fewer methods and sets are used.

Helper methods and logic execution methods often reside in the same class. For example, a parser that has control methods for the parse flow also can have utility methods for String analysis.

Some classes offer overloaded versions of methods. Each version has nearly the same functionality but using different parameters. These methods are often ‘dangling’ because if one is used, the other is not needed. Also, some objects appear to be an envelope that is used for passing messages. The envelope can be routed through a chain of classes and for each class, it offers ‘dangling’ methods that are used by the class to get routing information.

In many cases, SRP violations appear to have a strong overlap with the God Class Code Smell. Classes that do or delegate a lot of work often have many methods. This often translates to many responsibilities.

Classes might implement interfaces or extend abstract classes that cause them to have multiple responsibilities. For example, the DefaultDrawingView class from the JHotDraw project seems to be concerned with many aspects of drawing the canvas: size, visibility, settings, drawing execution, caching, transformation and providing hooks for change listeners. Most of these responsibilities seem to be forced upon this component because it extends an abstract class and implements multiple interfaces.

False positives

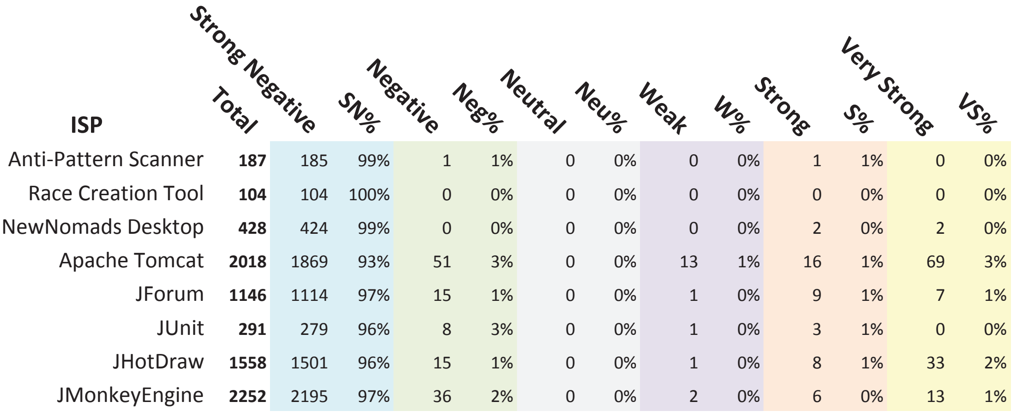
Dynamic method invocation occurs. Methods are mapped dynamically and the usage of the methods is not visible in the source. Manual inspection has revealed that this can cause problems. Methods that are not invoked can become ‘dangling’ methods that can cause a class to get a higher rating towards violation of this principle. Also, the scanner might miss that two sets of methods actually belong to the same set because the invocation from a bridging method is never seen.

Classes that are not used, used only once, or few times have a higher probability for being rated positive. If there are more methods that invoke methods from a class, that class will probably have fewer sets containing more methods. Some classes which are used only few times have a positive indication. This indication is correct based on the information that is available but the confidence in such a rating is lower.

Abstract classes and classes that extend abstract classes might suffer from false positive indications. When some of the invocations occur on the methods of the parent class and some on those of the child class, the invocations are not processed in the same scanner pass. This might result in more sets of methods. The solution would be to see all instances of overridden and base methods as the same method, meaning that the actual usage is determined by an entire inheritance structure.

5.2.8 Interface Segregation Principle

Figure 5.15 shows the scan results for this Design Principle.



ISP	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	185	99%	1	1%	0	0%	0	0%	1	1%	0	0%
Race Creation Tool	**104**	104	100%	0	0%	0	0%	0	0%	0	0%	0	0%
NewNomads Desktop	**428**	424	99%	0	0%	0	0%	0	0%	2	0%	2	0%
Apache Tomcat	**2018**	1869	93%	51	3%	0	0%	13	1%	16	1%	69	3%
JForum	**1146**	1114	97%	15	1%	0	0%	1	0%	9	1%	7	1%
JUnit	**291**	279	96%	8	3%	0	0%	1	0%	3	1%	0	0%
JHotDraw	**1558**	1501	96%	15	1%	0	0%	1	0%	8	1%	33	2%
JMonkeyEngine	**2252**	2195	97%	36	2%	0	0%	2	0%	6	0%	13	1%

Figure 5.15: Scan results for the Interface Segregation Principle.

Positive cases

Some of the detected cases:

The Anti-Pattern Scanner NamedScannerInterface.
NewNomads Desktop Application module - CRM: The generic CRM entry interface.
Apache Tomcat CatalinaCluster, Context, HttpServletRequest.
JForum Most of the interfaces that extend Repository<T>.
JUnit JUnitSystem, Test.
JHotDraw Application, CompositeFigure.
JMonkeyEngine InputCapsule, Savable.

Observations

The percentage of positive indications for the Interface Segregation Principle among interfaces is often high, sometimes almost 50%. Figure 5.15 shows the amount of indications. These are the indications for *all classes*. Classes that are not an interface are marked as Strong Negative. For example, in the Tomcat project, of the 2018 classes examined, around 7% is an interface. Around 3% of all those classes is an interface that is also marked as Very Strong.

Figure 5.16 shows an example of an ISP violation.

One of the classes only uses the ‘getName’ method, the other also uses ‘getExaplanation’ method. Larger versions of this violation are often found. In many cases, the methods have no usage overlap at all.

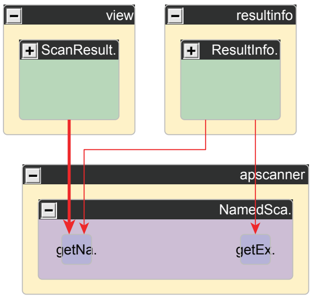


Figure 5.16: A small violation of the ISP in the Anti-Pattern Scanner.

Interfaces that define a library access point such as a List typically have methods like get(), set(), getAll(), add() or remove(). Many of the methods are situational. Most of the time only two of the methods are used and the others do not match the situation. This is even more true for different types of methods, such as helper condition checking and control methods. For example, the isEmpty() and add() methods are typically not used in the same situation, or not by the same client.

Some interfaces are used to define a universal data object or another universal component type. Such interfaces might offer methods for lifecycle management, such as init(), setID() and tearDown(). Interfaces might offer conversion methods such as write(), toXML(), toArray() etc. All of these roles are often needed in combination. The good solution would be to define an interface for every role or even part of a role. In practice however, the combinations of roles are often defined in the interface itself, meaning that the interface also has multiple responsibilities. Instead, the object that implements the interface should be able to pick those role-defining interfaces in the combination that it needs.

An example of these roles is the Filter interface from Tomcat. This interface defines methods for lifecycle support and it defines the actual ‘filter’ method. The lifecycle is then used by one of the Tomcat components to manage the starting and stopping of the filters. The actual filtering is invoked from a completely different component.

In general, interfaces that are small, such as three or two methods, rarely suffer from ISP violations. They are usually compact and to-the-point, defining only one role.

An interface that extends an interface should only expand the role already defined by the superinterface. This could keep the superinterface small and the extending interface clear about what the extension adds. In practice however, some interfaces tend to add roles, causing their ISP violation rating to increase. An example is the AvatarRepository from the JForum project. The base Repository<Avatar> interface defines methods for adding and removing entries. Interfaces that extend Repository<Avatar> typically add a getAll() option. However, the AvatarRepository (extends Repository<Avatar>) also adds methods for managing a gallery.

False positives

Rare cases of empty interfaces have been reported. Even if that interface extends another interface, the indication is biased because it claims that ‘none of the methods are invoked’. This causes a positive indication although the interface is not violating the principle.

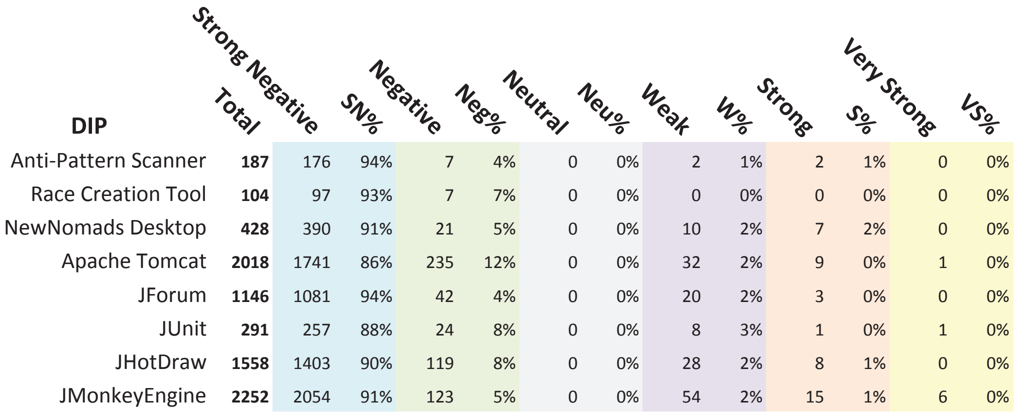
Deprecated interfaces or interfaces that have a few deprecated methods might yield a biased indication. Such interfaces tell implementing classes that the methods should not be used. The problem is that when these methods are indeed not used, the interface forces unnecessary method implementation upon clients. In the context of keeping deprecated methods, it is questionable whether this is good or bad practice.

In some framework projects, such as JHotDraw, interfaces are used to provide an API that allows other applications to access the framework. Such interfaces are not yet used when examining the framework itself, causing their rating to weakly increase. For this reason, interfaces that are only meant for other programs could be ignored.

Interfaces that are seeing little use at all cannot provide proper indications. This problem is similar to that of the SRP.

5.2.9 Dependency Inversion Principle

Figure 5.17 shows the scan results for this Design Principle.



DIP	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**187**	176	94%	7	4%	0	0%	2	1%	2	1%	0	0%
Race Creation Tool	**104**	97	93%	7	7%	0	0%	0	0%	0	0%	0	0%
NewNomads Desktop	**428**	390	91%	21	5%	0	0%	10	2%	7	2%	0	0%
Apache Tomcat	**2018**	1741	86%	235	12%	0	0%	32	2%	9	0%	1	0%
JForum	**1146**	1081	94%	42	4%	0	0%	20	2%	3	0%	0	0%
JUnit	**291**	257	88%	24	8%	0	0%	8	3%	1	0%	1	0%
JHotDraw	**1558**	1403	90%	119	8%	0	0%	28	2%	8	1%	0	0%
JMonkeyEngine	**2252**	2054	91%	123	5%	0	0%	54	2%	15	1%	6	0%

Figure 5.17: Scan results for the Dependency Inversion Principle.

Positive cases

Some of the detected cases:

The Anti-Pattern Scanner AbstractComplexScanner, AbstractSimpleScanner.

NewNomads Desktop The abstract CRM entry, which depends on various concrete framework classes.

Apache Tomcat RealmBase.

JForum ForumRepository, TagRepository.

JUnit ParentRunner, Request.

JHotDraw AbstractFigure, AbstractRotateHandle.

JMonkeyEngine BaseSimpleGame, ParticleSystem.

Observations

Violations of this principle have been found in many of the studied systems. Figure 5.18 shows a detected instance from the JUnit project. The Request class is an abstract class and it has concrete dependencies inside the package. It also has concrete dependencies outside of it's package, such as FilterRequest and AllDefaultPossibilitiesBuilder.

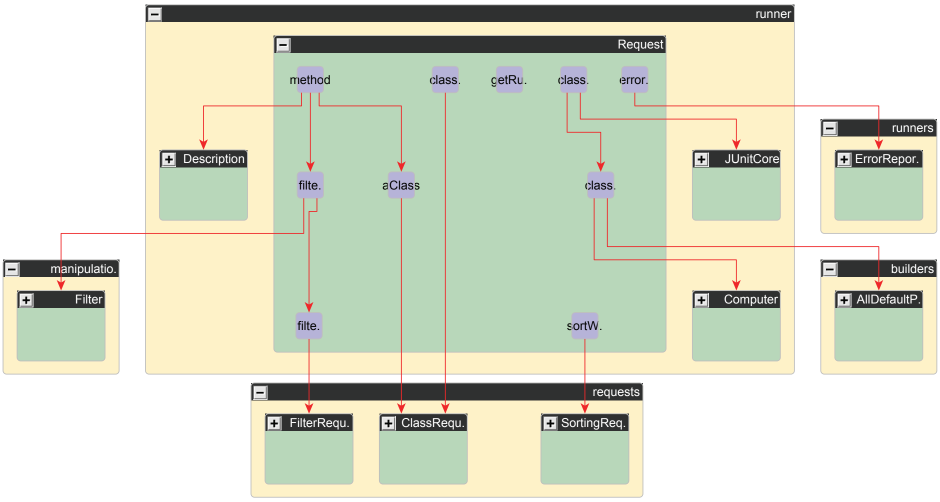


Figure 5.18: The Request class from the JUnit project.

Typical concrete dependencies for both interfaces and abstract classes are framework API classes, utility classes such as logging, data structures and exception types. All kinds of classes have been seen as concrete dependencies although utility classes and data structures appear to be the most common. In many cases this could still be considered a violation of this principle. Many utility classes and data structures do have interfaces that define them. Components often can and should depend on those interfaces instead of the concrete implementations.

In JForum, many of the data classes are mentioned in interfaces or abstract classes. Those data classes are the data containers that also provide persistent storage for the data types. In JMonkeyEngine, many of the abstract examples use concrete classes that provide functionality.

Sometimes two different packages are tightly coupled. They do not always use abstractions for this. For example, the JMonkeyEngine particle system uses many different types of particles (which are concrete). The particles and the engine API are not in the same package thus this relation triggers a positive rating.

It appears to be more common for abstract classes to use concrete dependencies than for interfaces. This is probably because abstract classes already provide some concrete functionality and they might need concrete classes to do so. Interfaces on the other hand need only to orchestrate definitions on a non-concrete level.

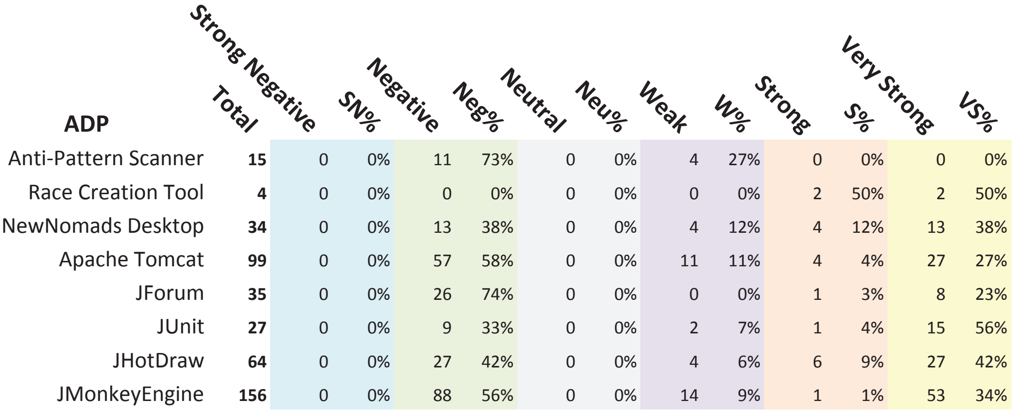
When an abstract class extends another abstraction, and that superclass violates this principle, then the extending class is likely to also violate this principle. For example, The ParentRunner in the JUnit project extends Runner. The latter has one DIP violation. The first one has that same DIP violation, as well as some other violations of this principle.

False positives

Final classes occasionally pop up as concrete dependencies for abstract classes and interfaces. Final classes could be considered less of a threat for this principle, even though they are concrete. Because they are final, they can be expected to be more stable. Keeping in mind that final classes are less severe than other concrete classes, some of the ratings could be lowered slightly.

5.2.10 Acyclic Dependencies Principle

Figure 5.19 shows the scan results for this Design Principle.



ADP	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
Anti-Pattern Scanner	**15**	0	0%	11	73%	0	0%	4	27%	0	0%	0	0%
Race Creation Tool	**4**	0	0%	0	0%	0	0%	0	0%	2	50%	2	50%
NewNomads Desktop	**34**	0	0%	13	38%	0	0%	4	12%	4	12%	13	38%
Apache Tomcat	**99**	0	0%	57	58%	0	0%	11	11%	4	4%	27	27%
JForum	**35**	0	0%	26	74%	0	0%	0	0%	1	3%	8	23%
JUnit	**27**	0	0%	9	33%	0	0%	2	7%	1	4%	15	56%
JHotDraw	**64**	0	0%	27	42%	0	0%	4	6%	6	9%	27	42%
JMonkeyEngine	**156**	0	0%	88	56%	0	0%	14	9%	1	1%	53	34%

Figure 5.19: Scan results for the Acyclic Dependencies Principle.

Positive cases

Some of the detected cases:

The Anti-Pattern Scanner Package ‘apscanner.rating’.
Race Creation Tool Core and GUI.
NewNomads Desktop Many of the application modules have cyclic application dependencies.
Apache Tomcat Packages ‘org.apache.catalina’, ‘org.apache.catalina.startup’, ‘org.apache.catalina.util’.
JForum Packages ‘net.jforum.core’, ‘net.jforum.util’.
JUnit Packages ‘org.junit’, ‘org.junit.runner’.
JHotDraw Packages ‘org.jhotdraw.app’, ‘org.jhotdraw.draw.action’.

JMonkeyEngine Package ‘com.jme.util.export’.

Observations

Cyclic package dependencies occur in many systems and the cycles typically have lengths from two packages to ten, occasionally even much longer. Figure 5.20 shows a small cycle. The packages in this example are inside the ‘org.apache.jk’ package.

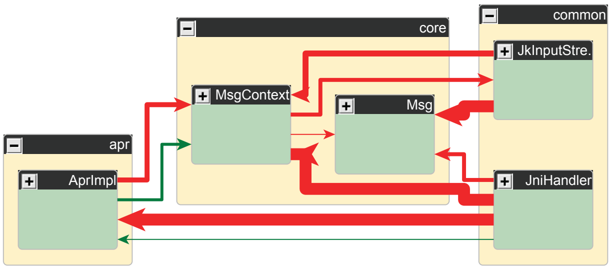


Figure 5.20: An example cyclic relations between packages.

Package cycles often occur in utility packages. Such packages contain classes that provide some generic functionality that can be used by the application logic but that does not seem to be categorized or placed in the package that uses it. The problem is that such utility classes tend to query the framework or upper layers for data as well as invoking application logic to process it. Utility classes might be invoked from all levels. Since the utility classes are not properly categorized, they might serve multiple purposes and all be placed in the same package. With outgoing and incoming dependencies from all levels, cycles are bound to occur.

Another type of cycle that is seen in Tomcat is that the application runs a script at a high level. This script starts up low level components, which in their turn depend on the high level framework.

Cycles often occur within components (for example, package domains), they occur less often across components. For example, in the Tomcat project, domain1.app1.*.* has cyclic dependencies, domain2.app2.*.* has cyclic dependencies, but there are no cyclic dependencies between those. Even though they do depend on each other.

5.3 Statistics

In addition to presenting an overview of positive cases for every pattern, the Anti-Pattern Scanner writes a file containing more detailed statistics for every project. The statistics do not contain reasoning information and they do not name the positive or negative candidates, instead the file contains hard numbers about number of entities, number of scan results and an soft indication of conditional probabilities of patterns relating to each other. An overview of these statistics are shown in this section. For the full data set obtained from every project, see Appendix B.

5.3.1 Accuracy

In general, the accuracy of the implemented heuristics is good. Positive indications always have a reason for being positive, such as an exceeded metric threshold or the presence of a specific structure. Most of the false positive cases can be backtracked to either of the following two reasons. The first reason is that the reasoning is incorrect: The concept behind the heuristic is incorrect for a specific case because according to personal opinion, the specified case is not a design problem. For example, according to personal opinion, an Exception inheritance structure can never be a Tower of Voodoo because it is a personal preference to specify the exception type as detailed as possible. The second reason is that the abstract model is unable to capture a property that would indicate an exception: Information such as annotations, comments etcetera are not included in the model. For example, classes can have Hibernate annotations. These annotations are not included in the model. The annotations would be an indication that a class has a data persistence responsibility. Due to the lack of this information, the class is marked as a Data Class.

For those projects where false positive cases have been found, typically one or two of the examined cases were false positive. A subset of all cases has been examined and this results in an estimation of accuracy. In general, the accuracy of all problem types is between 50% to 100%. The numbers vary per project because not every project uses features that cannot be captured in the model.

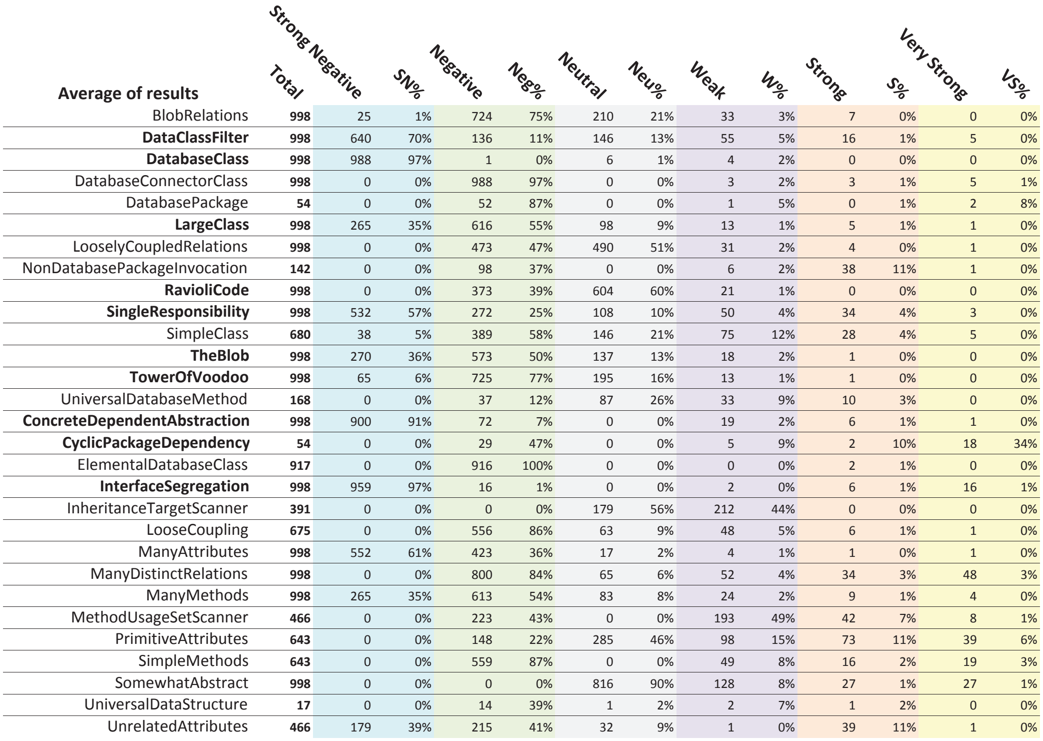
5.3.2 Results by fragment scanner

Figure 5.21 shows the amount of indications for every scanner component for every indication strength category. This table contains the sum of the result data, divided by eight (the number of projects) and rounded for convenience. It may be impossible to draw any solid conclusions from this table because averaging the results of all tables may not be scientifically sound. Therefore, the table should only be used as an indication about the amounts of work for all scanners. Top level pattern scanners have been printed in boldface.

5.3.3 Conditional probabilities

Conditional probabilities have been calculated for each of the projects. The conditional probability $p(Y|X)$ is the probability that an entity receives either a Strong or Very Strong positive rating from scanner Y (vertical axis), given the fact that scanner X (horizontal axis) has given it a Strong or Very Strong positive rating. In other words, when looking at the columns of the table, every number in that column is the probability that when there is a positive indication X (top name), there will also be a positive indication Y (left name).

Figure 5.22 shows the conditional probabilities for all top-level scanners for the JMonkeyEngine project. The amount of positive cases of every scanner X is shown behind the name of the scanner. The more indications, the more reliable this statistic is. From this table it can be seen that for the 14 Large Class Code Smell instances that were found, 7 (50%)



Average of results	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	**998**	25	1%	724	75%	210	21%	33	3%	7	0%	0	0%
DataClassFilter	**998**	640	70%	136	11%	146	13%	55	5%	16	1%	5	0%
DatabaseClass	**998**	988	97%	1	0%	6	1%	4	2%	0	0%	0	0%
DatabaseConnectorClass	**998**	0	0%	988	97%	0	0%	3	2%	3	1%	5	1%
DatabasePackage	**54**	0	0%	52	87%	0	0%	1	5%	0	1%	2	8%
LargeClass	**998**	265	35%	616	55%	98	9%	13	1%	5	1%	1	0%
LooselyCoupledRelations	**998**	0	0%	473	47%	490	51%	31	2%	4	0%	1	0%
NonDatabasePackageInvocation	**142**	0	0%	98	37%	0	0%	6	2%	38	11%	1	0%
RavioliCode	**998**	0	0%	373	39%	604	60%	21	1%	0	0%	0	0%
SingleResponsibility	**998**	532	57%	272	25%	108	10%	50	4%	34	4%	3	0%
SimpleClass	**680**	38	5%	389	58%	146	21%	75	12%	28	4%	5	0%
TheBlob	**998**	270	36%	573	50%	137	13%	18	2%	1	0%	0	0%
TowerOfVoodoo	**998**	65	6%	725	77%	195	16%	13	1%	1	0%	0	0%
UniversalDatabaseMethod	**168**	0	0%	37	12%	87	26%	33	9%	10	3%	0	0%
ConcreteDependentAbstraction	**998**	900	91%	72	7%	0	0%	19	2%	6	1%	1	0%
CyclicPackageDependency	**54**	0	0%	29	47%	0	0%	5	9%	2	10%	18	34%
ElementalDatabaseClass	**917**	0	0%	916	100%	0	0%	0	0%	2	1%	0	0%
InterfaceSegregation	**998**	959	97%	16	1%	0	0%	2	0%	6	1%	16	1%
InheritanceTargetScanner	**391**	0	0%	0	0%	179	56%	212	44%	0	0%	0	0%
LooseCoupling	**675**	0	0%	556	86%	63	9%	48	5%	6	1%	1	0%
ManyAttributes	**998**	552	61%	423	36%	17	2%	4	1%	1	0%	1	0%
ManyDistinctRelations	**998**	0	0%	800	84%	65	6%	52	4%	34	3%	48	3%
ManyMethods	**998**	265	35%	613	54%	83	8%	24	2%	9	1%	4	0%
MethodUsageSetScanner	**466**	0	0%	223	43%	0	0%	193	49%	42	7%	8	1%
PrimitiveAttributes	**643**	0	0%	148	22%	285	46%	98	15%	73	11%	39	6%
SimpleMethods	**643**	0	0%	559	87%	0	0%	49	8%	16	2%	19	3%
SomewhatAbstract	**998**	0	0%	0	0%	816	90%	128	8%	27	1%	27	1%
UniversalDataStructure	**17**	0	0%	14	39%	1	2%	2	7%	1	2%	0	0%
UnrelatedAttributes	**466**	179	39%	215	41%	32	9%	1	0%	39	11%	1	0%

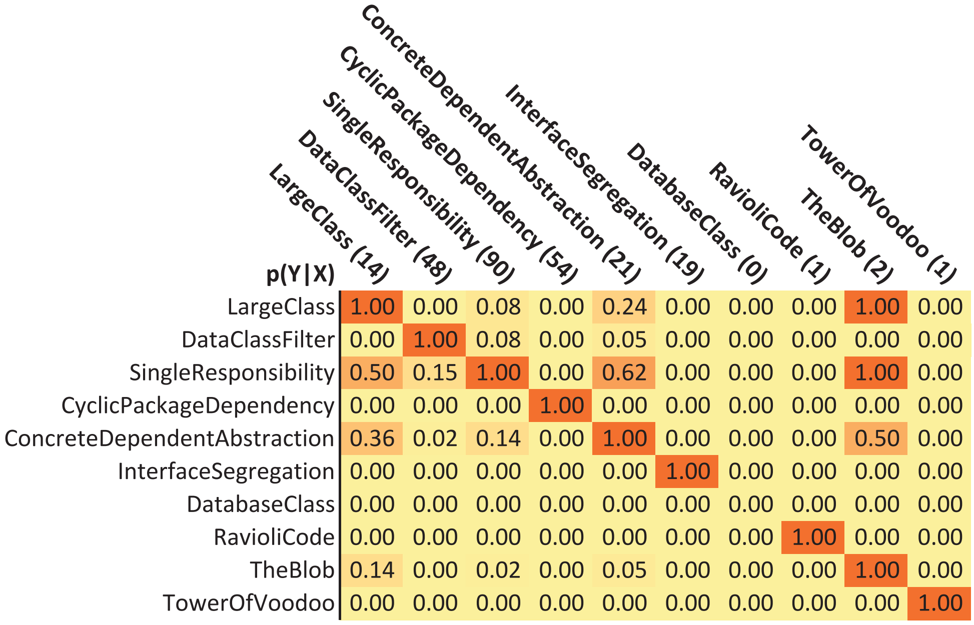
Figure 5.21: Average scan results for the eight scanned projects.

also had a positive indication for Single Responsibility Principle violations. Some indications are interesting and will be evaluated in the next chapter. Other indications are trivial and will be ignored. An example of a trivial indication is that instances of The Blob also have the Large Class Code Smell. This is trivial because The Blob is based on the Large Class Code Smell, meaning that a positive indication of the smell contributes to a positive indication of The Blob Anti-Pattern.

Tables for all projects can be found in Appendix B, as well as conditional probability tables for all individual fragment scanners.

5.3.4 Scanner benchmark

Runtime of the Anti-Pattern Scanner depends greatly on the amount of entities in the abstract model. Runtime is not linear with project size. In all studied cases, runtime is determined by the amount of classes, packages and methods in the projects. To a lesser extend, runtime is determined by cohesion strength of classes. Depending on the amounts of dependencies between classes, some of the scanners need a non-linear amount of time compared to just the overall amount of entities in a model. Cyclic relations can also cause a strongly increased amount of runtime.



p(Y|X)	LargeClass (14)	DataClassFilter (48)	SingleResponsibility (90)	CyclicPackageDependency (54)	ConcreteDependentAbstraction (21)	InterfaceSegregation (19)	DatabaseClass (0)	RavioliCode (1)	TheBlob (2)	TowerOfVoodoo (1)
LargeClass	1.00	0.00	0.08	0.00	0.24	0.00	0.00	0.00	1.00	0.00
DataClassFilter	0.00	1.00	0.08	0.00	0.05	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.50	0.15	1.00	0.00	0.62	0.00	0.00	0.00	1.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.36	0.02	0.14	0.00	1.00	0.00	0.00	0.00	0.50	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
TheBlob	0.14	0.00	0.02	0.00	0.05	0.00	0.00	0.00	1.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

Figure 5.22: Conditional probabilities of the JMonkeyEngine project.

Figure 5.23 shows how much time in milliseconds is needed to scan the JHotDraw project. The computer on which the benchmark is performed is a computer with a Xeon 8-core processor, running at 2.33 GHz. Memory requirements are around 1 gigabyte of RAM with approximately 400 megabytes of memory being used by the fragment scanners, 500 megabytes being used by the Famix parser and around 100 megabytes by the Eclipse IDE. The benchmark shown in Figure 5.23 is performed using only a single processing core.

Depending on package layout and the amount of cycles, the Acyclic Dependencies Principle can have an exploding search space, causing the corresponding scanners to soak up many more gigabytes of memory and much more processing time. To prevent the search space from becoming larger than feasible, the search depth is capped. Limiting the search space could result in missed cases. In practice however, this is no problem because most cycles are found at lower search depths.

Using multiple processing cores can reduce the amount of time needed for scanning significantly. Figure 5.24 shows how using eight cores reduces the time scanning to almost one-third. Parse time is not included because this task cannot be parallelized.

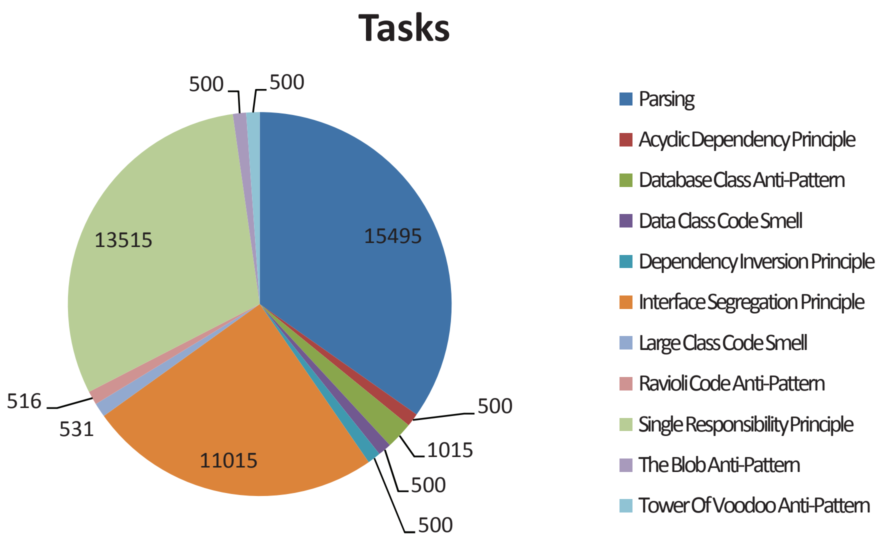


Figure 5.23: Sample runtime requirements (milliseconds) for the JHotDraw project.

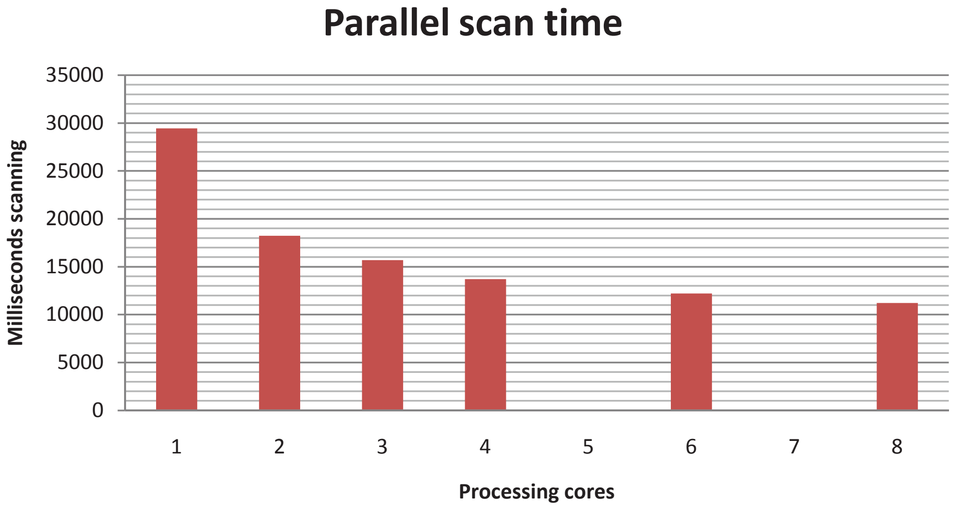


Figure 5.24: Average scan time for the JHotDraw project (parsing excluded).

Chapter 6

Discussion of results

Based on the results of the previous chapter, the questions presented in Chapter 1 are answered in this chapter. General indications are evaluated as well as specific indications for the various pattern types. Based on the results from the study, an indication of the severeness of the problem types is given.

6.1 Evaluation of results

6.1.1 Evaluation of detected occurrences

The positive or negative match of a design problem is partially based on personal opinion. The definitions of the design problems cannot strictly determine which occurrences are positive or negative and conclusions drawn about whether an occurrence is malicious is based on intuition. Inspection of a program's source code and knowledge about the history of a project does not yield hard facts about whether a detected problem is a problem. To get hard facts about the effects of design problems, software bug databases or program maintenance histories should also be inspected. Gathering such information might require MSR. See Chapter 2 for related work where such strategies have been applied. MSR is outside the scope of this study. For this reason, the results presented in this section are based on personal experience and on what information is available in a program's source. These results are not hard facts and cannot be used as proof to claim that a specific design problem is malicious.

Severeness of indications

In general, the Anti-Pattern Scanner can give a good indication of which entities do not possess a given pattern. The negative indications are conservative, meaning that a negative indication is only given to entities that do not possess any of the scanned properties or only a minor portion of them. If there is any doubt about an entity possessing the scanned property, the indication shifts more towards positive than negative. The neutral and weak positive indications often occur when an entity possesses only slightly the sought properties. These indications are numerically close to each other, with neutral leaning more towards negative and weak more towards positive. The neutral and weak positive indications should be seen as a negative indication with the exception that *something* is not good about the entity. The entity shows some hints of Anti-Patterns or principle violations thus it could be worth investigating. In general, any indication of weak or lower does not signal any significant problems and it will probably not be a reason to start refactoring.

The strong positive indication indicates that something more significant is wrong and that it should be investigated. Those cases match the criteria for a pattern in most aspects of that pattern. In practice however, many strong cases have a mix of very strong positive and weaker indications, often even negative. When there is a mix of negative and very strong properties, the numerical indication might yield a strong case but in reality, the examined entity does not match a given pattern. For example, classes that have a very strong violation of the Single Responsibility Principle and that do not have many relations might get a strong indication for The Blob. Closer examination should reveal that the entity does indeed have some strong violations but eventually should not be considered a 'bad case' of the detected pattern. In general, strong cases are never problematic. They will have some flaws, possibly a mix of flaws and good parts. Still, it might be rewarding to refactor those cases because their flaws could become worse. In many cases, small changes could significantly lower the rating towards negative as the flaws are either not widespread or not severe.

Very strong positive indications are almost always significant. They are usually a mix of weak, strong and very strong indications, meaning that the scanned properties are clearly seen in the scanned entity. Very strong positive indications are often the result of crossing the thresholds by a significant amount as the thresholds for very strong positive are often multitudes of what would be considered 'reasonable' for a software entity. Each of these very strong positive indications should be investigated. Some of them might still be lesser cases, such as cases where thresholds have been exceeded with a minimal amount. Others are cases that cast all principles aside and become problematic instances beyond reason. Some cases are just not problematic. Examining them more closely shows that the code smells, the layout is difficult to comprehend yet the implementation will work properly. Those cases benefit from refactoring since they do possess the sought problem. The more structural problems might not be economic to refactor. Removing some theoretical problems from a perfectly working entity can be too expensive and even unnecessary. More severe cases might also be found. When a very strong case is found, it should be refactored.

Severeness of positive cases

The strengths and severeness of detected occurrences varies between the patterns. In general, the Code Smells are local problems inside entities. Positive cases are not a severe threat and refactoring is probably easier than that of structural patterns. For Anti-Patterns, it is more difficult to estimate whether they are malicious. Positive indications require manual evaluation of the project source. Some of the projects examined in this study are familiar, others are not. It is often easier to tell from familiar projects whether a design problem is malicious. For unfamiliar projects, this is much harder to tell.

Anti-Patterns are rather 'soft' in how bad they are and whether a case is positive or negative. A programmer who has developed a program for a long time will have less difficulties with an implementation than someone who is new to the program. When someone is used to working with an Anti-Pattern, he or she will probably not see it as an Anti-Pattern because that person knows how to work with it and how to avoid problems. Furthermore, the person has seen it grow over time or even from the start. The design might have been good in the first place but many changes have slowly caused Anti-Patterns to develop. This might yield a perfectly good explanation for why things are the way they are, even if they have grown to be Anti-Patterns.

The actual positive or negative occurrence of Anti-Patterns is also a matter of taste. Judgement will vary from person to person. Code Smells and Design Principle violations will probably be less dependent on taste because they are more crisp. Either a metric threshold is exceeded or it is not. Either the implementation violates the rule of a Design Principle or it adheres to it.

Unlike Code Smells, Design Principle violations are structural problems. Cyclic dependencies, concrete relations, class responsibilities and most of the other principles tell how the structure should be. Violating principles is probably as easy as implementing smelly code and as difficult to refactor as an Anti-Pattern. The occurrence of a principle violation can be indicated by crisp rules but the badness of a violations is -like an Anti-Pattern- very much subject to personal taste.

In most cases, it is not possible to say whether a piece of code should be refactored or not. It depends on how much maintenance it is going to receive, how much it is being used and many more factors. Refactoring is expensive and in most cases, it should only be done after evaluating the benefits. Similarly, MSR could be used to identify whether a problem occurrence has already caused problems in the past. Without such means of gathering information, there is no reliable information about whether the detected occurrences are malicious.

Severety of detected cases

Detected cases have been examined to get an indication of whether they are malicious to a project. The indication is an opinion and not based on facts such as maintenance history.

The **Data Class** Code Smell appears in most projects. The worst case of a Data Class is a class that only has unprotected attributes and no methods. Such a class is probably a risk because the responsibilities for data consistency is enforced upon all classes that depend on the class. Data Class instances that do protect their data are usually less of a problem. Data classes that are only used locally, such as inside the package where they are defined or as an inner class only, are no big risk. The responsibilities for the data are also kept local. Data Classes that are used by multiple packages may be a risk as they do not always enforce data consistency and potential inconsistent data can travel across the application classes using the Data Class as envelope. Such classes should be refactored to at least check the consistency of the data inserted into the class and bear some responsibility for the values.

The **Large Class** Code Smell is found in most projects. A large class does not have to be problematic. Some classes are responsible for managing one big part of the program logic and need many methods or attributes to do so. Those classes should not be a big problem, even though they are harder to maintain than smaller classes. When a class also has multiple responsibilities, it may be large for the wrong reason. It is doing too many different things instead of being required to handle one big thing. The biggest problem is probably that the class is having multiple responsibilities. The lesser problem is that the class is also large. The latter will often be fixed if the multiple responsibilities are refactored.

The **Database Class** Anti-Pattern is specific to applications that manually facilitate persistent storage by using a database explicitly. Applications that use a persistence framework that enforces abstraction, such as the Hibernate[1] framework will probably never get this Anti-Pattern. In practice, the increased amount of database and error handling caused by this Anti-Pattern is indeed a problem. The absence of an abstraction and simplification of error management will require much more coding in the long run than implementing an abstraction layer in the first place. Not using an abstraction layer should be a careful consideration. Even though an abstraction layer might require slightly more processing power and initial design labor, it will probably pay off in the long run.

There is some uncertainty about whether the **Ravioli Code** Anti-Pattern is a real Anti-Pattern. The concept of Ravioli Code is that classes are using many abstractions and delegations. This is actually a good design principle. In the case of the Ravioli Code Anti-Pattern, the abstractions are overdone. Whether and when an overdone-good principle becomes an Anti-Pattern can be difficult to say. The detected cases have indeed a far stronger level of abstraction and amount of abstract relations than average classes. They truly are outliers. Very strong indications have not been found, meaning that only just strong indications can be evaluated. From those cases, it can be seen that the code is indeed more complex and requires more searching than average class relations do. After examining the structure, the user finds himself swamped in the open source files and still has only a limited vision of what the code should do. The cases are not bad, they are more difficult to comprehend. This Anti-Pattern is probably only a risk when it is encountered in an even more excessive form. Until then, it can be considered ‘mostly harmless’.

Lesser versions of **The Blob** have been detected in some projects. In general, these lesser cases are not harmful although the candidate classes for The Blob are often large with lots of dependencies and responsibilities. They are always much larger than what can be considered appropriate for a class and in most cases it is rather easy to point out at least two responsibilities that should have been in different classes. Refactoring is strongly suggested because The Blob will probably only grow in size if it is being maintained as it is.

The **Tower of Voodoo** can occasionally be found in projects, both in lesser and greater form. Lesser versions of this Anti-Pattern are Exception-class inheritance trees or extensions to data structures. The top levels of the tower are often nearly empty and poorly documented. They are almost harmless due to their limited addition in functionality and responsibilities. Such examples could be examined and redesigned. Proper design could point out that such extensions are not needed. Greater versions of the Tower of Voodoo are typically well-designed classes being extended by other classes using multiple levels of extension and at each level, adding a significant amount of functionality. The problem is that the extensions may lack explicit definition. Such cases may be harmful to development.

[1] http://www.hibernate.org/

Changes to one level of the tower typically result in changes to other levels and for every change, effects on the entire structure must be understood and accounted for. Should there be a good reason to use many levels of inheritance in a structure, then it might be best to implement interfaces on intermediate levels. The interfaces could be used to explicitly define the added functionality for that level.

Violations of the **Single Responsibility Principle** are common in all projects. The violations occur in the entire range from weak to very strong. In many cases, granting multiple roles to one class is a matter of practicality. It is simply convenient to give one class multiple roles because that makes the class 'glue' subcomponents together. Every role can be seen as one direction or subcomponent. A class with multiple roles creates a relation between those components. In most cases, connecting components to each other this way is not a problem. This means that the weaker cases of SRP violations are in fact violations but they do not cause problems. More severe cases of SRP violations are considered bad. When a class has more than two roles, it may be doing too much. It may still be for convenience but there is a risk that the class will grow and start doing too much. In general, the SRP should not be a strict principle but it should be a guideline to keep the number of responsibilities to a minimum.

The **Interface Segregation Principle** is often violated. Some interfaces appear to be focused on one role but eventually only a subset of the methods are being used. Other interfaces appear to violate the Single Responsibility Principle, causing them to violate the ISP as well. The latter case is probably a bad case that should be avoided by keeping the interfaces compact. The weaker cases may be unavoidable as interfaces need to define multiple methods to create a complete service for one role. There will eventually be situations where only a small portion of the role is being used. For such cases, violations of this principle are probably harmless.

The **Dependency Inversion Principle** suffers from occasional violations in the various projects. Most of the cases are rather innocent. Interfaces or abstract classes that depend on a custom list, a final class or exception type are no severe cases. Interfaces that explicitly depend on logic or main classes of a framework are more rare and they could be a problem. However, there is no clear indication that they actually cause any problems.

The **Acyclic Dependencies Principle** is occasionally violated. Whenever it is, there are automatically multiple entities that violate it. In general, any violation of this principle is wrong. Violations are not necessarily malicious to a program but they do always pose a risk that maintenance will be hindered due to a cyclic dependency. In many cases, cyclic dependencies are caused by a 'utility' package that contains classes that cannot be associated with one another package explicitly but that together do have relations in all directions. Classes that use a utility class from such a package might find themselves entangled in cyclic dependencies. In general, utility packages are a risk and should be avoided. Cycles also often occur from framework dependencies to the application logic and vice versa. The relations of framework classes should be designed properly because a framework should not depend on application logic. Furthermore, once the boundaries between framework and detail levels start to blur, cyclic dependencies will occur with every addition to the application logic.

6.1.2 Possible relations between occurrences

The data sets in Appendix B show that there may be relations between the various properties of patterns. Some of the relations which are observed multiple times are explained here. Observations are not only based on probability values from the tables, they are also examined in the source code of projects.

Large classes suffer from Single Responsibility Principle violations. Although the relation may be obvious, it is not trivial. Classes that are large have many methods and/or attributes and it often happens that these are not used for the same task. In general, many of the size metrics are related. If one type of size metric is inflated, the other is likely to follow.

Having many methods in a class can result in unrelated attributes. When methods require some attributes to perform their tasks, they often do not share the attributes. This can result in attributes that are used for a few methods, causing multiple sets of attributes to appear in a class when it has many methods.

A good part of the abstract classes that depend on concrete classes also suffer from Single Responsibility Principle violations and large class size. Some abstract classes serve as a base class that provide multiple services and use concrete classes to do so. These base services can be extended by concrete classes.

6.2 Answers and validity

6.2.1 Answers to the research questions

The research questions presented in Section 1.3 can be answered based on the results of the empirical evaluation.

- *Can heuristics be formulated that can be used to automatically detect Anti-Patterns and Design Principle violations in software systems?* Yes. The heuristics presented in Chapter 4 are a selection of the Code Smells, Anti-Patterns and Design Principle violations that can be automatically detected.
- *Can a program be built that can use these heuristics to find the design problems automatically and that can tell the reasons why it has found any problem?* Yes. The Anti-Pattern Scanner is the tool developed during this project and it is explained in Chapter 3. This tool has proven itself during the evaluation.
- *Do Anti-Patterns and Design Principle violations occur in software systems? Are they common or rare?* Yes. The Code Smells are more common than the Anti-Patterns. Design Principle violations also occur frequently. Many of these design problems occur multiple times in a software project. Full information of the detected cases can be found in Section 5.2.
- *Are they as severe and full-featured as described in literature?* The Anti-Patterns often occur in a lesser form, such as with only a few of their ‘bad’ aspects. They rarely occur in their full form. Code Smells and Design Principle violations are more likely to occur fully. The reason for this is that they have fewer aspects that need to be

present before the sought design problem is detected. The occurrences are discussed in Section 5.2.

- *For those occurrences, can an indication be given whether they are malicious to a project?* It appears that only the most severe cases of a design problem are malicious. This has been estimated by examining the source code and the history of these problems, where possible. This study cannot answer this question with hard facts. More information is necessary to do so. The severeness of the design problems is discussed in Section 6.1.

6.2.2 Threats to validity

The results acquired with this exploratory study yield some interesting facts regarding the presence and properties of Anti-Patterns and Design Principles in software systems. These discoveries should be interpreted with some regard to the limitations of this study. There are various reasons for why the results may be inaccurate or possibly even incorrect.

The eight projects being examined in this study are a selection of the available software systems for testing. At least two of these systems are no longer being developed and are mainly used for testing and tuning the heuristics. The other projects are projects that are receiving active development by a professional team, but even those projects have their limitations. The chosen projects do only cover a small portion of the available themes for which software can be used. With this limited selection of projects and corresponding product categories, only a small part of the range of software products is being covered. The results are only partially representative for software development in general. Furthermore, all projects are Java projects. The results may be Java-specific although many of the results are likely to be similar for other programming languages. Similarly, typical results of the Anti-Pattern Scanner may differ per software development team. With the sample set of software projects being rather small and the size of projects being small to medium size projects, the results may not be the same for all kinds of software projects. Also for these reasons, this study is meant to explore Anti-Patterns and thus does not provide an exhaustive list of facts.

For some Anti-Patterns and Design Principles, no ‘very strong positive’ cases have been found. Furthermore, a generally small number of positive cases may have been found, meaning that it can be difficult to claim meaningful facts about the patterns. There are many reasons for having small numbers of positive indications. The chosen projects could be an ‘unlucky’ choice of projects, having none of those problems at all. In general, this means that the sample set of examined projects is small. Furthermore, the heuristics may be incorrect or provide a low rating in general. Without a good sample set of positive and negative cases, it is difficult to adjust the heuristics to determine what the thresholds are. Some of the Anti-Patterns are only theoretical patterns that are the result of brainstorming sessions. They might not even be Anti-Patterns or the pattern can be so rare that it may never be found. The solution to each of these problems is to examine more projects. Should more cases be found then the heuristics can be adjusted. Should no cases every be found then that could be seen as a positive result too. Not having an Anti-Pattern is also a research result and even better: it is probably good news for the software projects.

The Code Smells, Anti-Patterns and Design Principles have been interpreted and a detection heuristic has been designed. The interpretation of these patterns taints the heuristics in some ways. The interpretations are only a limited view of the patterns and will not fully cover all possible cases of all patterns. This is specifically true for some of the heuristics presented in Chapter 4 which are explained as being only a limited approach. However, it is generally true for all heuristics as they are the interpretations of the design problems by the author. The heuristics are open to discussion and interpretation. They probably always will be. A brainstorming session with experts on this field of software engineering could yield positive contributions to the heuristics.

The suggested heuristic for the Acyclic Dependencies Principles is flawed in the sense that it's search space is very large. Let the number of outgoing dependencies of a package by D and the number of packages in a software system be P, then the search space could theoretically be of size D^P. Even though it is always much less in practice, it can still be huge. Due to optimizations in the scanner for this principle, such as limiting the maximum value of P in the search space, the results may not be entirely correct. This is especially true for the JMonkeyEngine project where an excessive number of cycles require memory optimizations for the program. Without the optimizations, the search space would require over 30 gigabytes of computer memory per processing core. Limiting the search space reduces the number of cycles detected and theoretically could cause the program to return a negative indication, should all cycles have been missed. In practice however, the major part of the cycles have been found at low search depths, meaning that the indications are correct.

Taking into consideration the small sample set of projects examined, it should be clear that any statistics gathered in this study may not be statistically valid. In general, the statistics and specifically the conditional probabilities discussed in Section 6.1.2 should be used as indications for closer examination or further research. Actual statistical significance has not been tested and proven and correlations have not been examined. Furthermore, it is questionable whether the fuzzy results of the various negative and positive indications (e.g., 'very', 'strong', ...) can translate to crisp statistical values and whether such values can be used in probability formulae.

Last and probably least, the Anti-Pattern Scanner may not be free of bugs. While it is shown that many results are significant and usable, software problems may always threaten the validity of the program output. There is even a remote possibility that, should it ever be proven that a specific pattern threatens all validity of a program and that the Anti-Pattern Scanner possesses this pattern, then it's results have never been valid in the first place. Should such proof ever be constructed with the Anti-Pattern Scanner, then it can at least be shown that it is not correct.

Chapter 7

Conclusion and future work

7.1 Conclusion

During this project, we have developed a design problem detection tool called the Anti-Pattern Scanner. This tool has proven itself on eight software systems in total. The scanner program can show which software entities have what potential problems and why. The scanner program is still a prototype that is meant to be used for research, yet the initial results are promising.

The tool currently detects two Code Smells, four Anti-Patterns and four Design Principle violations. The results of our empirical evaluation with the software projects show:

- Many of the design problems occur frequently in software systems. Design Principle violations and Code Smells are common, Anti-Patterns are rare.
- Our approach of using modules for scanning design problem aspects and combining the results using fuzzy values is effective. This approach can give a good categorization of problem strengths and it can tell the reasoning behind indications.
- The accuracy of the indications is good, typically 50% to 100% depending on problem type.
- For most of these problem types, indications can be given that they are malicious to a software project. None of these problems have been shown to be problematically harmful to any project.

The Code Smells and the Design Principles are interesting as they can often be found in software systems, even those that have been carefully designed and that are receiving active development by a dedicated team. The Anti-Patterns are more rare but do occasionally appear in such software systems, often in a lesser form. Many Anti-Patterns seem to be more theoretical than Design Principles. For some of these patterns, this is the first time concrete cases have been found.

This project provides various hooks for further research. Anti-Patterns and Desing Principle violations are shown to occur in software projects. This project delivers a working program that can show the presence of these problems in any Java project. This project also suggests many heuristics for implementations of more Design Principles. It also offers directions and questions which could be the aim of further research. Due to all of these results, this scanner program is a successful proof of concept, the study is a broad exploration and this research will hopefully be a basis for other research.

7.2 Future work

The results discussed in this chapter may be the tip of the iceberg of what can be found in software systems. Only a few projects have been examined and only a few patterns have been examined in these projects. Examining more software projects could yield more and stronger positive cases of the various patterns. Expanding the exploratory study to more projects would result in a larger sample set and would reduce some of the threats discussed in the previous section.

More heuristics have been suggested in Chapter 4. Most of these heuristics can be implemented in the Anti-Pattern Scanner as it is now. Any study with more Design Principles would probably be more broader that this study and could yield many more interesting facts about the presence and severity of those principles. There are many more Code Smells and Anti-Patterns available both from literature and on the Internet. Adding more of such patterns to the Anti-Pattern Scanner could yield either a broader or a more focused study, depending on the choice of patterns. In general, continued development on the Anti-Pattern Scanner could result in a larger library of Code Smells and Design Principles as well as a selection of Anti-Patterns that consist partly of components from that library. Examination of those patterns might reveal more occurrences and relations between them, possibly revealing categories of problems and correlations of problems.

In general, few relations have been observed. The lack of clear relations could be due to the limited number and size of projects that have been examined. It may also be caused by the diversity of chosen patterns for this empirical evaluation. The choice of patterns is meant to create a wide view of which problem types can be found in projects and it was meant to have as little overlap as possible. A study that would focus on multiple different but similar problems might yield more interesting relations. Similarly, conditional probability could be useful if properties like change-proneness or number of bugs are added to the tables. The probabilities should also be tested for significance. Probabilities are influenced by the general amount of positive indications that scanners give. When A is often positive and B is not, then $p(A|B)$ can be a high probability. This may be a relation, but it could also be caused due to the general amount of positive results for A and B.

An interesting addition to the examined cases would be to see how those cases are developed over time. To create a timeline of the development of any problem, MSR could be used. Seeing how problems appear in the software, how they shrink, grow and when they combine would be interesting. MSR could also be used to answer questions regarding the appearance of the problem types. Such as, are the Anti-Patterns a symptom of underlying problems or a cause? Can they be detected during early development or can they only be found once they have fully appeared? Some of the heuristics suggest both an MSR approach and a non-MSR approach. Both approaches could be compared against each other, allowing one to validate the other.

Another direction for research would be to include dynamic analysis to the detection possibilities. Using the current static analysis approach, it may be hard to detect behavioral problems. Dynamic analysis would complement the current analysis approach and open up the way for more heuristics.

Last but not least, the Anti-Pattern Scanner could be improved to be much more user-friendly. The user interface could be improved to be much easier to use and the result output could have a lot of explanation and helpfulness added to it. With more development to the program itself and the addition of more problem heuristics, the program could be useful for everyday programming. In many ways it could be similar to the code assistant programs already available. Even with the current set of detected problems, it would complement existing programs. Alternatively, the heuristics presented in this project could be implemented in some of the existing programs.

Bibliography

[1] Lerina Aversano, Gerardo Canfora, Luigi Cerulo, Concettina Del Grosso, and Massimiliano Di Penta. An empirical study on the evolution of design patterns. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 385–394, New York, NY, USA, 2007. ACM.

[2] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Softw. Eng.*, 22(10):751–761, 1996.

[3] Dirk Beyer and Claus Lewerentz. Crocopat: Efficient pattern analysis in object-oriented programs. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 294, Washington, DC, USA, 2003. IEEE Computer Society.

[4] James M. Bieman, Greg Straw, Huxia Wang, P. Willard Munger, and Roger T. Alexander. Design patterns and change proneness: An examination of five evolving systems. In *METRICS '03: Proceedings of the 9th International Symposium on Software Metrics*, page 40, Washington, DC, USA, 2003. IEEE Computer Society.

[5] William J. Brown, Raphael C. Malveau, and Hays W. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998. (ISBN-13: 978-0471197133).

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[7] Gennaro Costagliola, Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery by visual language parsing. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 102–111, Washington, DC, USA, 2005. IEEE Computer Society.

[8] Karim Dhambri, Houari Sahraoui, and Pierre Poulin. Visual detection of design anomalies. In *CSMR '08: Proceedings of the 2008 12th European Conference on*

Software Maintenance and Reengineering, pages 279–283, Washington, DC, USA, 2008. IEEE Computer Society.

[9] Jens Dietrich and Chris Elgar. Towards a web of patterns. *Web Semant.*, 5(2):108–116, 2007.

[10] Maged Elaasar, Lionel C. Bri, and Yvan Labiche. A metamodeling approach to pattern specification and detection, 2006.

[11] El Emam, Kalhed, Benlarbi, Saïda, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE Trans. Softw. Eng.*, 27(7):630–650, 2001.

[12] Norman E. Fenton and Niclas Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, 2000.

[13] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[14] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley Professional; 1 edition, 1999. (ISBN 978-0201485677).

[15] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[16] Harald C. Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26:26–33, 2009.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. (ISBN-13: 978-0201633610).

[18] Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.*, 34(5):667–684, 2008.

[19] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, 2005.

[20] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19(2):77–131, 2007.

[21] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 75–84, Washington, DC, USA, 2009. IEEE Computer Society.

[22] Foutse Khomh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Playing roles in design patterns: An empirical descriptive and analytic study. volume 0, pages 83–92, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[23] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006. (ISBN-13: 978-3540244295).

[24] Hakjin Lee, Hyunsang Youn, and Eunseok Lee. A design pattern detection technique that aids reverse engineering, 2008.

[25] Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[26] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall; 1st edition, 2002. (ISBN 978-0135974445).

[27] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Trans. Softw. Eng.*, 33(1):2–13, 2007.

[28] Matthias Meyer. Pattern-based reengineering of software systems. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 305–306, Washington, DC, USA, 2006. IEEE Computer Society.

[29] Petru Florin Mihancea and Radu Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 92–101, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 120, Washington, DC, USA, 2000. IEEE Computer Society.

[31] Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Francoise Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, 2010.

[32] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium*, page 15, Washington, DC, USA, 2005. IEEE Computer Society.

[33] Jörg Niere, Wilhelm Schäfer, Jörg P. Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 338–348, New York, NY, USA, 2002. ACM.

[34] Rocco Oliveto, Foutse Khomh, Giuliano Antoniol, and Yann-Gael Gueheneuc. Numerical signatures of antipatterns: An approach based on b-splines. *Software Maintenance and Reengineering, European Conference on*, 0:248–251, 2010.

[35] Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness, 2008.

[36] Martin Pinzger, Katja Graefenhain, Patrick Knab, and Harald C. Gall. A tool for visual understanding of source code dependencies. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 254–259, Washington, DC, USA, 2008. IEEE Computer Society.

[37] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[38] Jason McC. Smith, Jason Mcc Smith, David Stotts, and David Stotts. Elemental design patterns - a link between architecture and object semantics. Technical report, Proceedings of OOPSLA 2002, 2002.

[39] Alecsandar Stoianov and Ioana Sora. Detecting patterns and antipatterns in software using prolog rules. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 253–258, Department of Computers, Politehnica University of Timisoara, Romania, 2010.

[40] David Stotts, Jason McC. Smith, and Jason Mcc Smith. Spqr: Flexible automated design pattern extraction from source code. In *In 18th IEEE Intl Conf on Automated Software Engineering*, pages 215–224. IEEE Computer Society Press, 2003.

[41] Guilherme Travassos, Forrest Shull, Michael Fredericks, and Victor R. Basili. Detecting defects in object-oriented designs: using reading techniques to increase software quality. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 47–56, New York, NY, USA, 1999. ACM.

[42] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32(11):896–909, 2006.

[43] E. Van Emden and L. Moonen. Java quality assurance by detecting code smells. In *WCRE '02: Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, page 97, Washington, DC, USA, 2002. IEEE Computer Society.

[44] H. Vliet. *Software engineering: principles and practice*. Safari Books Online. John Wiley & Sons, 2008.

[45] Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Trans. Softw. Eng.*, 30(12):904–917, 2004.

Appendix A

Glossary

Commonly used terms and abbreviations.

Abstraction: A simplification or definition that is used to hide details from clients. In the Java programming language, Abstract classes or Interfaces are used for this purpose.

Anti-Pattern Scanner: The program designed and build for this Thesis Project.

Anti-Pattern: Known ‘bad’ Design Pattern or a pattern that is applied in the wrong situation.

ASG: Abstract Syntax Graph. A graph representation of the abstract syntax of a program’s code, e.g. leaving out many of the fine details specified in the source. Not to be confused with Abstract Semantic Graph.

AST: Abstract Syntax Tree. A tree representation of the abstract syntax of a program’s code.

Bad Smell: Poorly written section of code.

C++: A programming language.

Change-proneness: How often a software object (OO) or class changes or can be expected to change.

Class Diagram: An UML diagram that shows the relations between classes.

Class: An uninstantiated object (OO). The template from which objects can be created when a program is executed.

Code Smell: See Bad Smell.

Compiler: A program that translates another program written in a high-level language into machine language so that it can be executed.

Concrete implementation: An implementation that contains details and that is not abstract. See also Abstraction.

Coupling: A situation where two or more objects in OOP interact with each other in such a way that changing one of the objects will usually require changes to the other object(s) as well. This dependency can be problematic during maintenance.

CVS: Concurrent Versioning System. A Repository.

Design Pattern: Template that describes how objects should interact in an Object-Oriented program. This template serves as a known solution for a problem in a specific context.

Design Principle: Guideline for software design. For more information, see [26].

Design Smell: See Anti-Pattern.

Eclipse: A programming environment, often used to produce code for Java or other languages, see “http://eclipse.org/”.

EDP: Elemental Design Pattern. A basic building block for OOP, such as ‘Inheritance’.

Error-proneness: How often a software object or class has produced an error or can be expected to produce an error.

Gang-of-Four (GOF): The four authors of [17].

Heuristic: A rule or set of rules meant to increase the probability of solving a problem, without the exhaustive application of an algorithm.

IDE: Integrated Development Environment. An environment for programming (such as Eclipse).

Java: A programming language.

Meta-model: An abstract representation of a software system.

Metric: A measurement type of a software system, such as (number of) Lines of Code (LOC) or Number of Comments (NOC).

MSR: Mining Software Repositories. A technique(s) for examining a software repository(s).

Rating: In the context of *Anti-Pattern Scanner*, a rating is a value that indicates how strongly an entity participates in a given Anti-Pattern or aspect of an Anti-Pattern. A strong rating means that a given entity is a candidate, a negative rating means that it is not. See Section 3.1.3 for more information.

Refactoring: Modifying a software system to repair or change it.

Repository: A storage for software code and artifacts. Usually some kind of automatic versioning is used so changes can be backtracked.

Rulebase: A set(s) of rules that when combined with facts, can identify occurrences of items specified by the rules.

Scanner: In the context of *Anti-Pattern Scanner*, a scanner is a module that rates (an aspect of) an Anti-Pattern. See Section 3.1.4 for more information.

SQL: Simple Query Language. A language that can be used to query databases.

UML: Unified Modeling Language. A standardized, universal modeling language for software engineering.

Appendix B

Data sets

This appendix contains all the data tables of all the projects. Every section begins with a table with the amount of results for every scanner, for every indication strength. The second table in every section contains the conditional probabilities for all top level scanners: Large Class Code Smell (LargeClass), Data Class Code Smell (DataClassFilter), Single Responsibility Principle (SingleResponsibility), Acyclic Dependencies Principle (CyclicPackageDependency), Dependency Inversion Principle (ConcreteDependentAbstraction), Interface Segregation Principle (InterfaceSegregation) and the four Anti-Patterns (DatabaseClass, RavioliCode, TheBlob, TowerOfVoodoo). The last table contains the conditional probabilities for all separate fragment scanners. The full description of every scanner can be found in Section 4. For each of the conditional probability tables, only 'Strong' and 'Very Strong' positive indications count as positive. All other indications ('Weak' and lower) count as negative. The conditional probabilities should be seen as indications for closer examination. The probabilities indicate that there could be relations. Statistical significance has not been tested and is threatened by the general number of positive indications per scanner and the number of samples in general.

The scanner design supports all kinds of entities as input. In practice, the 'Class' entity, 'Package' entity or the 'Method' entity are most convenient as starting point for any scanner implementation. The scanners are restricted to one type of entity at compile time, meaning that some scanners cannot be compared to each other because they accept different types of entities. This means that they have mostly empty rows in the conditional probability tables. The following types are accepted as input types:

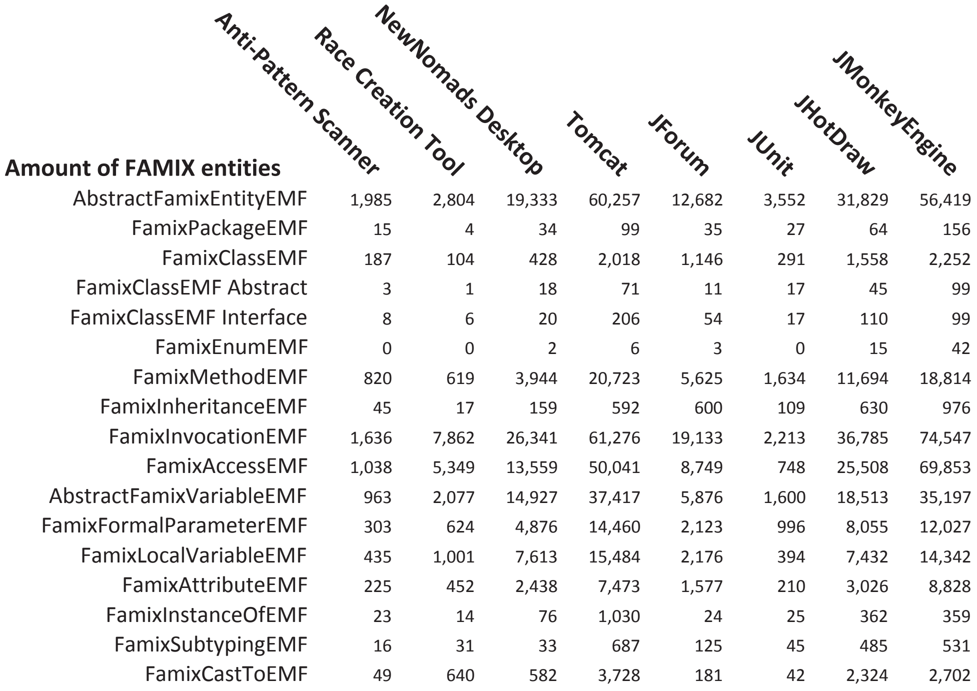
Package Scanners 'DatabasePackage', 'CyclicPackageDependency'.

Method Scanners 'UniversalDatabaseMethod', 'NonDatabasePackageInvocation'.

Inheritance Scanner 'InheritanceTarget'.

Class All other scanners (although some automatically return 'Strong Negative' for incompatible types such as (non-)abstract or (non-)interface).

Due to search tree pruning, some scanners were never activated and never got a chance to register themselves with the framework. The result is that for some projects, the scanners 'NonDatabasePackageInvocation', 'UniversalDatabaseMethod' and 'UniversalDataStructure' do not appear in the probability tables.

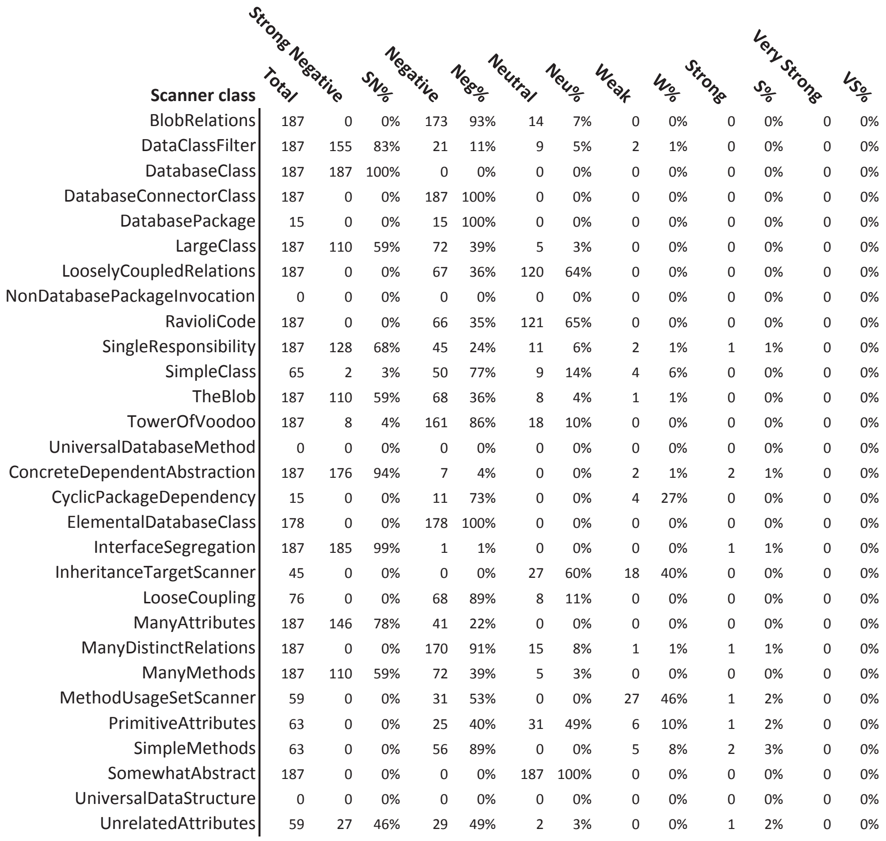


Amount of FAMIX entities	Anti-Pattern Scanner	Race Creation Tool	NewNomads Desktop	Tomcat	JForum	JUnit	JHotDraw	JMonkeyEngine
AbstractFamixEntityEMF	1,985	2,804	19,333	60,257	12,682	3,552	31,829	56,419
FamixPackageEMF	15	4	34	99	35	27	64	156
FamixClassEMF	187	104	428	2,018	1,146	291	1,558	2,252
FamixClassEMF Abstract	3	1	18	71	11	17	45	99
FamixClassEMF Interface	8	6	20	206	54	17	110	99
FamixEnumEMF	0	0	2	6	3	0	15	42
FamixMethodEMF	820	619	3,944	20,723	5,625	1,634	11,694	18,814
FamixInheritanceEMF	45	17	159	592	600	109	630	976
FamixInvocationEMF	1,636	7,862	26,341	61,276	19,133	2,213	36,785	74,547
FamixAccessEMF	1,038	5,349	13,559	50,041	8,749	748	25,508	69,853
AbstractFamixVariableEMF	963	2,077	14,927	37,417	5,876	1,600	18,513	35,197
FamixFormalParameterEMF	303	624	4,876	14,460	2,123	996	8,055	12,027
FamixLocalVariableEMF	435	1,001	7,613	15,484	2,176	394	7,432	14,342
FamixAttributeEMF	225	452	2,438	7,473	1,577	210	3,026	8,828
FamixInstanceOfEMF	23	14	76	1,030	24	25	362	359
FamixSubtypingEMF	16	31	33	687	125	45	485	531
FamixCastToEMF	49	640	582	3,728	181	42	2,324	2,702

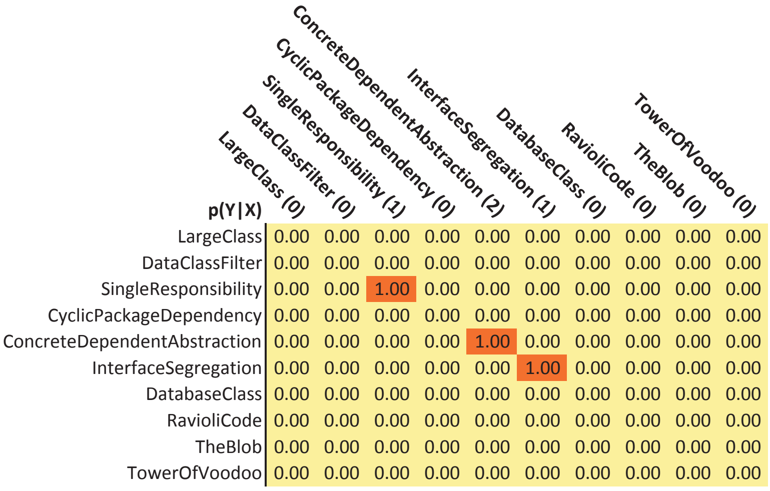
Figure B.1: Amounts of FAMIX entities found in each project. (This figure is a copy of figure 5.1.)

This page is intentionally left empty. The data sets start at the next page.

B.1 The Anti-Pattern Scanner



Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	187	0	0%	173	93%	14	7%	0	0%	0	0%	0	0%
DataClassFilter	187	155	83%	21	11%	9	5%	2	1%	0	0%	0	0%
DatabaseClass	187	187	100%	0	0%	0	0%	0	0%	0	0%	0	0%
DatabaseConnectorClass	187	0	0%	187	100%	0	0%	0	0%	0	0%	0	0%
DatabasePackage	15	0	0%	15	100%	0	0%	0	0%	0	0%	0	0%
LargeClass	187	110	59%	72	39%	5	3%	0	0%	0	0%	0	0%
LooselyCoupledRelations	187	0	0%	67	36%	120	64%	0	0%	0	0%	0	0%
NonDatabasePackageInvocation	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
RavioliCode	187	0	0%	66	35%	121	65%	0	0%	0	0%	0	0%
SingleResponsibility	187	128	68%	45	24%	11	6%	2	1%	1	1%	0	0%
SimpleClass	65	2	3%	50	77%	9	14%	4	6%	0	0%	0	0%
TheBlob	187	110	59%	68	36%	8	4%	1	1%	0	0%	0	0%
TowerOfVoodoo	187	8	4%	161	86%	18	10%	0	0%	0	0%	0	0%
UniversalDatabaseMethod	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
ConcreteDependentAbstraction	187	176	94%	7	4%	0	0%	2	1%	2	1%	0	0%
CyclicPackageDependency	15	0	0%	11	73%	0	0%	4	27%	0	0%	0	0%
ElementalDatabaseClass	178	0	0%	178	100%	0	0%	0	0%	0	0%	0	0%
InterfaceSegregation	187	185	99%	1	1%	0	0%	0	0%	1	1%	0	0%
InheritanceTargetScanner	45	0	0%	0	0%	27	60%	18	40%	0	0%	0	0%
LooseCoupling	76	0	0%	68	89%	8	11%	0	0%	0	0%	0	0%
ManyAttributes	187	146	78%	41	22%	0	0%	0	0%	0	0%	0	0%
ManyDistinctRelations	187	0	0%	170	91%	15	8%	1	1%	1	1%	0	0%
ManyMethods	187	110	59%	72	39%	5	3%	0	0%	0	0%	0	0%
MethodUsageSetScanner	59	0	0%	31	53%	0	0%	27	46%	1	2%	0	0%
PrimitiveAttributes	63	0	0%	25	40%	31	49%	6	10%	1	2%	0	0%
SimpleMethods	63	0	0%	56	89%	0	0%	5	8%	2	3%	0	0%
SomewhatAbstract	187	0	0%	0	0%	187	100%	0	0%	0	0%	0	0%
UniversalDataStructure	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
UnrelatedAttributes	59	27	46%	29	49%	2	3%	0	0%	1	2%	0	0%



p(Y|X)	LargeClass (0)	DataClassFilter (0)	SingleResponsibility (1)	CyclicPackageDependency (0)	ConcreteDependentAbstraction (2)	InterfaceSegregation (1)	DatabaseClass (0)	RavioliCode (0)	TheBlob (0)	TowerOfVoodoo (0)
LargeClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

p(Y|X)	**BlobRelations (0)**	**ConcreteDependentAbstraction (2)**	**CyclicPackageDependency (0)**	**DataClassFilter (0)**	**DatabaseClass (0)**	**DatabaseConnectorClass (0)**	**DatabasePackage (0)**	**ElementalDatabaseClass (0)**	**InheritanceTargetScanner (0)**	**InterfaceSegregation (1)**	**LargeClass (0)**	**LooseCoupling (0)**	**LooselyCoupledRelations (0)**	**ManyAttributes (0)**	**ManyDistinctRelations (1)**	**ManyMethods (0)**	**MethodUsageSetScanner (1)**	**PrimitiveAttributes (1)**	**RavioliCode (0)**	**SimpleClass (0)**	**SimpleMethods (2)**	**SingleResponsibility (1)**	**SomewhatAbstract (0)**	**TheBlob (0)**	**TowerOfVoodoo (0)**	**UnrelatedAttributes (1)**
BlobRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabasePackage	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LargeClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyDistinctRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyMethods	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MethodUsageSetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PrimitiveAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleMethods	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00
SomewhatAbstract	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UnrelatedAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00

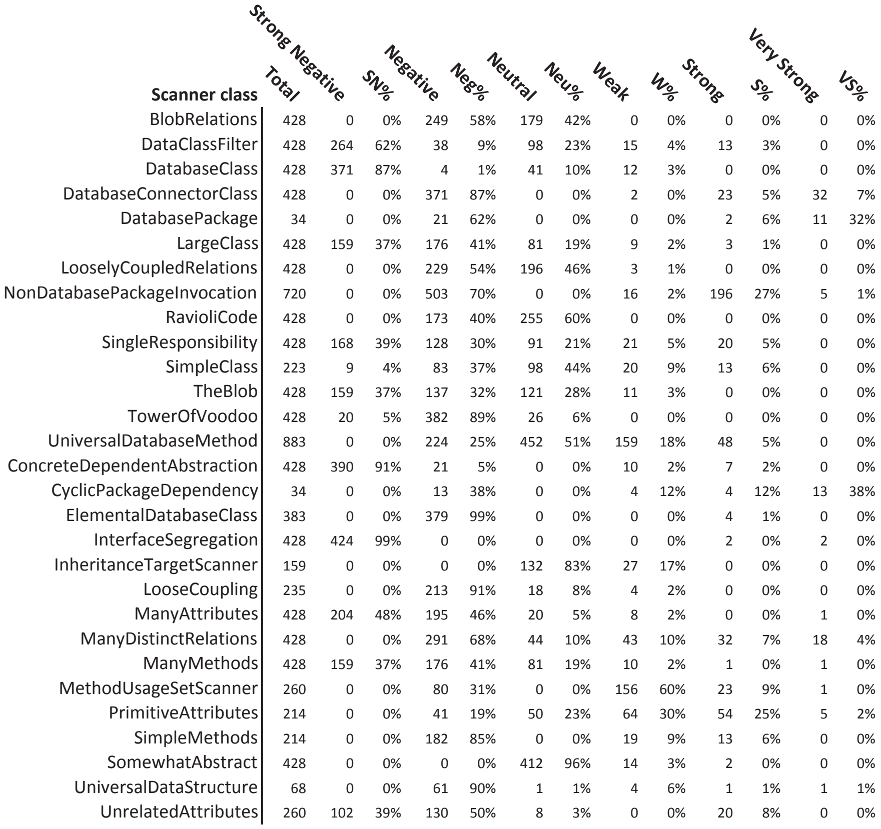
B.2 Race Creation Tool

Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	104	0	0%	77	74%	26	25%	1	1%	0	0%	0	0%
DataClassFilter	104	70	67%	4	4%	13	13%	16	15%	1	1%	0	0%
DatabaseClass	104	88	85%	0	0%	1	1%	15	14%	0	0%	0	0%
DatabaseConnectorClass	104	0	0%	88	85%	0	0%	15	14%	0	0%	1	1%
DatabasePackage	4	0	0%	2	50%	0	0%	1	25%	0	0%	1	25%
LargeClass	104	57	55%	37	36%	7	7%	2	2%	1	1%	0	0%
LooselyCoupledRelations	104	0	0%	44	42%	60	58%	0	0%	0	0%	0	0%
NonDatabasePackageInvocation	195	0	0%	74	38%	0	0%	26	13%	95	49%	0	0%
RavioliCode	104	0	0%	42	40%	62	60%	0	0%	0	0%	0	0%
SingleResponsibility	104	79	76%	1	1%	5	5%	9	9%	10	10%	0	0%
SimpleClass	47	0	0%	17	36%	13	28%	16	34%	1	2%	0	0%
TheBlob	104	57	55%	27	26%	14	13%	5	5%	1	1%	0	0%
TowerOfVoodoo	104	6	6%	97	93%	1	1%	0	0%	0	0%	0	0%
UniversalDatabaseMethod	228	0	0%	2	1%	105	46%	87	38%	34	15%	0	0%
ConcreteDependentAbstraction	104	97	93%	7	7%	0	0%	0	0%	0	0%	0	0%
CyclicPackageDependency	4	0	0%	0	0%	0	0%	0	0%	2	50%	2	50%
ElementalDatabaseClass	96	0	0%	93	97%	0	0%	0	0%	3	3%	0	0%
InterfaceSegregation	104	104	100%	0	0%	0	0%	0	0%	0	0%	0	0%
InheritanceTargetScanner	17	0	0%	0	0%	15	88%	2	12%	0	0%	0	0%
LooseCoupling	47	0	0%	45	96%	2	4%	0	0%	0	0%	0	0%
ManyAttributes	104	70	67%	29	28%	2	2%	2	2%	0	0%	1	1%
ManyDistinctRelations	104	0	0%	101	97%	1	1%	0	0%	1	1%	1	1%
ManyMethods	104	57	55%	37	36%	9	9%	1	1%	0	0%	0	0%
MethodUsageSetScanner	25	0	0%	1	4%	0	0%	22	88%	2	8%	0	0%
PrimitiveAttributes	47	0	0%	5	11%	13	28%	10	21%	6	13%	13	28%
SimpleMethods	47	0	0%	43	91%	0	0%	3	6%	0	0%	1	2%
SomewhatAbstract	104	0	0%	0	0%	104	100%	0	0%	0	0%	0	0%
UniversalDataStructure	19	0	0%	16	84%	1	5%	2	11%	0	0%	0	0%
UnrelatedAttributes	25	0	0%	6	24%	9	36%	0	0%	10	40%	0	0%

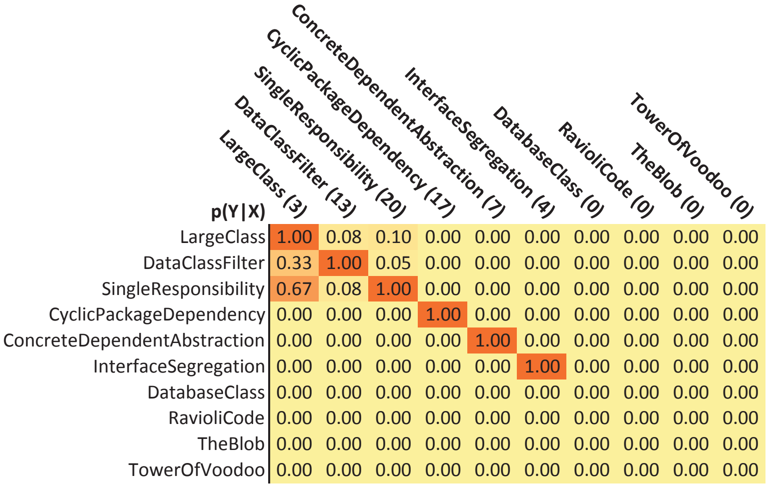
p(Y|X)	LargeClass (1)	DataClassFilter (1)	SingleResponsibility (10)	CyclicPackageDependency (4)	ConcreteDependentAbstraction (0)	InterfaceSegregation (0)	DatabaseClass (0)	RavioliCode (0)	TheBlob (1)	TowerOfVoodoo (0)
LargeClass	1.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	1.00	0.00
DataClassFilter	0.00	1.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	1.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	1.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	1.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

p(Y|X)	**BlobRelations (0)**	**ConcreteDependentAbstraction (0)**	**CyclicPackageDependency (4)**	**DataClassFilter (1)**	**DatabaseClass (0)**	**DatabaseConnectorClass (1)**	**DatabasePackage (1)**	**ElementalDatabaseClass (3)**	**InheritanceTargetScanner (0)**	**InterfaceSegregation (0)**	**LargeClass (1)**	**LooseCoupling (0)**	**LooselyCoupledRelations (0)**	**ManyAttributes (1)**	**ManyDistinctRelations (2)**	**ManyMethods (0)**	**MethodUsageSetScanner (2)**	**NonDatabasePackageInvocation (95)**	**PrimitiveAttributes (19)**	**RavioliCode (0)**	**SimpleClass (1)**	**SimpleMethods (1)**	**SingleResponsibility (10)**	**SomewhatAbstract (0)**	**TheBlob (1)**	**TowerOfVoodoo (0)**	**UniversalDataStructure (0)**	**UniversalDatabaseMethod (34)**	**UnrelatedAttributes (10)**
BlobRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.10	0.00	0.00	0.00	0.00	0.00	0.10
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	0.10
DatabasePackage	0.00	0.00	0.25	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LargeClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.50	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.10	0.00	1.00	0.00	0.00	0.00	0.10
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.50	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.10	0.00	1.00	0.00	0.00	0.00	0.10
ManyDistinctRelations	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	1.00	0.00	0.00	0.00	0.20
ManyMethods	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MethodUsageSetScanner	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.20	0.00	1.00	0.00	0.00	0.00	0.20
NonDatabasePackageInvocation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
PrimitiveAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.40	0.00	0.00	0.00	0.00	0.00	0.40
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.10	0.00	0.00	0.00	0.00	0.00	0.10
SimpleMethods	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.10	0.00	0.00	0.00	0.00	0.00	0.10
SingleResponsibility	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00	0.21	0.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00	1.00
SomewhatAbstract	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.50	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.10	0.00	1.00	0.00	0.00	0.00	0.10
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UniversalDataStructure	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UniversalDatabaseMethod	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
UnrelatedAttributes	0.00	0.00	0.00	1.00	0.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00	0.21	0.00	1.00	1.00	1.00	0.00	1.00	0.00	0.00	0.00	1.00

B.3 NewNomads Desktop



Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	428	0	0%	249	58%	179	42%	0	0%	0	0%	0	0%
DataClassFilter	428	264	62%	38	9%	98	23%	15	4%	13	3%	0	0%
DatabaseClass	428	371	87%	4	1%	41	10%	12	3%	0	0%	0	0%
DatabaseConnectorClass	428	0	0%	371	87%	0	0%	2	0%	23	5%	32	7%
DatabasePackage	34	0	0%	21	62%	0	0%	0	0%	2	6%	11	32%
LargeClass	428	159	37%	176	41%	81	19%	9	2%	3	1%	0	0%
LooselyCoupledRelations	428	0	0%	229	54%	196	46%	3	1%	0	0%	0	0%
NonDatabasePackageInvocation	720	0	0%	503	70%	0	0%	16	2%	196	27%	5	1%
RavioliCode	428	0	0%	173	40%	255	60%	0	0%	0	0%	0	0%
SingleResponsibility	428	168	39%	128	30%	91	21%	21	5%	20	5%	0	0%
SimpleClass	223	9	4%	83	37%	98	44%	20	9%	13	6%	0	0%
TheBlob	428	159	37%	137	32%	121	28%	11	3%	0	0%	0	0%
TowerOfVoodoo	428	20	5%	382	89%	26	6%	0	0%	0	0%	0	0%
UniversalDatabaseMethod	883	0	0%	224	25%	452	51%	159	18%	48	5%	0	0%
ConcreteDependentAbstraction	428	390	91%	21	5%	0	0%	10	2%	7	2%	0	0%
CyclicPackageDependency	34	0	0%	13	38%	0	0%	4	12%	4	12%	13	38%
ElementalDatabaseClass	383	0	0%	379	99%	0	0%	0	0%	4	1%	0	0%
InterfaceSegregation	428	424	99%	0	0%	0	0%	0	0%	2	0%	2	0%
InheritanceTargetScanner	159	0	0%	0	0%	132	83%	27	17%	0	0%	0	0%
LooseCoupling	235	0	0%	213	91%	18	8%	4	2%	0	0%	0	0%
ManyAttributes	428	204	48%	195	46%	20	5%	8	2%	0	0%	1	0%
ManyDistinctRelations	428	0	0%	291	68%	44	10%	43	10%	32	7%	18	4%
ManyMethods	428	159	37%	176	41%	81	19%	10	2%	1	0%	1	0%
MethodUsageSetScanner	260	0	0%	80	31%	0	0%	156	60%	23	9%	1	0%
PrimitiveAttributes	214	0	0%	41	19%	50	23%	64	30%	54	25%	5	2%
SimpleMethods	214	0	0%	182	85%	0	0%	19	9%	13	6%	0	0%
SomewhatAbstract	428	0	0%	0	0%	412	96%	14	3%	2	0%	0	0%
UniversalDataStructure	68	0	0%	61	90%	1	1%	4	6%	1	1%	1	1%
UnrelatedAttributes	260	102	39%	130	50%	8	3%	0	0%	20	8%	0	0%



p(Y|X)	LargeClass (3)	DataClassFilter (13)	SingleResponsibility (20)	CyclicPackageDependency (17)	ConcreteDependentAbstraction (7)	InterfaceSegregation (4)	DatabaseClass (0)	RavioliCode (0)	TheBlob (0)	TowerOfVoodoo (0)
LargeClass	1.00	0.08	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.33	1.00	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.67	0.08	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

p(Y|X)	BlobRelations (0)	ConcreteDependentAbstraction (7)	CyclicPackageDependency (17)	DataClassFilter (13)	DatabaseClass (0)	DatabaseConnectorClass (55)	DatabasePackage (13)	ElementalDatabaseClass (4)	InheritanceTargetScanner (0)	InterfaceSegregation (4)	LargeClass (3)	LooseCoupling (0)	LooselyCoupledRelations (0)	ManyAttributes (1)	ManyDistinctRelations (50)	ManyMethods (2)	MethodUsageSetScanner (24)	NonDatabasePackageInvocation (201)	PrimitiveAttributes (59)	RavioliCode (0)	SimpleClass (13)	SimpleMethods (13)	SingleResponsibility (20)	SomewhatAbstract (2)	TheBlob (0)	TowerOfVoodoo (0)	UniversalDataStructure (2)	UniversalDatabaseMethod (48)	UnrelatedAttributes (20)
BlobRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	1.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.03	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	0.38	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.02	0.50	0.04	0.00	0.15	0.00	1.00	1.00	0.05	0.00	0.00	0.00	0.00	0.00	0.10
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.14	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.67	0.00	0.00	1.00	0.44	0.50	0.21	0.00	0.10	0.00	0.00	0.00	0.55	0.00	0.00	0.00	0.00	0.00	0.55
DatabasePackage	0.00	0.00	0.29	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LargeClass	0.00	0.00	0.00	0.08	0.00	0.04	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.04	1.00	0.08	0.00	0.02	0.00	0.08	0.08	0.10	0.00	0.00	0.00	0.00	0.00	0.10
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.33	0.00	0.00	1.00	0.02	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyDistinctRelations	0.00	0.00	0.00	0.08	0.00	0.40	0.00	0.00	0.00	0.00	0.67	0.00	0.00	1.00	1.00	0.50	0.50	0.00	0.08	0.00	0.08	0.08	0.65	0.50	0.00	0.00	0.00	0.00	0.60
ManyMethods	0.00	0.00	0.00	0.08	0.00	0.02	0.00	0.00	0.00	0.00	0.67	0.00	0.00	0.00	0.02	1.00	0.04	0.00	0.02	0.00	0.08	0.08	0.10	0.00	0.00	0.00	0.00	0.00	0.10
MethodUsageSetScanner	0.00	0.14	0.00	0.08	0.00	0.09	0.00	0.00	0.00	0.00	0.67	0.00	0.00	1.00	0.24	0.50	1.00	0.00	0.07	0.00	0.08	0.08	0.25	0.50	0.00	0.00	0.00	0.00	0.20
NonDatabasePackageInvocation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.98	0.00
PrimitiveAttributes	0.00	0.29	0.00	0.69	0.00	0.11	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.10	0.50	0.17	0.00	1.00	0.00	0.69	0.69	0.05	0.50	0.00	0.00	0.00	0.00	0.10
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.02	0.50	0.04	0.00	0.15	0.00	1.00	1.00	0.05	0.00	0.00	0.00	0.00	0.00	0.10
SimpleMethods	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.02	0.50	0.04	0.00	0.15	0.00	1.00	1.00	0.05	0.00	0.00	0.00	0.00	0.00	0.10
SingleResponsibility	0.00	0.00	0.00	0.08	0.00	0.20	0.00	0.00	0.00	0.00	0.67	0.00	0.00	0.00	0.26	1.00	0.21	0.00	0.02	0.00	0.08	0.08	1.00	0.00	0.00	0.00	0.00	0.00	0.95
SomewhatAbstract	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.04	0.00	0.02	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UniversalDataStructure	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
UniversalDatabaseMethod	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.23	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
UnrelatedAttributes	0.00	0.00	0.00	0.15	0.00	0.20	0.00	0.00	0.00	0.00	0.67	0.00	0.00	0.00	0.24	1.00	0.17	0.00	0.03	0.00	0.15	0.15	0.95	0.00	0.00	0.00	0.00	0.00	1.00

B.4 Tomcat

Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	2018	56	3%	1555	77%	340	17%	61	3%	6	0%	0	0%
DataClassFilter	2018	1118	55%	266	13%	396	20%	163	8%	58	3%	17	1%
DatabaseClass	2018	2014	100%	0	0%	4	0%	0	0%	0	0%	0	0%
DatabaseConnectorClass	2018	0	0%	2014	100%	0	0%	0	0%	0	0%	4	0%
DatabasePackage	99	0	0%	96	97%	0	0%	0	0%	0	0%	3	3%
LargeClass	2018	471	23%	1207	60%	268	13%	48	2%	21	1%	3	0%
LooselyCoupledRelations	2018	0	0%	857	42%	1029	51%	115	6%	15	1%	2	0%
NonDatabasePackageInvocation	129	0	0%	119	92%	0	0%	0	0%	10	8%	0	0%
RavioliCode	2018	0	0%	726	36%	1214	60%	78	4%	0	0%	0	0%
SingleResponsibility	2018	942	47%	621	31%	236	12%	138	7%	73	4%	8	0%
SimpleClass	1431	137	10%	589	41%	396	28%	202	14%	90	6%	17	1%
TheBlob	2018	477	24%	1201	60%	296	15%	43	2%	1	0%	0	0%
TowerOfVoodoo	2018	206	10%	1527	76%	257	13%	24	1%	4	0%	0	0%
UniversalDatabaseMethod	138	0	0%	19	14%	108	78%	11	8%	0	0%	0	0%
ConcreteDependentAbstraction	2018	1741	86%	235	12%	0	0%	32	2%	9	0%	1	0%
CyclicPackageDependency	99	0	0%	57	58%	0	0%	11	11%	4	4%	27	27%
ElementalDatabaseClass	1836	0	0%	1832	100%	0	0%	0	0%	4	0%	0	0%
InterfaceSegregation	2018	1869	93%	51	3%	0	0%	13	1%	16	1%	69	3%
InheritanceTargetScanner	592	0	0%	0	0%	303	51%	289	49%	0	0%	0	0%
LooseCoupling	1390	0	0%	1073	77%	168	12%	138	10%	9	1%	2	0%
ManyAttributes	2018	979	49%	958	47%	65	3%	10	0%	2	0%	4	0%
ManyDistinctRelations	2018	0	0%	1712	85%	99	5%	79	4%	65	3%	63	3%
ManyMethods	2018	471	23%	1194	59%	210	10%	91	5%	32	2%	20	1%
MethodUsageSetScanner	1076	0	0%	470	44%	0	0%	429	40%	155	14%	22	2%
PrimitiveAttributes	1294	0	0%	294	23%	394	30%	260	20%	250	19%	96	7%
SimpleMethods	1294	0	0%	1036	80%	0	0%	146	11%	58	4%	54	4%
SomewhatAbstract	2018	0	0%	0	0%	1681	83%	242	12%	47	2%	48	2%
UniversalDataStructure	12	0	0%	6	50%	1	8%	4	33%	1	8%	0	0%
UnrelatedAttributes	1076	469	44%	458	43%	72	7%	1	0%	74	7%	2	0%

p(Y|X)	LargeClass (24)	DataClassFilter (75)	SingleResponsibility (81)	CyclicPackageDependency (31)	ConcreteDependentAbstraction (10)	InterfaceSegregation (85)	DatabaseClass (0)	RavioliCode (0)	TheBlob (1)	TowerOfVoodoo (4)
LargeClass	1.00	0.04	0.17	0.00	0.10	0.01	0.00	0.00	1.00	0.00
DataClassFilter	0.13	1.00	0.11	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.58	0.12	1.00	0.00	0.50	0.00	0.00	0.00	1.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.04	0.00	0.06	0.00	1.00	0.01	0.00	0.00	0.00	0.00
InterfaceSegregation	0.04	0.00	0.00	0.00	0.10	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.04	0.00	0.01	0.00	0.00	0.00	0.00	0.00	1.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

p(Y|X)	**BlobRelations (6)**	**ConcreteDependentAbstraction (10)**	**CyclicPackageDependency (31)**	**DataClassFilter (75)**	**DatabaseClass (0)**	**DatabaseConnectorClass (4)**	**DatabasePackage (3)**	**ElementalDatabaseClass (4)**	**InheritanceTargetScanner (0)**	**InterfaceSegregation (85)**	**LargeClass (24)**	**LooseCoupling (11)**	**LooselyCoupledRelations (17)**	**ManyAttributes (6)**	**ManyDistinctRelations (128)**	**ManyMethods (52)**	**MethodUsageSetScanner (177)**	**NonDatabasePackageInvocation (10)**	**PrimitiveAttributes (346)**	**RavioliCode (0)**	**SimpleClass (107)**	**SimpleMethods (112)**	**SingleResponsibility (81)**	**SomewhatAbstract (95)**	**TheBlob (1)**	**TowerOfVoodoo (4)**	**UniversalDataStructure (1)**	**UniversalDatabaseMethod (0)**	**UnrelatedAttributes (76)**
BlobRelations	1.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.01
ConcreteDependentAbstraction	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.04	0.00	0.00	0.00	0.05	0.10	0.06	0.00	0.01	0.00	0.00	0.00	0.06	0.07	0.00	0.00	0.00	0.00	0.07
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.17	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.09	0.00	0.00	0.01	0.10	0.10	0.00	0.17	0.00	0.70	0.57	0.11	0.02	0.00	0.00	0.00	0.00	0.14
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.01	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.01
DatabasePackage	0.00	0.00	0.10	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.04	0.18	0.18	0.00	0.00	0.02	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00
LargeClass	0.17	0.10	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.01	1.00	0.00	0.00	1.00	0.09	0.40	0.12	0.00	0.03	0.00	0.03	0.03	0.17	0.12	1.00	0.00	0.00	0.00	0.12
LooseCoupling	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.02	0.00	1.00	0.06	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.02	0.03	0.00	0.03	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.09	1.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	1.00	0.02	0.06	0.02	0.00	0.01	0.00	0.00	0.02	0.04	0.01	0.00	0.00	0.00	0.00	0.03
ManyDistinctRelations	0.00	0.70	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.46	0.00	0.00	0.33	1.00	0.54	0.33	0.00	0.09	0.00	0.01	0.00	0.37	0.76	0.00	0.00	0.00	0.00	0.30
ManyMethods	0.00	0.50	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.01	0.88	0.00	0.00	0.50	0.22	1.00	0.28	0.00	0.07	0.00	0.06	0.04	0.36	0.22	1.00	0.00	0.00	0.00	0.28
MethodUsageSetScanner	0.00	1.00	0.00	0.23	0.00	1.00	0.00	0.00	0.00	0.15	0.88	0.09	0.06	0.67	0.46	0.94	1.00	0.00	0.19	0.00	0.16	0.15	0.65	0.53	1.00	0.00	0.00	0.00	0.54
NonDatabasePackageInvocation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PrimitiveAttributes	0.17	0.20	0.00	0.77	0.00	1.00	0.00	0.00	0.00	0.00	0.50	0.09	0.00	0.50	0.24	0.46	0.37	0.00	1.00	0.00	0.54	0.42	0.46	0.21	1.00	0.00	0.00	0.00	0.49
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.17	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.18	0.06	0.00	0.01	0.12	0.10	0.00	0.17	0.00	1.00	0.86	0.11	0.02	0.00	0.00	0.00	0.00	0.14
SimpleMethods	0.33	0.00	0.00	0.85	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.27	0.06	0.33	0.00	0.08	0.10	0.00	0.14	0.00	0.90	1.00	0.10	0.03	0.00	0.00	0.00	0.00	0.16
SingleResponsibility	0.00	0.50	0.00	0.12	0.00	0.25	0.00	0.00	0.00	0.00	0.58	0.00	0.00	0.50	0.23	0.56	0.30	0.00	0.11	0.00	0.08	0.07	1.00	0.23	1.00	0.00	0.00	0.00	0.91
SomewhatAbstract	0.00	0.70	0.00	0.03	0.00	0.00	0.00	0.00	0.00	0.02	0.46	0.27	0.00	0.17	0.56	0.40	0.28	0.00	0.06	0.00	0.02	0.03	0.27	1.00	0.00	0.00	0.00	0.00	0.20
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.01	0.00	1.00	0.00	0.00	0.00	0.01
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
UniversalDataStructure	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
UniversalDatabaseMethod	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UnrelatedAttributes	0.17	0.50	0.00	0.15	0.00	0.25	0.00	0.00	0.00	0.00	0.38	0.00	0.00	0.33	0.18	0.40	0.23	0.00	0.11	0.00	0.10	0.11	0.85	0.16	1.00	0.00	0.00	0.00	1.00

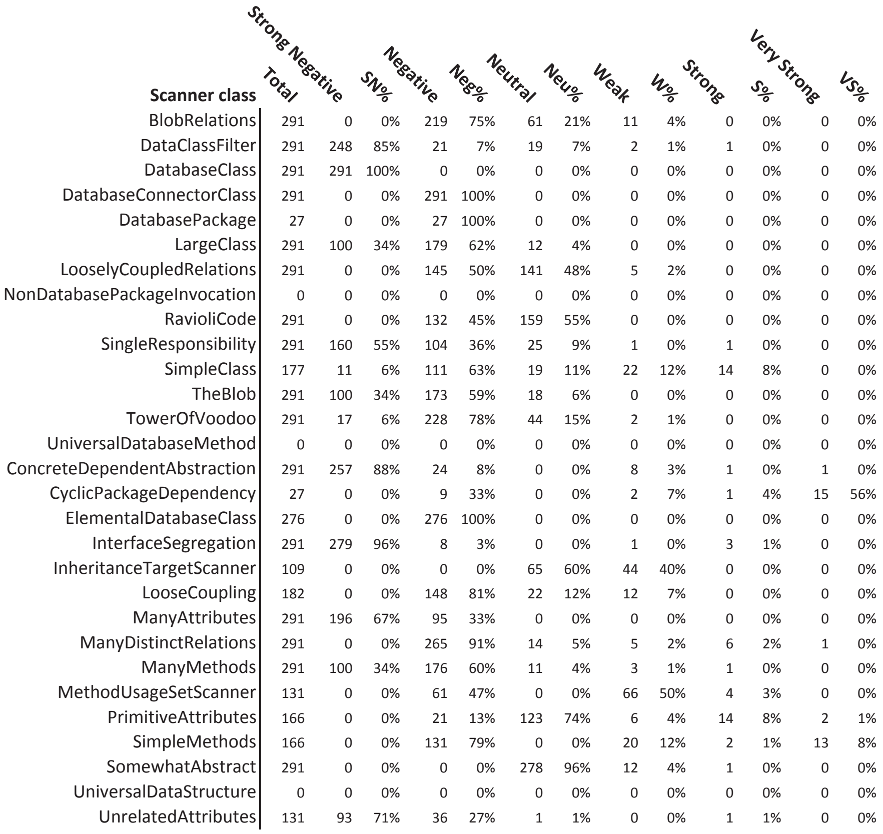
B.5 JForum

Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	1146	22	2%	904	79%	154	13%	65	6%	1	0%	0	0%
DataClassFilter	1146	915	80%	151	13%	43	4%	12	1%	21	2%	4	0%
DatabaseClass	1146	1142	100%	1	0%	2	0%	1	0%	0	0%	0	0%
DatabaseConnectorClass	1146	0	0%	1142	100%	0	0%	4	0%	0	0%	0	0%
DatabasePackage	35	0	0%	31	89%	0	0%	4	11%	0	0%	0	0%
LargeClass	1146	248	22%	841	73%	51	4%	4	0%	2	0%	0	0%
LooselyCoupledRelations	1146	0	0%	676	59%	470	41%	0	0%	0	0%	0	0%
NonDatabasePackageInvocation	88	0	0%	85	97%	0	0%	2	2%	0	0%	1	1%
RavioliCode	1146	0	0%	604	53%	541	47%	1	0%	0	0%	0	0%
SingleResponsibility	1146	726	63%	333	29%	44	4%	11	1%	30	3%	2	0%
SimpleClass	802	33	4%	662	83%	43	5%	18	2%	42	5%	4	0%
TheBlob	1146	270	24%	793	69%	69	6%	14	1%	0	0%	0	0%
TowerOfVoodoo	1146	54	5%	575	50%	517	45%	0	0%	0	0%	0	0%
UniversalDatabaseMethod	97	0	0%	54	56%	34	35%	8	8%	1	1%	0	0%
ConcreteDependentAbstraction	1146	1081	94%	42	4%	0	0%	20	2%	3	0%	0	0%
CyclicPackageDependency	35	0	0%	26	74%	0	0%	0	0%	1	3%	8	23%
ElementalDatabaseClass	989	0	0%	988	100%	0	0%	0	0%	1	0%	0	0%
InterfaceSegregation	1146	1114	97%	15	1%	0	0%	1	0%	9	1%	7	1%
InheritanceTargetScanner	600	0	0%	0	0%	66	11%	534	89%	0	0%	0	0%
LooseCoupling	766	0	0%	639	83%	90	12%	37	5%	0	0%	0	0%
ManyAttributes	1146	825	72%	313	27%	6	1%	1	0%	0	0%	1	0%
ManyDistinctRelations	1146	0	0%	977	85%	78	7%	52	5%	19	2%	20	2%
ManyMethods	1146	248	22%	835	73%	50	4%	8	1%	4	0%	1	0%
MethodUsageSetScanner	420	0	0%	305	73%	0	0%	102	24%	11	3%	2	0%
PrimitiveAttributes	769	0	0%	167	22%	538	70%	35	5%	27	4%	2	0%
SimpleMethods	769	0	0%	690	90%	0	0%	32	4%	19	2%	28	4%
SomewhatAbstract	1146	0	0%	0	0%	1060	92%	84	7%	2	0%	0	0%
UniversalDataStructure	34	0	0%	29	85%	1	3%	2	6%	2	6%	0	0%
UnrelatedAttributes	420	193	46%	181	43%	8	2%	3	1%	35	8%	0	0%

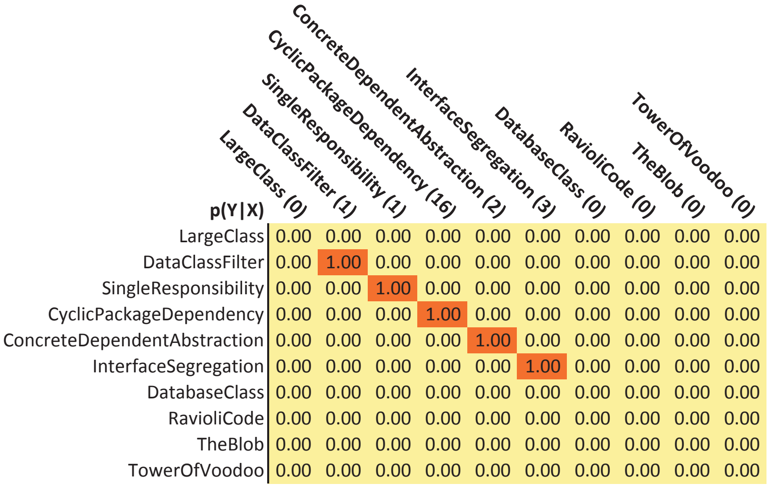
p(Y|X)	LargeClass (2)	DataClassFilter (25)	SingleResponsibility (32)	CyclicPackageDependency (9)	ConcreteDependentAbstraction (3)	InterfaceSegregation (16)	DatabaseClass (0)	RavioliCode (0)	TheBlob (0)	TowerOfVoodoo (0)
LargeClass	1.00	0.04	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.50	1.00	0.41	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.50	0.52	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.00	0.00	1.00	0.06	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.33	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

p(Y|X)	BlobRelations (1)	ConcreteDependentAbstraction (3)	CyclicPackageDependency (9)	DataClassFilter (25)	DatabaseClass (0)	DatabaseConnectorClass (0)	DatabasePackage (0)	ElementalDatabaseClass (1)	InheritanceTargetScanner (0)	InterfaceSegregation (16)	LargeClass (2)	LooseCoupling (0)	LooselyCoupledRelations (0)	ManyAttributes (1)	ManyDistinctRelations (39)	ManyMethods (5)	MethodUsageSetScanner (13)	NonDatabasePackageInvocation (1)	PrimitiveAttributes (29)	RavioliCode (0)	SimpleClass (46)	SimpleMethods (47)	SingleResponsibility (32)	SomewhatAbstract (2)	TheBlob (0)	TowerOfVoodoo (0)	UniversalDataStructure (2)	UniversalDatabaseMethod (1)	UnrelatedAttributes (35)
BlobRelations	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.40	0.38	0.00	0.48	0.00	0.54	0.53	0.41	0.00	0.00	0.00	0.00	0.00	0.43
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabasePackage	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LargeClass	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.00	0.20	0.08	0.00	0.03	0.00	0.02	0.02	0.03	0.00	0.00	0.00	0.00	0.00	0.03
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyDistinctRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
ManyMethods	0.00	0.00	0.00	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	1.00	0.15	0.00	0.10	0.00	0.04	0.04	0.09	0.00	0.00	0.00	0.00	0.00	0.09
MethodUsageSetScanner	0.00	0.33	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.06	0.50	0.00	0.00	0.00	0.00	0.40	1.00	0.00	0.17	0.00	0.11	0.11	0.28	0.00	0.00	0.00	0.00	0.00	0.23
NonDatabasePackageInvocation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
PrimitiveAttributes	0.00	0.00	0.00	0.56	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.60	0.38	0.00	1.00	0.00	0.30	0.30	0.47	0.00	0.00	0.00	0.00	0.00	0.49
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.40	0.38	0.00	0.48	0.00	1.00	0.98	0.41	0.00	0.00	0.00	0.00	0.00	0.43
SimpleMethods	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.40	0.38	0.00	0.48	0.00	1.00	1.00	0.41	0.00	0.00	0.00	0.00	0.00	0.43
SingleResponsibility	0.00	0.00	0.00	0.52	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.60	0.69	0.00	0.52	0.00	0.28	0.28	1.00	0.00	0.00	0.00	0.00	0.00	0.89
SomewhatAbstract	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UniversalDataStructure	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
UniversalDatabaseMethod	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
UnrelatedAttributes	0.00	0.00	0.00	0.60	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.60	0.62	0.00	0.59	0.00	0.33	0.32	0.97	0.00	0.00	0.00	0.00	0.00	1.00

B.6 JUnit



Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	291	0	0%	219	75%	61	21%	11	4%	0	0%	0	0%
DataClassFilter	291	248	85%	21	7%	19	7%	2	1%	1	0%	0	0%
DatabaseClass	291	291	100%	0	0%	0	0%	0	0%	0	0%	0	0%
DatabaseConnectorClass	291	0	0%	291	100%	0	0%	0	0%	0	0%	0	0%
DatabasePackage	27	0	0%	27	100%	0	0%	0	0%	0	0%	0	0%
LargeClass	291	100	34%	179	62%	12	4%	0	0%	0	0%	0	0%
LooselyCoupledRelations	291	0	0%	145	50%	141	48%	5	2%	0	0%	0	0%
NonDatabasePackageInvocation	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
RavioliCode	291	0	0%	132	45%	159	55%	0	0%	0	0%	0	0%
SingleResponsibility	291	160	55%	104	36%	25	9%	1	0%	1	0%	0	0%
SimpleClass	177	11	6%	111	63%	19	11%	22	12%	14	8%	0	0%
TheBlob	291	100	34%	173	59%	18	6%	0	0%	0	0%	0	0%
TowerOfVoodoo	291	17	6%	228	78%	44	15%	2	1%	0	0%	0	0%
UniversalDatabaseMethod	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
ConcreteDependentAbstraction	291	257	88%	24	8%	0	0%	8	3%	1	0%	1	0%
CyclicPackageDependency	27	0	0%	9	33%	0	0%	2	7%	1	4%	15	56%
ElementalDatabaseClass	276	0	0%	276	100%	0	0%	0	0%	0	0%	0	0%
InterfaceSegregation	291	279	96%	8	3%	0	0%	1	0%	3	1%	0	0%
InheritanceTargetScanner	109	0	0%	0	0%	65	60%	44	40%	0	0%	0	0%
LooseCoupling	182	0	0%	148	81%	22	12%	12	7%	0	0%	0	0%
ManyAttributes	291	196	67%	95	33%	0	0%	0	0%	0	0%	0	0%
ManyDistinctRelations	291	0	0%	265	91%	14	5%	5	2%	6	2%	1	0%
ManyMethods	291	100	34%	176	60%	11	4%	3	1%	1	0%	0	0%
MethodUsageSetScanner	131	0	0%	61	47%	0	0%	66	50%	4	3%	0	0%
PrimitiveAttributes	166	0	0%	21	13%	123	74%	6	4%	14	8%	2	1%
SimpleMethods	166	0	0%	131	79%	0	0%	20	12%	2	1%	13	8%
SomewhatAbstract	291	0	0%	0	0%	278	96%	12	4%	1	0%	0	0%
UniversalDataStructure	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
UnrelatedAttributes	131	93	71%	36	27%	1	1%	0	0%	1	1%	0	0%



p(Y|X)	LargeClass (0)	DataClassFilter (1)	SingleResponsibility (1)	CyclicPackageDependency (16)	ConcreteDependentAbstraction (2)	InterfaceSegregation (3)	DatabaseClass (0)	RavioliCode (0)	TheBlob (0)	TowerOfVoodoo (0)
LargeClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

p(Y|X)	**BlobRelations (0)**	**ConcreteDependentAbstraction (2)**	**CyclicPackageDependency (16)**	**DataClassFilter (1)**	**DatabaseClass (0)**	**DatabaseConnectorClass (0)**	**DatabasePackage (0)**	**ElementalDatabaseClass (0)**	**InheritanceTargetScanner (0)**	**InterfaceSegregation (3)**	**LargeClass (0)**	**LooseCoupling (0)**	**LooselyCoupledRelations (0)**	**ManyAttributes (0)**	**ManyDistinctRelations (7)**	**ManyMethods (1)**	**MethodUsageSetScanner (4)**	**PrimitiveAttributes (16)**	**RavioliCode (0)**	**SimpleClass (14)**	**SimpleMethods (15)**	**SingleResponsibility (1)**	**SomewhatAbstract (1)**	**TheBlob (0)**	**TowerOfVoodoo (0)**	**UnrelatedAttributes (1)**
BlobRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabasePackage	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LargeClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyDistinctRelations	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
ManyMethods	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
MethodUsageSetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PrimitiveAttributes	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.07	0.00	1.00	0.00	0.00	0.00	1.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	1.00	0.87	0.00	0.00	0.00	0.00	0.00
SimpleMethods	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.93	1.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00
SomewhatAbstract	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UnrelatedAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.00	0.00	1.00	0.00	0.00	0.00	1.00

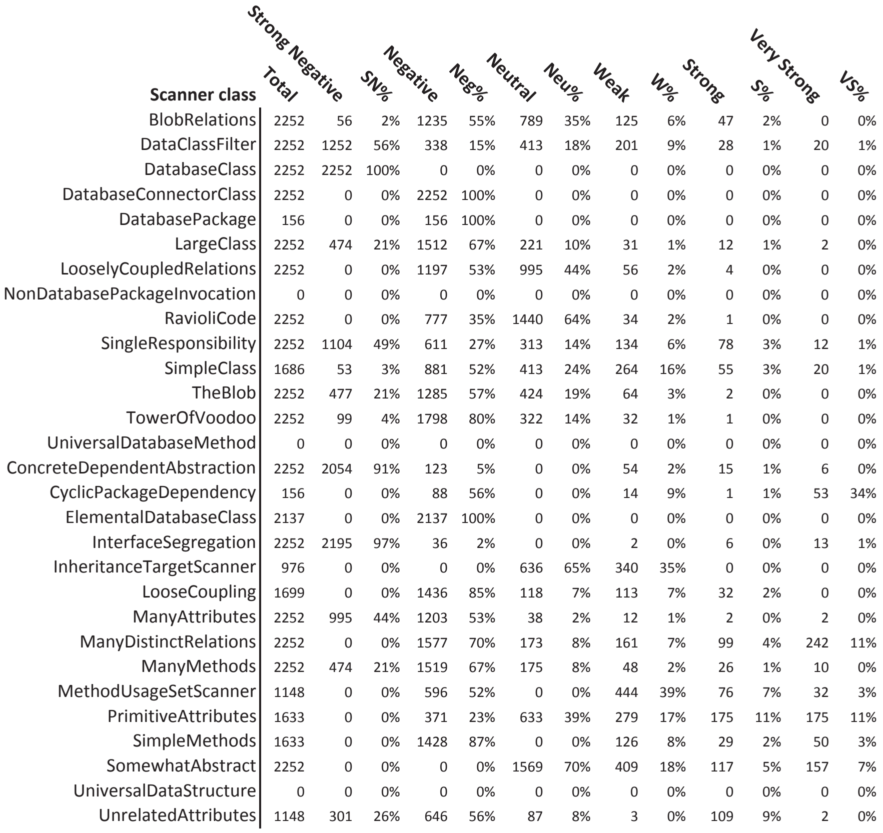
B.7 JHotDraw

Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	1558	62	4%	1382	89%	113	7%	1	0%	0	0%	0	0%
DataClassFilter	1558	1101	71%	246	16%	177	11%	26	2%	6	0%	2	0%
DatabaseClass	1558	1558	100%	0	0%	0	0%	0	0%	0	0%	0	0%
DatabaseConnectorClass	1558	0	0%	1558	100%	0	0%	0	0%	0	0%	0	0%
DatabasePackage	64	0	0%	64	100%	0	0%	0	0%	0	0%	0	0%
LargeClass	1558	501	32%	906	58%	137	9%	12	1%	2	0%	0	0%
LooselyCoupledRelations	1558	0	0%	565	36%	911	58%	66	4%	11	1%	5	0%
NonDatabasePackageInvocation	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
RavioliCode	1558	0	0%	466	30%	1037	67%	54	3%	1	0%	0	0%
SingleResponsibility	1558	947	61%	329	21%	139	9%	82	5%	60	4%	1	0%
SimpleClass	1012	57	6%	715	71%	177	17%	51	5%	10	1%	2	0%
TheBlob	1558	510	33%	896	58%	148	9%	4	0%	0	0%	0	0%
TowerOfVoodoo	1558	110	7%	1032	66%	373	24%	43	3%	0	0%	0	0%
UniversalDatabaseMethod	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
ConcreteDependentAbstraction	1558	1403	90%	119	8%	0	0%	28	2%	8	1%	0	0%
CyclicPackageDependency	64	0	0%	27	42%	0	0%	4	6%	6	9%	27	42%
ElementalDatabaseClass	1441	0	0%	1441	100%	0	0%	0	0%	0	0%	0	0%
InterfaceSegregation	1558	1501	96%	15	1%	0	0%	1	0%	8	1%	33	2%
InheritanceTargetScanner	630	0	0%	0	0%	189	30%	441	70%	0	0%	0	0%
LooseCoupling	1003	0	0%	827	82%	81	8%	83	8%	10	1%	2	0%
ManyAttributes	1558	1003	64%	546	35%	6	0%	2	0%	0	0%	1	0%
ManyDistinctRelations	1558	0	0%	1303	84%	94	6%	71	5%	50	3%	40	3%
ManyMethods	1558	501	32%	894	57%	124	8%	27	2%	11	1%	1	0%
MethodUsageSetScanner	611	0	0%	242	40%	0	0%	295	48%	65	11%	9	1%
PrimitiveAttributes	955	0	0%	257	27%	498	52%	123	13%	56	6%	21	2%
SimpleMethods	955	0	0%	902	94%	0	0%	40	4%	5	1%	8	1%
SomewhatAbstract	1558	0	0%	0	0%	1237	79%	261	17%	48	3%	12	1%
UniversalDataStructure	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
UnrelatedAttributes	611	246	40%	237	39%	65	11%	1	0%	61	10%	1	0%

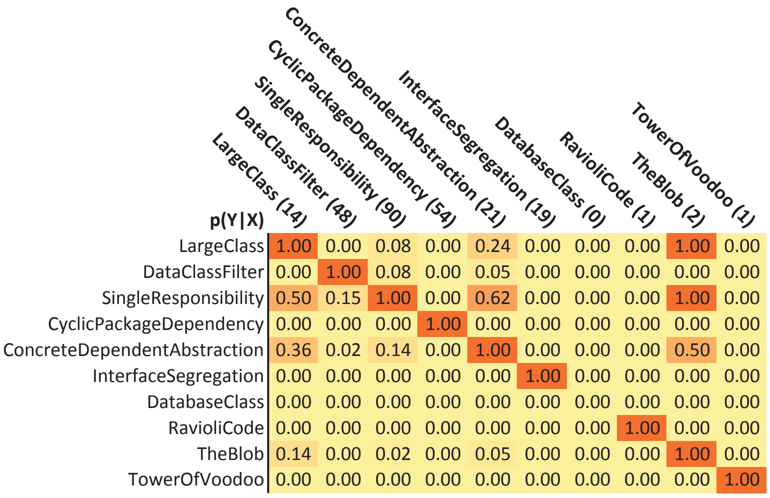
p(Y|X)	LargeClass (2)	DataClassFilter (8)	SingleResponsibility (61)	CyclicPackageDependency (33)	ConcreteDependentAbstraction (8)	InterfaceSegregation (41)	DatabaseClass (0)	RavioliCode (1)	TheBlob (0)	TowerOfVoodoo (0)
LargeClass	1.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.50	0.00	1.00	0.00	0.13	0.00	0.00	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	0.00	0.02	0.00	1.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

p(Y|X)	BlobRelations (0)	ConcreteDependentAbstraction (8)	CyclicPackageDependency (33)	DataClassFilter (8)	DatabaseClass (0)	DatabaseConnectorClass (0)	DatabasePackage (0)	ElementalDatabaseClass (0)	InheritanceTargetScanner (0)	InterfaceSegregation (41)	LargeClass (2)	LooseCoupling (12)	LooselyCoupledRelations (16)	ManyAttributes (1)	ManyDistinctRelations (90)	ManyMethods (12)	MethodUsageSetScanner (74)	PrimitiveAttributes (77)	RavioliCode (1)	SimpleClass (12)	SimpleMethods (13)	SingleResponsibility (61)	SomewhatAbstract (60)	TheBlob (0)	TowerOfVoodoo (0)	UnrelatedAttributes (62)
BlobRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.17	0.04	0.01	0.00	0.00	0.00	0.02	0.07	0.00	0.00	0.00
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	0.67	0.46	0.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabasePackage	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.25	0.06	0.00	0.00	0.17	0.12	0.00	0.00	0.00	0.00	0.00	0.03	0.00	0.00	0.00
LargeClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.01	0.08	0.01	0.00	0.00	0.00	0.00	0.02	0.02	0.00	0.00	0.02
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.07	0.00	1.00	0.06	0.00	0.00	0.08	0.03	0.00	0.00	0.00	0.00	0.00	0.05	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.00	0.08	1.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ManyDistinctRelations	0.00	0.63	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	1.00	0.67	0.31	0.03	1.00	0.00	0.00	0.43	0.63	0.00	0.00	0.40
ManyMethods	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.50	0.08	0.00	0.00	0.09	1.00	0.12	0.01	0.00	0.00	0.00	0.11	0.15	0.00	0.00	0.08
MethodUsageSetScanner	0.00	0.38	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.22	0.50	0.17	0.00	0.00	0.26	0.75	1.00	0.05	1.00	0.00	0.00	0.36	0.37	0.00	0.00	0.27
PrimitiveAttributes	0.00	0.13	0.00	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.06	0.00	0.02	0.08	0.05	1.00	0.00	0.50	0.31	0.05	0.10	0.00	0.00	0.08
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.01	0.00	1.00	0.00	0.00	0.00	0.02	0.00	0.00	0.00
SimpleClass	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.08	0.00	1.00	0.77	0.00	0.00	0.00	0.00	0.00
SimpleMethods	0.00	0.00	0.00	0.75	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.00	0.83	1.00	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.00	0.13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.29	0.58	0.30	0.04	0.00	0.00	0.00	1.00	0.28	0.00	0.00	0.90
SomewhatAbstract	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.50	0.25	0.00	0.00	0.42	0.75	0.30	0.08	1.00	0.00	0.00	0.28	1.00	0.00	0.00	0.23
TheBlob	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
UnrelatedAttributes	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.28	0.42	0.23	0.06	0.00	0.00	0.00	0.92	0.23	0.00	0.00	1.00

B.8 JMonkeyEngine



Scanner class	Total	Strong Negative	SN%	Negative	Neg%	Neutral	Neu%	Weak	W%	Strong	S%	Very Strong	VS%
BlobRelations	2252	56	2%	1235	55%	789	35%	125	6%	47	2%	0	0%
DataClassFilter	2252	1252	56%	338	15%	413	18%	201	9%	28	1%	20	1%
DatabaseClass	2252	2252	100%	0	0%	0	0%	0	0%	0	0%	0	0%
DatabaseConnectorClass	2252	0	0%	2252	100%	0	0%	0	0%	0	0%	0	0%
DatabasePackage	156	0	0%	156	100%	0	0%	0	0%	0	0%	0	0%
LargeClass	2252	474	21%	1512	67%	221	10%	31	1%	12	1%	2	0%
LooselyCoupledRelations	2252	0	0%	1197	53%	995	44%	56	2%	4	0%	0	0%
NonDatabasePackageInvocation	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
RavioliCode	2252	0	0%	777	35%	1440	64%	34	2%	1	0%	0	0%
SingleResponsibility	2252	1104	49%	611	27%	313	14%	134	6%	78	3%	12	1%
SimpleClass	1686	53	3%	881	52%	413	24%	264	16%	55	3%	20	1%
TheBlob	2252	477	21%	1285	57%	424	19%	64	3%	2	0%	0	0%
TowerOfVoodoo	2252	99	4%	1798	80%	322	14%	32	1%	1	0%	0	0%
UniversalDatabaseMethod	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
ConcreteDependentAbstraction	2252	2054	91%	123	5%	0	0%	54	2%	15	1%	6	0%
CyclicPackageDependency	156	0	0%	88	56%	0	0%	14	9%	1	1%	53	34%
ElementalDatabaseClass	2137	0	0%	2137	100%	0	0%	0	0%	0	0%	0	0%
InterfaceSegregation	2252	2195	97%	36	2%	0	0%	2	0%	6	0%	13	1%
InheritanceTargetScanner	976	0	0%	0	0%	636	65%	340	35%	0	0%	0	0%
LooseCoupling	1699	0	0%	1436	85%	118	7%	113	7%	32	2%	0	0%
ManyAttributes	2252	995	44%	1203	53%	38	2%	12	1%	2	0%	2	0%
ManyDistinctRelations	2252	0	0%	1577	70%	173	8%	161	7%	99	4%	242	11%
ManyMethods	2252	474	21%	1519	67%	175	8%	48	2%	26	1%	10	0%
MethodUsageSetScanner	1148	0	0%	596	52%	0	0%	444	39%	76	7%	32	3%
PrimitiveAttributes	1633	0	0%	371	23%	633	39%	279	17%	175	11%	175	11%
SimpleMethods	1633	0	0%	1428	87%	0	0%	126	8%	29	2%	50	3%
SomewhatAbstract	2252	0	0%	0	0%	1569	70%	409	18%	117	5%	157	7%
UniversalDataStructure	0	0	0%	0	0%	0	0%	0	0%	0	0%	0	0%
UnrelatedAttributes	1148	301	26%	646	56%	87	8%	3	0%	109	9%	2	0%



p(Y|X)	LargeClass (14)	DataClassFilter (48)	SingleResponsibility (90)	CyclicPackageDependency (54)	ConcreteDependentAbstraction (21)	InterfaceSegregation (19)	DatabaseClass (0)	RavioliCode (1)	TheBlob (2)	TowerOfVoodoo (1)
LargeClass	1.00	0.00	0.08	0.00	0.24	0.00	0.00	0.00	1.00	0.00
DataClassFilter	0.00	1.00	0.08	0.00	0.05	0.00	0.00	0.00	0.00	0.00
SingleResponsibility	0.50	0.15	1.00	0.00	0.62	0.00	0.00	0.00	1.00	0.00
CyclicPackageDependency	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.36	0.02	0.14	0.00	1.00	0.00	0.00	0.00	0.50	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
TheBlob	0.14	0.00	0.02	0.00	0.05	0.00	0.00	0.00	1.00	0.00
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00

p(Y|X)	**BlobRelations (47)**	**ConcreteDependentAbstraction (21)**	**CyclicPackageDependency (54)**	**DataClassFilter (48)**	**DatabaseClass (0)**	**DatabaseConnectorClass (0)**	**DatabasePackage (0)**	**ElementalDatabaseClass (0)**	**InheritanceTargetScanner (0)**	**InterfaceSegregation (19)**	**LargeClass (14)**	**LooseCoupling (32)**	**LooselyCoupledRelations (4)**	**ManyAttributes (4)**	**ManyDistinctRelations (341)**	**ManyMethods (36)**	**MethodUsageSetScanner (108)**	**PrimitiveAttributes (350)**	**RavioliCode (1)**	**SimpleClass (75)**	**SimpleMethods (79)**	**SingleResponsibility (90)**	**SomewhatAbstract (274)**	**TheBlob (2)**	**TowerOfVoodoo (1)**	**UnrelatedAttributes (111)**
BlobRelations	1.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.00	0.23	0.23	0.00	0.00	0.00	0.00	0.00
ConcreteDependentAbstraction	0.00	1.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.00	0.00	0.25	0.04	0.31	0.11	0.01	0.00	0.01	0.01	0.14	0.05	0.50	0.00	0.09
CyclicPackageDependency	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DataClassFilter	0.04	0.05	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.03	0.02	0.10	0.00	0.64	0.57	0.08	0.00	0.00	0.00	0.06
DatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabaseConnectorClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
DatabasePackage	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
ElementalDatabaseClass	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InheritanceTargetScanner	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
InterfaceSegregation	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.06	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LargeClass	0.00	0.24	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	1.00	0.02	0.33	0.11	0.01	0.00	0.00	0.00	0.08	0.02	1.00	0.00	0.05
LooseCoupling	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.01	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
LooselyCoupledRelations	0.00	0.00	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.01	1.00	0.01	0.01	0.00	0.00	0.00	0.00	0.00
ManyAttributes	0.00	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.00	0.00	1.00	0.01	0.06	0.03	0.00	0.00	0.00	0.00	0.02	0.01	1.00	0.00	0.02
ManyDistinctRelations	0.00	0.67	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.25	0.50	1.00	0.47	0.37	0.14	1.00	0.01	0.01	0.52	0.94	1.00	0.00	0.55
ManyMethods	0.00	0.52	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.11	0.86	0.00	0.00	0.50	0.05	1.00	0.30	0.03	0.00	0.01	0.01	0.20	0.06	1.00	0.00	0.11
MethodUsageSetScanner	0.00	0.57	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.11	0.86	0.03	0.00	0.75	0.12	0.89	1.00	0.09	0.00	0.03	0.03	0.52	0.11	1.00	0.00	0.29
PrimitiveAttributes	0.34	0.19	0.00	0.71	0.00	0.00	0.00	0.00	0.00	0.00	0.29	0.22	0.50	0.25	0.14	0.31	0.30	1.00	0.00	0.45	0.39	0.27	0.12	0.50	0.00	0.28
RavioliCode	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
SimpleClass	0.36	0.05	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.03	0.02	0.10	0.00	1.00	0.91	0.08	0.00	0.00	0.00	0.06
SimpleMethods	0.38	0.05	0.00	0.94	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.00	0.00	0.03	0.02	0.09	0.00	0.96	1.00	0.08	0.00	0.00	0.00	0.06
SingleResponsibility	0.00	0.62	0.00	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.50	0.14	0.50	0.44	0.07	0.00	0.09	0.09	1.00	0.14	1.00	0.00	0.68
SomewhatAbstract	0.00	0.67	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.43	0.03	0.00	0.50	0.75	0.47	0.29	0.10	0.00	0.01	0.01	0.43	1.00	1.00	0.00	0.48
TheBlob	0.00	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.50	0.01	0.06	0.02	0.00	0.00	0.00	0.00	0.02	0.01	1.00	0.00	0.02
TowerOfVoodoo	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00
UnrelatedAttributes	0.00	0.48	0.00	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.43	0.00	0.00	0.50	0.18	0.33	0.30	0.09	0.00	0.09	0.09	0.83	0.19	1.00	0.00	1.00