

Smell Detection in Context

Diplomarbeit

Sebastian Jancke

Gemarkenstr. 100 – 45147 Essen
sebastian.jancke@googlemail.com

Bonn, den 20. März 2010

Sebastian Jancke: *Smell Detection in Context*, Diplom Informatik, © March 2010

SUPERVISORS:

Prof. Dr. A. B. Cremers

Prof. Dr. R. Manthey

Daniel Speicher

LOCATION:

Bonn

One of the easiest way to make a piece of code readable is to have someone read it.

— Venkat Subramaniam (on Twitter)

Dedicated to Miriam.

I'd like to thank Jan, Uli, Christian, Peter, Andreas, Günter and Daniel.

This thesis investigates the use of context sensitivity in smell detection to support very frequent refactoring. Frequent refactoring needs very easy access to relevant smells. So far, it has not been studied, how context sensitivity can improve smell detection to support frequent refactoring. Further, current approaches do not exploit the impact of structures on false positives during smell detection. The approach “Smell Detection in Context” provides concepts for three contexts, modeling the impact of structures, the relevance of smells and the relationship between smells. The usefulness and applicability are investigated with the case study research method. The case study of a typical industrial project showed how developers leverage context sensitivity to help them in their process of frequent refactoring.

The detailed analysis of various structures revealed that structures have a significant impact on smells. This impact has to be regarded during smell detection. Taking this information into account, the structural context avoids false positives. The huge amount of smells are made manageable with the help of relevance filtering and an appropriate relevance context. The modeling of smell details and relationships between smells makes smells and their complex detection rules more understandable for the user. Manual investigation of these details and relations is automated and improves the access to them.

CONTENTS

1	INTRODUCTION	15
2	REVIEW OF THE STATE OF THE ART	17
2.1	Refactoring Opportunities: Smells, Design Flaws, Antipatterns	17
2.2	Analysis based on Structural Detection	18
2.2.1	Logic Meta-Programming Framework for Supporting the Refactoring Process	18
2.2.2	Java Quality Assurance by Detecting Code Smells	19
2.2.3	Code Smell Detection in Eclipse	19
2.3	Analysis based on Metrics	20
2.3.1	Distance Based Cohesion	20
2.3.2	Detection Strategies	20
2.3.3	Using History in Detection Strategies	22
2.4	Usability and Efficient Smell Detection	23
2.5	Tools	24
3	DESIGN FLAWS	29
3.1	Godclass	30
3.2	Dataclass	31
3.3	Feature Envy	32
3.4	Brainclass	33
3.5	Brainmethod	33
3.6	Middle Man	34
3.7	Iceberg Class	35
3.8	Law of Demeter Violation	36
3.9	Shotgun Surgery	37
3.10	Intensive Coupling	38
3.11	Dispersed Coupling	39
3.12	Refused Parent Bequest	39
3.13	Tradition Breaker	40
3.14	Summary	41
4	RESEARCH QUESTION	43

4.1	Remaining problems in the related work	43
4.2	Hypotheses	45
5	RELATION OF STRUCTURES, SMELLS AND FALSE POSITIVES	47
5.1	Design Patterns	47
5.1.1	Decorator	47
5.1.2	Composite	48
5.1.3	Visitor	49
5.1.4	Strategy	49
5.1.5	Facade	49
5.1.6	Adapter	50
5.1.7	Mediator	50
5.1.8	Creation Methods	51
5.2	Procedural Components	51
5.3	Embedded Domain Specific Languages	52
5.3.1	Method Chaining	52
5.3.2	Nested functions	55
5.3.3	Examining the use of Google Guice 2.0	56
5.4	Application Programming Interface Design	57
5.5	Presentation Layers	59
5.6	Unittests	60
5.7	Summary of the Structure-Smell-Tradeoff	61
6	APPROACH: SMELL DETECTION IN CONTEXT	63
6.1	Improving State of the Art in Cultivate	63
6.1.1	Architecture of Cultivate	63
6.1.2	Architecture of Smell Detection in Context	67
6.1.3	Improving Detection Strategies	67
6.2	Structure Context to reduce False Positives	68
6.2.1	Friend Relations - A Model for Structures and Smells	69
6.2.2	Providing Instances of Structures	70
6.2.3	Case: Visitor Pattern in JRefactory 2.6.24	72
6.2.4	Case: Decorator Pattern in JHotDraw 6	74
6.2.5	Case: Strategy Pattern in JHotDraw 6	75
6.2.6	Case: Public Api in Cultivate Core, Revision 2264	76
6.3	Relevance Context: Task Relevance	77
6.3.1	Task Relevance	77
6.3.2	Integrating Cultivate with Eclipse Mylyn	79

6.4	Relevance Context: Temporal Relevance	81
6.4.1	Temporal Relevance	81
6.4.2	Computing Temporal Degree of Interest	83
6.4.3	Case	85
6.5	Interlinking Context	86
6.5.1	Interlinking of Design Flaws	86
6.5.2	Examples: Feature Envy and Shotgun Surgery	87
6.5.3	Implementation of Smell Relations and Details	88
6.5.4	Implementation of the Smell Context View	92
7	VALIDATION OF THE APPROACH	97
7.1	The Case Study Method	97
7.2	Case Study Design	98
7.2.1	Research Questions	98
7.2.2	Case and Subjects Selection	99
7.2.3	Data Collection Procedures	101
7.2.4	Analysis Procedures	102
7.2.5	Validity Procedures	104
7.3	Context: SOPTIM ProSys	105
7.4	Cross Case Analysis	106
7.5	Threads to Validity	108
7.6	Limitations of this Case Study	109
8	CONCLUSIONS	111
8.1	Conclusions	111
8.2	Summary of Contributions	112
8.3	Future Research	113
A	APPENDIX	115
A.1	Metric Definitions	115
A.2	Observation Guide	117
A.3	Interview Guide	118
A.4	Conducted Data	120

LIST OF FIGURES

Figure 2.1	Checkstyle "Violations Chart" giving an overview of the violations shares	25
Figure 2.2	CodeNose	25
Figure 2.3	JDeodorant showing instances of feature envy in a special view.	26
Figure 2.4	Smell detection within the Refactoring Browser of Visual-Works Smalltalk	27
Figure 3.1	Feature Envy	32
Figure 3.2	Middle Man Smell	34
Figure 3.3	Iceberg Class	35
Figure 3.4	Shotgun Surgery	37
Figure 3.5	Intensive Coupling	38
Figure 3.6	Dispersed Coupling	39
Figure 5.1	Classdiagram with dependencies in the sample 5.1	54
Figure 5.2	Classdiagram of dependencies in sample 5.4	58
Figure 6.1	Architecture of Cultivate. Green boxes are Cultivate Addons, blue boxes are components written in Java, red boxes are written in Prolog.	64
Figure 6.2	Result View of Cultivate, showing Godclass Smells	65
Figure 6.3	Cultivate using Eclipse Markers to indicate a Godclass	66
Figure 6.4	Architecture of Smell Detection in Context	68
Figure 6.5	Model of Structures	69
Figure 6.6	Extract of the Google Guice DSL with Structural Annotations	72
Figure 6.7	SummaryVisitor in JRefactory 2.6.24	73
Figure 6.8	DecoratorFigure in JHotdraw	74
Figure 6.9	Strategy pattern in JHotdraw	76
Figure 6.10	Focused Eclipse Package Explorer using Eclipse Mylyn	79
Figure 6.11	Focused Cultivate Result View	80
Figure 6.12	Architecture of Cultivate Histogram Component	83
Figure 6.13	Change Histogram of class ResultViewPart	85
Figure 6.14	Change Histogram of class AbstractDiagramConfiguration	86

Figure 6.15	Smell Context View showing Feature Envy in JHotdraw	93
Figure 6.16	Smell Context View showing First Oder Details of a Godclass in JHotdraw	94
Figure 6.17	Smell Context View showing Shotgun Surgery and Second Order Details	94
Figure 7.1	Godclass smell listed in the result view of the classic prototype	99
Figure 7.2	Elements in the current Eclipse Mylyn task “Working on holiday feature”	100
Figure 7.3	Smells relevant to the current task listed in the SDIC prototype	101

LIST OF TABLES

Table 2.1	Statistical thresholds based on 45 Java projects [LM06, page 16]	21
Table 5.1	Overview of Structures and Smells	62
Table 6.1	Smell Relations and Details	89
Table 7.1	Features of Prototypes in both Cases	100
Table 7.2	Linking Data To Propositions	103
Table A.1	Interviews Before the Case Study with Developer A	120
Table A.2	Interviews Before the Case Study with Developer B	120
Table A.3	Observations during Case: ProSys + Classic	120
Table A.4	Interviews After Case: ProSys + Classic	121
Table A.5	Observations during Case: ProSys + SDIC	121
Table A.6	Interviews After Case: ProSys + SDIC	122
Table A.7	Late Interviews after the case study, two weeks after the study	122

INTRODUCTION

Maintaining design quality is one of the most difficult problems in software engineering. Several factors contribute to this. Among those are the intrinsic complexity of the modeled domains, an additional incidental complexity and the complexity of human communication in larger organizations [LM06]. Further, systems rot and age under the load of frequent requirement and technology changes. Maintaining and raising their design quality is mandatory to keep the design alive and the system successful:

Software must evolve and change to be successful [...] A design often degrades and gets more and more complex and thus harder to evolve [...] a design is not written in stone but must be revised and improved over and over again to fight against the effects of aging and decay that software systems inevitably incur. [LM06, page 2]

Frequently refactoring a software is a method to maintain and raise its design quality. Floss refactoring (see 2.4) is a common approach to frequent refactoring [MHB08a]. To foster frequent refactorings, efficient smell detection is needed. Such efficient smell detection should integrate well into floss refactorings and provide sufficient details to help the user understand a design problem.

This raises the question, how smell detection during floss refactorings can be improved. The core hypothesis of this thesis is, that it can be improved essentially with the usage of contexts. The diploma thesis is structured as follows. First, the current state of the art in smell detection is reviewed in chapter 2. The studied and implemented design flaws are reviewed in chapter 3. Chapter 4 investigates unsolved problems in the current state of research and gives a detailed statement of the research question and hypotheses.

The impact of structures on the existence of smells is investigated in chapter 5. The results are used in the next chapter. Chapter 6 explains the context sensitive approach to smell detection, called “Smell Detection in Context”. Three contexts are constructed to enhance smell detection: a structural context, a relevance context and an interlinking context. The approach makes use of the before investigated impact in the structural context. The usability and applicability of the approach is validated in chapter 7. The validation is done with a case study of an industrial project. Finally, in

chapter 8 conclusions are drawn and contributions and future work are summarized. Implementation details of the metrics and smell detectors, as well as details of the case study are given in an appendix.

The diploma thesis shows, that structures have an significant impact on the existence of smells. The concepts of friend relations and friend smells allow to model these effects. These structural descriptions are used to reveal false positives in smell detection. The usage of relevance filtering makes smell detection usable during floss refactorings. Relations and details that are important to understand a design problem are defined for each smell and used to present related smells in context. All three measures prooved to be useful in the conducted case study. They have a positive impact on the overall usability of smell detection for floss refactorings.

REVIEW OF THE STATE OF THE ART

The term “Smell” as a potential opportunity for a refactoring has been mentioned first by Beck and Fowler in [Fow99]. Since then, automated tools to detect code centric problems (like too many parameters) have become available (see 2.5). Research has focused on more efficient ways to detect smells [TM03] and has especially investigated how to detect more complex design problems [Mar02].

The problem of automated smell detection has been attacked from different sides. Approaches can be divided into two categories. On the one hand, there are approaches exploiting the static structure of smells. Some of these use logic programming to analyse the structures, others analyse the abstract syntax tree of the program’s source code. On the other hand, some approaches base the detection on metrics and their automated interpretation [Mar02].

Another analysed aspect is the usability and user experience of such tools. Both are important for efficient smell detection and acceptance of automated approaches.

2.1 REFACTORING OPPORTUNITIES: SMELLS, DESIGN FLAWS, ANTIPATTERNS

Kent Beck and Martin Fowler coined the term “smell” in [Fow99] for structures in code that possibly need a refactoring. Fowler and Beck describe a set of 22 smells. However, they explicitly do not give any precise criteria to identify those smells in code. Instead, Fowler and Beck refer to the developer’s intuition and experience [Fow99, page 75]. These 22 smell descriptions range from complex design problems (like “Parallel Inheritance Hierarchies”) and problems on class level (e.g. “Large Class”) to tiny problems like “Method Has Too Many Parameters” [EM02]. These different levels are not stated though implicitly given. Beck and Fowler give only some hints on the relation of design patterns [GHJV95] and smells. They reason on the relation of feature envy and the strategy and visitor pattern [Fow99, page 80]. Gamma et al. give further hints (see 5.1), however no comprehensive catalog is provided currently.

Marinescu gives a more formal definition called “design flaw”. A design flaw is a negative property of an element in a system. The design flaws are explicitly related to a level of design, i.e. package, class, method, etc. Such elements with negative

properties expose a deviation from criteria characterizing non-functional high-quality designs [Mar02, page 63]. This deviation from a given set of criteria is expressed in metrics and automated interpretation of measured values. The characterization of high-quality designs is based upon a study of 45 Java projects in [LM06].

Tom Tourwé and Tom Mens refer to these structures also as “refactoring opportunities” [TM03]. They report on a framework using smells to propose adequate refactorings.

Brown et al. provide a pattern language to describe so called “AntiPatterns” [BMMIM98]. AntiPatterns are commonly occurring solutions that cause obvious negative consequences. AntiPatterns can be the result of actions taken by the different participants of a software project. AntiPatterns can occur, among others, in design, architecture and processes [BMMIM98, page 7].

Mika Mäntylä et al. propose a taxonomy for the initial set of smells provided by Beck and Fowler [MVL03]. The 22 smells of Beck and Fowler are grouped into six categories: Bloaters, Object Orientation Abusers, Change Preventers, Dispensables, Encapsulators and Couplers. From a survey done in an industrial project, Mäntylä et al. compute correlations between smells. Their taxonomy is applied to these correlations. They conclude that this taxonomy and the empirical study of smell relations is only of initial nature. The study shows that relations between smells exist, and are not only pure theory.

2.2 ANALYSIS BASED ON STRUCTURAL DETECTION

One type of approach to smell detection is the use of pattern matching. Several approaches in research make use of such a “structural detection” to find design problems by their typical structure in code.

2.2.1 *Logic Meta-Programming Framework for Supporting the Refactoring Process*

Bravo, Tourwé and Mens developed a framework based on logic meta-programming. Their framework uses logic to reason about sourcecode and propose refactorings [Bra03] [TM03]. All detections and refactoring proposals are implemented with logic in the SOUL language. SOUL is based on prolog and integrated with Smalltalk Visualworks ¹.

¹<http://www.cincomsmalltalk.com/> accessed on 28.12.2009

The detectors are implemented as logic predicates that reason about the structures present in analysed sourcecode. Some detectors make also use of metrics. All smell instances have an attached weighting. This weighting is computed from the “badness” of a smell itself, weighting between smell pairs and user given sorting.

The selection of logic programming is based on the fact that this paradigm enables fast execution and easy description of complex queries on huge factbases [Bra03]. The work of Kniesel et al. does support this selection [KHR07]. Speicher et al. show in [SRK07] that complex formal analysis can be directly translated to logic predicates.

In [SAK07], Speicher et al. present “GenTL”, a generic analysis and transformation language. GenTL uses snippets of the analyzed language, called “code patterns”, to select elements during analysis. The patterns may contain variables as placeholders for elements of the analyzed language. The use of code patterns and variables balances needed expressiveness, ease of use and high abstractness with the power of logic meta programming. However, currently no implementation is available.

2.2.2 *Java Quality Assurance by Detecting Code Smells*

Van Emden and Moonen present in [EM02] their early work on code inspection and smell detection. Their approach is based on a generic parser generation and term rewriting framework. Some smells are detected and presented as graphs [Sli05, page 14]

They state that smells are characterized by different smell aspects. They distinguish between primitive smell aspects and derived smells aspects [EM02] to split smell detection into two steps. Derived smell aspects are inferred from primitive smell aspects.

2.2.3 *Code Smell Detection in Eclipse*

Inspired by the work of van Emden and Moonen, Slinger presents in [Sli05] an approach to detect code smells. The approach is implemented as plugin for the Eclipse Integrated Development Environment (Eclipse IDE).

Primitive smells and smell aspects are collected by visitors traversing the abstract syntax tree of the analysed sourcecode. These facts are input to a calculator for relational algebra. This calculator is used to infer more complex design problems from the collected facts.

2.3 ANALYSIS BASED ON METRICS

Another type of approach to smell detection is the use of metric computations. Either complex, specialized metrics are developed or several simpler metrics are combined into a strategy to detect smells.

2.3.1 *Distance Based Cohesion*

Frank Simon et al. report in [FS00] a detection algorithm to find opportunities for four different refactorings. Those are the refactorings “Move Method”, “Move Field”, “Extract Class” and “Inline Class” [Fow99]. Simon et al developed the metric “Distance Based Cohesion” to measure the similarity between two entities (methods and fields in the cited paper). This metric can be used to cluster entities. It provides opportunities for refactorings that improve code towards the principle “Put together what belongs together”. These detected opportunities suggest to move a method or field and extract or inline a class.

2.3.2 *Detection Strategies*

Radu Marinescu developed a whole framework to define smell detectors from the composition of metrics in [Mar02], [Mar04] and [LM06]. Marinescu uses metrics and filterings, that are logically composed into so called “detection strategies”.

To define new detection strategies for a design heuristic, appropriate metrics have to be chosen first. The next step is to select filterings for these metrics. These metric filters are logically composed with *and*, *or* and *not* operators. The result is a single, encapsulated detection rule for a design flaw, that can be effectively computed.

Some heuristics explicitly state semantics that can be related to the selected metrics. Those allow for a “semantical filtering”. If absolute numbers are part of the heuristic, an “absolute semantical filter” is possible.

Classes should not contain more objects than a developer can fit in his or her short-term memory. A favorite value for this number is six. [Rie96, page 72]

Marinescu states that some heuristics do not allow for absolute semantical filters. Marinescu calls these heuristics fuzzy and presents two further types of filters. A “relative semantical filter” considers the highest or lowest values of a dataset. An example for this heuristic is

Methods of high complexity should be split. [Mar02, page 56]

For heuristics mentioning extreme values a, “statistical filter” is proposed by Marinescu. Statistical filters catch extreme, abnormal values. An example for such a heuristic is

Avoid packages with an excessively high number of classes. [Mar02, page 56]

In the latest publication [LM06] of Marinescu and Lanza in 2006, only absolute semantical filters are used. For fuzzy heuristics, deviations from normal values are encoded into the thresholds. These are based on a study about 45 Java projects, measuring complexity per line, lines of code per method and number of methods per class. Averages and standard deviations as well as lower and higher margins are computed from this collected data. Assuming normal distribution, 70 % of the values will be in this interval between lower and higher margin. Very high values are assumed to be 50 percent higher than the higher margin (see also table 2.1). This statistic is completed by universally accepted thresholds like one, few (2-5), short memory cap (7-8) and fractions that seem natural to humans like a quarter, a half and two thirds [LM06, page 17f]. Marinescu and Lanza have been able to show concrete detection strategies over a dozen design flaws in [Mar02] and [LM06].

Metric	Low	Average	High	Very High
CYCLO / Line of Code	0.16	0.20	0.24	0.36
LOC / Method	7	10	13	19.5
NOM / Class	4	7	10	15
WMC(CYCLO)	5	14	31	47
Average CYCLO (AMW)	1.1	2.0	3.1	4.7
LOC / Class	28	70	130	195
NOM / Class	4	7	10	15

Table 2.1: Statistical thresholds based on 45 Java projects [LM06, page 16]

An example for such a concrete detection strategy is the following rule to detect the “feature envy” design flaw: Fowler and Beck define feature envy as a method that is more interested in foreign data than in data of their own class [Fow99, page 80]. Fowler suggests to move such methods to the used data. Arthur Riel has a similar heuristic for good object oriented design: keep related data and behavior in one place [Rie96, page 27].

Marinescu identifies three symptoms to be characteristic for this design flaw. The method accesses more than a few foreign attributes, it uses more foreign than local attributes and those foreign attributes belong only to few unrelated classes. The first two symptoms detect feature envy that only hurts evolution. Excessive use of foreign

data while ignoring the class's own data result in methods at the wrong place. Such methods are hard to maintain as changes and bugs ripple through the call chains (also called “ripple effect”)[LM06, page 84]. Constraining the dispersion of used foreign data helps to identify relevant opportunities to move the method close to the used data. If the used data is provided by more than a few classes, moving the analysed method is harder (possibly some parts have to be extracted into new methods first)[LM06, page 85].

These symptoms are detected by three metrics with appropriate thresholds. They are derived from the underlying heuristics. To measure the access of foreign data the metric “Access of a method to Foreign Data” (AFTD) is used. It counts the number of accesses of a method to foreign data either directly or through accessor methods². The ratio of local and foreign used data is measured by “Locality of Attribute Accesses” (LAA) by relating AFTD to the total number of accessed data in the analysed method. Finally, the dispersion of accessed foreign data is measured with the metric “Foreign Data Providers” (FDP) that counts the number of unrelated classes that contain the accessed foreign data. A pitfall in the definition of these metrics are the notions of unrelated classes and accessed data. Their definitions are essential to implement the same metrics and reproduce the results. Marinescu and Lanza consider the use of accessor methods and direct accesses to fields identical. Further they exclude any accesses to data within the same inheritance hierarchy, thus unrelated means “not in the same inheritance hierarchy”.

Finally, all three symptoms are completed by choosing appropriate thresholds and are combined with a logical and operation (see listing 2.1).

Listing 2.1: Feature Envy Detection Strategy

```
A method has feature envy when:
    AFTD > few (2-5)
and LAA < a third (~0.33)
and FDP <= few (2-5)
```

This detection strategy is, among many more, analysed in chapter 3.

2.3.3 Using History in Detection Strategies

There are improvements to detection strategies using history information [RDGM04]. Daniel Ratiu et al. improved in [RDGM04] the detection strategies of Marinescu

²Accessor methods in Java are also called getter and setter methods[Ric00b].

[Mar04], for the godclass and dataclass design flaw, by applying them to a range of versions of a project.

Whenever a method was added or removed to a class, its version was taken into account. Changes within methods were not considered. Ratiu et al. compute stability and persistence of a design flaw by testing the class under inspection for the presence of design flaws for each version. Stability is the ratio of versions having the tested design flaw, while the version before also exposed the design flaw. Persistence is the ratio of versions having the design flaw in relation to all changed versions. Daniel Ratiu et al. conclude that persistent and stable design flaws are “harmless”. Such design flaws are part of the system for a long time and have almost never been refactored. Thus they seem not to do any harm to the development of the system, which aligns with the thesis’ idea of irrelevant smells.

2.4 USABILITY AND EFFICIENT SMELL DETECTION

Murphy-Hill and Black distinguish between **floss refactoring** and **root canal refactoring**. Floss refactoring is characterized by frequent refactorings that are interleaved with other changes (e.g. adding a feature). Root canal refactoring are infrequent, large blocks of refactoring, during which nearly no other changes are made. According to their study of 2008, the majority of refactorings is carried out as floss refactorings [MHB08a].

Murphy-Hill and Black postulate seven habits in order to build usable, highly effective, smell detectors [MHB08b]. These guidelines are backed by an empirical study, an experiment with a questionnaire, by Murphy-Hill [MH09, page 92]. He concludes that programmers value these guidelines and that the guidelines enable programmers to understand more smells with greater confidence [MH09, page 92]. Among these guidelines are the following:

Context-Sensitivity. A smell detector should first and foremost point out smells relevant to the current programming context. Fixing smells in a context-insensitive manner may be a premature optimization. [MHB08b]

Availability Rather than forcing the programmer to frequently go through a series of steps in order to see if a tool finds any code smells, a smell detector should make smell information as available as soon as possible, with little effort on the part of the programmer. [MHB08b]

Scalability. A proliferation of smells in code should not cause the tool to overload the programmer with smell information. [MHB08b]

Relationality. A smell detection tool should be capable of showing relationships between code fragments that give rise to smells. [MHB08b]

Mealy et al. conducted a usability study of software refactoring tools [MCSW07]. They derived a set of usability guidelines from 11 collections of such guidelines. Further, guidelines on the required level of automation are added. Mealy et al. propose nearly full automation (called “level six” out of eight levels) for the phases of acquiring and analysis during refactorings. Within this automation level, the computer selects a way to do the task, executes it automatically and then informs the human. Some smells are too fuzzy to be computed. An example is the speculative generality smell that is found in classes that do more or are more flexible than is required by the users of a system. It is not possible to reason about this “extra flexibility” [MCSW07]. Thus they conclude that smell detection should not be completely automated.

Mealy et al. use the derived set of guidelines to analyse four common refactoring tools. From the results of this study they infer that work on the provided level of automation in current tools is needed. Automation of smell detection and refactoring proposal is required to improve the usability of such tools [MCSW07].

2.5 TOOLS

A wide range of tools exist that provide static analysis of software systems [Chi06]. Available tools range from style and bug checks as well as maintainability indices and metric measuring used in industrial projects to research prototypes to detect smells.

Checkstyle³ is a static analysis tool to validate sourcecode conventions. The provided rules can be configured and the tool can be extended with new, custom defined, rules [Chi06, page 47]. Additionally Checkstyle is capable of computing several metrics. All results are presented in a report. The analysis and reporting is also integrated into Eclipse (shown in figure 2.1).

The interpretation of violations and metric results is up to the user of Checkstyle. There is no integration with a refactoring tool, like the refactorings that are part of the Eclipse Java Development Tools.

FindBugs⁴ uses static analysis to detect bugs in Java sourcecode. The analysis is based on patterns that are often errors [Chi06, page 62]. The tools does not detect any design problems nor is it integrated with refactoring tools.

³<http://checkstyle.sourceforge.net/> accessed on 29.12.2009

⁴<http://findbugs.sourceforge.net/> accesses on 29.12.2009

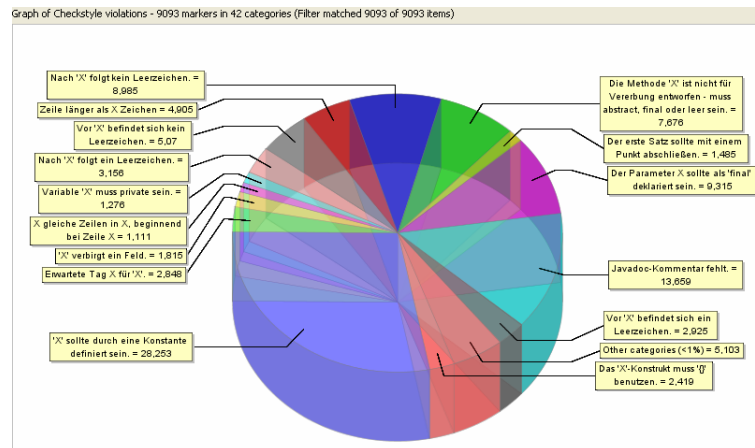


Figure 2.1: Checkstyle "Violations Chart" giving an overview of the violations shares

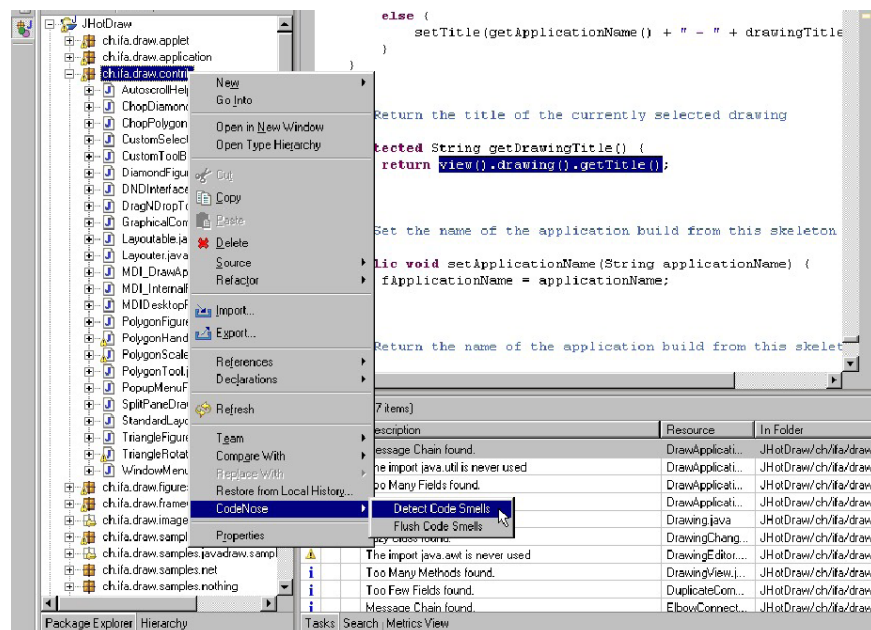


Figure 2.2: CodeNose

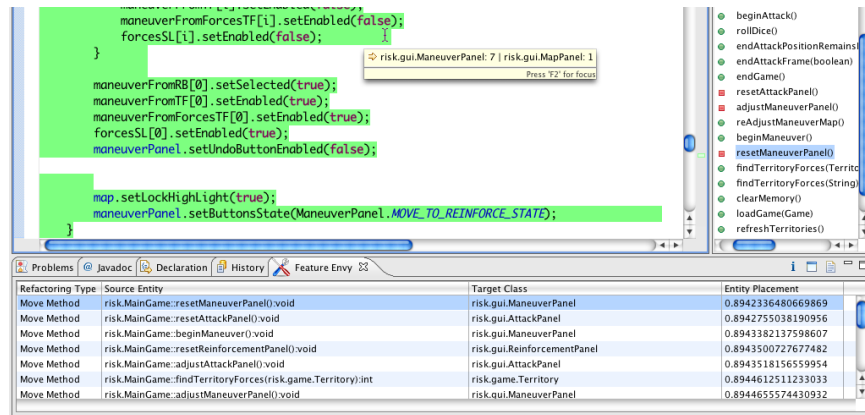


Figure 2.3: JDeodorant showing instances of feature envy in a special view.

XRadar is a meta-tool [Chi06, page 30]. It combines nine different tools to quantify several quality attributes for design, architecture, maintenance and testing. XRadar has been successfully used in an industrial project to identify problem areas and measure the refactoring progress in a legacy system [KLB05]. The reported process of prioritized refactorings and architectural changes supports the conclusion that XRadar fits a root canal refactoring approach.

CodeNose is the prototype developed by Slinger and Moonen [Sli05], see also section 2.2.3. CodeNose is a plugin for Eclipse. It detects design problems by traversing abstract syntax trees. Design problems are inferred with a relational algebra calculator using collected primitive and derived smell aspects [Sli05]. Figure 2.2 shows the manual starting of an analysis for a selected package (left side) in the Eclipse Package Explorer. All results are listed in the Eclipse Task View at the bottom of figure 2.2. It is possible to navigate from these listed smells in the task view to the attached locations in the Java editor of Eclipse. CodeNose does not provide any further integration with refactorings of Eclipse Java Development Tools.

JDeodorant detects smells and can automatically refactor selected smell instances [FTC07]. The selection is provided by the user and it starts only upon the command of the user. The tool is limited to only two smells. The two smells supported are feature envy and type checks [FTC07] [TCC08]. Each smell is presented in its own view, listing all detected instances (shown in figure 2.3).

The approach of Fokaefs et al. uses (similar to CodeNose [Sli05]) the abstract syntax trees for Java inside of Eclipse to analyse sourcecode, detect smells and apply refactorings. The inspection has to be triggered by the user of the tool. To apply a refactoring, the user has to select a proposed concrete refactoring and trigger its application.

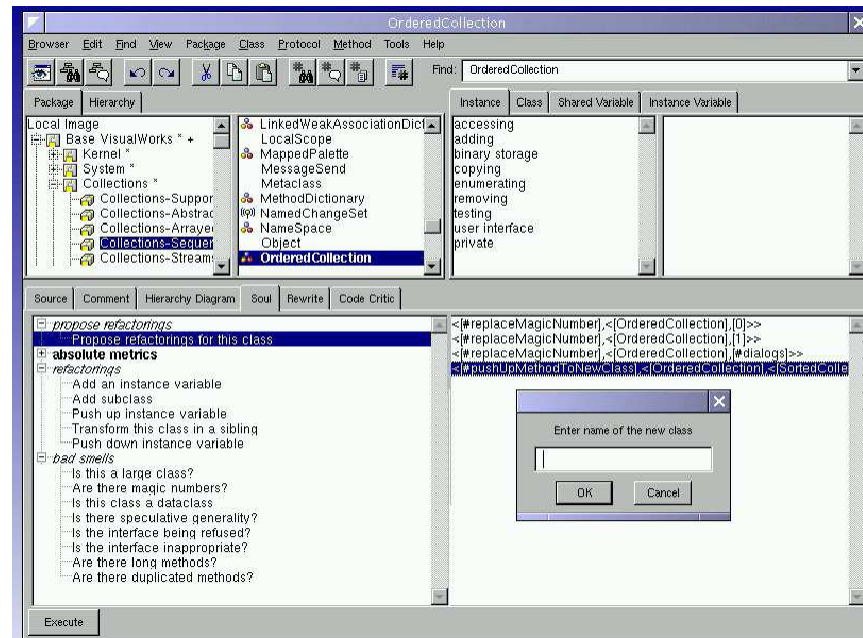


Figure 2.4: Smell detection within the Refactoring Browser of VisualWorks Smalltalk

Bravo et al. integrated their work into the **Refactoring Browser of VisualWorks Smalltalk** [Bra03]. Their tool integrates as a new tab into the bottom part of the refactoring browser (figure 2.4). Within the tab on the left hand side, different queries are presented to the user for execution. Among those are the detection of several smells and the proposal of different refactorings. This tool provides integration of smell detection and refactorings.

Murphy-Hill and Black presented an approach called **Ambient View** to display detected smells within a sourcecode editor. They state that their approach follows all guidelines presented in [MHB08b]. The tool does provide only a very small amount of different smell detectors.

Summarizing the current state of tooling in static analysis and especially smell detection, all presented tools lack one or several important points. Checkstyle and FindBugs are examining no design problems. XRadar is a meta-tool and provides no analysis itself. CodeNose and JDeodorant do not use metrics to find design problems. The tool of Bravo et al. works with Smalltalk and is not available for the currently more popular Java language. Currently nearly no tool is available to experiment with contexts and evaluate the usage of contexts in an industrial project. Thus for the development of different contexts and their evaluation, the state of art in smell detection has to be implemented in the Cultivate Project. Cultivate provides a stable platform to implement metrics and smell detectors within the logic meta-programming paradigm.

DESIGN FLAWS

This chapter analyzes several heuristics and detection strategies for design problems. The majority of detection strategies is provided by Marinescu and Lanza. Their strategies are set into relation to several heuristics of good and bad design quality. This chapter provides the foundations and definitions for the following chapters.

Marinescu and Lanza divide design flaws into three major categories: “Identity”, “Collaboration” and “Classification”. The first category groups flaws about the identity of an element, considering which data and ooperations are used and how complex the element is. The collaboration category contains design flaws concerned with the relation between elements. Design flaws regarding inheritance hierachies are grouped into the classification category.

Two detection strategies have been created by the author of this thesis: “middle man” and “iceberg class”. They have been developed as part of this thesis according to the recipe of Marinescu [Mar04].

- Identity Flaws
 - Godclass
 - Dataclass
 - Feature Envy
 - Brainclass
 - Brainmethod
 - Middle Man
 - Iceberg Class
 - Law of Demeter Violation
- Collaboration Flaws
 - Shotgun Surgery
 - Intensive Coupling
 - Dispersed Coupling
- Classification Flaws
 - Tradition Breaker

– Refused Bequest

3.1 GODCLASS

A characteristic of good object oriented design is the uniform distribution of complexity in a system among all classes. Further, data and related behavior should be kept close to each other while most of the methods of a class should collaborate on most of the classes' data [Rie96, page 27, 50]. Another phrasing of this heuristic is the “Single Responsibility Principle”, defined by Martin and Martin in [MM06, page 115]. The single responsibility principle states that every class should only have one reason to change.

A problematic deviation from this rule is the “godclass” design flaw [Rie96, page 50] [LM06, page 80]. Godclasses are locations in the system where complexity and knowledge about other classes' internals are centralized. These classes expose the symptom of exceeding complexity. Thus they have also an exceeding size, as size and complexity (when measured with McCabe's cyclomatic complexity [McC76]) are correlated measures [BW02]. Fowler's smell “Large Class” is another name for this symptom. Brown et al. have listed this design problem as an AntiPattern. They call it a “The Blob” [BMMIM98, pp 73].

The exceeding size and complexity are a problem for maintenance and evolution, because fault-proneness and number of faults are correlated measures [BW02]. Godclasses are centralizations of behavior and are serving more than one responsibility. Thus their cohesion is likely to be low. Multiple reasons to change a class are suspected to increase its fault-proneness [MM06, page 115]. As a consequence this design flaw should be avoided most of the time and refactoring them is a very complex task [LM06, page 83].

Measuring complexity, cohesion and access to foreign data of a class is used to detect a godclass design flaw. The complexity of a potential godclass is measured by the metric “Weighted Method Count”, i.e. the sum over the cyclomatic complexities of its methods. Marinescu and Lanza refer to their study of 45 projects to choose a threshold for the complexity of a class. As godclasses have an exceeding complexity, the value “very high WMC”, which is 47, in the study is reasonable (see also table 2.1). Marinescu and Lanza choose the metric “Tight Class Cohesion” (TCC) to measure the cohesion of a class. TCC is the relative number of methods accessing the same field (i.e. they are directly connected). This metric ranges from zero (no cohesion) to one (full cohesive). In order to detect low cohesion in potential godclasses, a threshold of a third works well [Mar04]. Access to foreign data is measured by counting all

direct and indirect accesses to fields of unrelated classes. Indirect accesses are to a field are calls using an accessor method of that field. Any access to foreign data breaks encapsulation. Selecting a threshold by the book would allow no access to foreign data. However, some access to foreign data does not raise many problems, thus Marinescu's detection strategy for godclasses allows a few (i.e. two to five) accesses to foreign data. [Mar01].

3.2 DATACLASS

Keeping related data and behavior at one place is another characteristic of good object oriented design [Rie96, page 50]. Classes that contain almost no behavior and define their interface only in terms of data and accessor methods deviate from this heuristic. Such classes are called "dataclasses" and represent a design flaw within the context of object oriented design [Fow99, page 86]. Dataclasses can be detected by exploiting the fact that its interface is only focused on data. This design flaw is in general a problem; however, there are cases when it is not (as discussed below).

The major characteristic of a dataclass is its data focused interface. This is visible in the number of public fields and accessor methods (e.g. getter and setter methods in Java). To measure these properties, the simple metrics "Number of Public Attributes" and "Number of Accessor Methods" can be used. Further, the interface exposes nearly no behavior, thus the complexity of the dataclass is low. This property can be measured with the already described WMC metric (see godclass).

The existence of a dataclass implies that related behavior is placed in other classes. These classes break the encapsulation of the dataclass and therefore violate a fundamental characteristic of object oriented design. Such designs may even exhibit a procedural nature using functions and data structures (i.e. dataclasses). This separates the behavior from the data. The separated manipulation of data is likely to cause data aliasing problems. This separation of data and behavior impacts negatively the understandability and maintainability of the chosen design [LM06, page 88].

Sometimes, dataclasses are not a problem - they are a well thought decisions. Robert C. Martin calls the rationale behind such a decision the *fundamental dichotomy between objects and data structures* [Mar08, page 97]. While objects hide data, data structures expose it. This is also called the *object/data antisymmetry*. As a result, procedural code makes it easy to add new functions. In contrast, adding new data structures is hard, because all related functions have to change [Mar08, page 97]. Whenever one needs to easily add new functions dataclasses are a solution, not a problem.

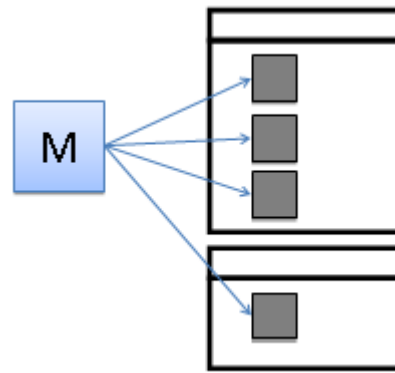


Figure 3.1: Feature Envy

The data driven interface of dataclasses is used to detect this design flaw. For object oriented designs exposing data and separating behavior is a problem, while in certain situations developers might explicitly choose datastructures and functions.

3.3 FEATURE ENVY

The design flaw “feature envy” is closely related to the problematic separation of behavior and data and the dataclass design flow. Methods exhibiting feature envy use too much foreign data while ignoring the local data of their enclosing classes. This characteristic is used to detect the design flaw [LM06, page 84]. Methods having feature envy should be moved closer to the used data.

Beck and Fowler describe feature envy as a refactoring opportunity for the “Move Method” refactoring [Fow99, page 80]. This is most successful in cases where the envied data is only provided by one or two classes. Figure 3.1 shows a situation, where the method M has feature envy. A possible refactoring is to move the method M into the class that contains the three accessed attributes.

To find these refactoring opportunities, the metric “Foreign Data Providers” (FDP) counts the number of types that provide the used data [LM06, page 85]. The amount of used foreign data is again measured with the ATFD metric (see godclass), this time defined for a single method. The ratio between own and all attributes (own and foreign) that are used is expressed with the “Locality of Attribute Accesses” metric. This metric ranges from zero, i.e. all used data is foreign, to one, i.e. all used data is owned by the enclosing class. Assuming that using some foreign data does not impose a problem to the overall design, the detection strategy filters for the usage of more than a few foreign attributes and a usage ratio below a third [LM06, page 85].

Cases of special importance are envied dataclasses. In such a situation, a class uses several attributes of the dataclass that itself lacks behavior. It might be necessary

to split the method and extract the part that uses only attributes of the dataclass. Afterwards the method can be moved into the dataclass [Fow99, page 80].

3.4 BRAINCLASS

The brainclass smell captures classes that centralize complexity in a system. It complements the godclass smell, because access to foreign data is ignored. Classes that centralize complexity are hard to change and maintain. Thus they slow down the evolution of the system [LM06, page 97 and 80].

Two variants of brainclasses exist. Brainclasses contain at least one brainmethod and are very large and complex. Other brainclasses may contain only one brainmethod, however are extremely large and complex. The detection strategy of Lanza and Marinescu takes both variants into account. The cohesion of a brainclass is low, as multiple unrelated sets of methods have been packaged into one class [LM06].

The detection strategy for brainclasses uses the detection strategy for brainmethods to detect the existence of at least one brainmethod in a class. Further the length is measured in lines of code or number of statements. Both measurements are correlated [BW02] and range from zero to infinity. The complexity of a method is computed with a metric measuring McCabes cyclomatic complexity [McC76].

Similar to godclasses, a brainclass contains also sets of unrelated methods. The low cohesion of brainclasses is again measured with the “Tight Class Cohesion” metric.

Both variants of brainclasses are important. Brainclasses with multiple brainmethods harm the evolvability and maintainability of a system, especially if the class is very large and complex in total. More complex and even larger classes harm the evolvability and maintainability. It is likely that one brainmethod exists in such cases, that provides a starting point for a splitting of the whole class into smaller, more cohesive parts [LM06].

3.5 BRAINMETHOD

Incremental changes of a method, adding constantly more functionality, can create abnormal methods. These methods centralize the functionality of a class and possibly also of the whole system. Such methods are called brainclasses.

Brainmethods tend to be very long methods. Beck and Fowler describe this characteristic alone as a smell, called “Long Method” [Fow99]. Usage of excessive branching makes a brainmethod also very complex. The number of used variables exceeds the

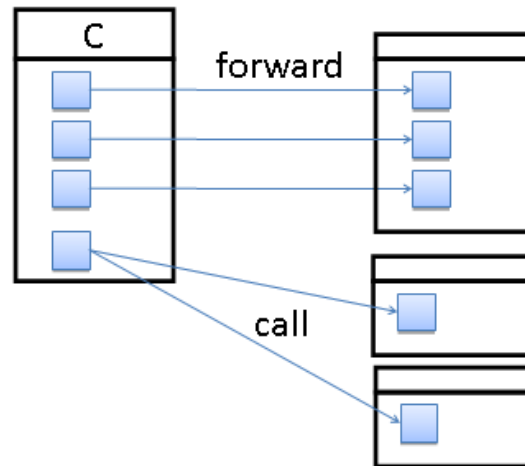


Figure 3.2: Middle Man Smell

capacity of the short term memory. Such methods are harder to understand, debug and maintain [LM06].

The detection strategy of Lanza and Marinescu uses the mentioned characteristics expressed as metrics. The complexity is measured with McCabe's cyclomatic complexity [McC76]. Additionally a simple metric called “Max Nesting” computes the maximal nesting level of a method. The metric “Number of accessed Variables” (NOAV) counts the number of accessed variables. Further, the size of the analysed method is measured using “Lines of Code” or “Number of Statements” [LM06].

3.6 MIDDLE MAN

Encapsulation, an important feature of object orientation, is often accompanied by forwarding [Fow99, page 85]. Several design patterns make extensive use of it to add flexibility to a design (like decorator, composite, adapter) [GHJV95]. Classes possibly forward too much and neglect taking real responsibility [Fow99, page 85]. Such classes are called “middle men”.

Beck and Fowler state that classes that forward too much complicate the understanding of such classes. This is less problematic for small classes, than bigger ones. Thus not only the share of forwarding methods is interesting but also their absolute count.

Figure 3.2 illustrate a middle man. The class “C” calls methods in three classes (on the right side). The first three methods of class C just forward to methods of another class. Only the fourth, last, method of class C takes responsibility and calls other methods. Over half of the methods of C take no responsibility. C is affected by the middle man smell.

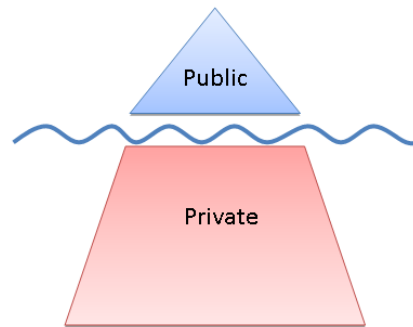


Figure 3.3: Iceberg Class

The detection strategy uses two metrics measuring on methods of an analysed class. The metric “Number of forwarding methods” counts all forwarding methods in a class. These methods forward the method calls to an instance of another type, hold in a member field of the analysed class. The number of forwarding methods is compared to the total number of methods in the analysed class. This comparison yields the ratio of forwarding methods.

Relevant instances of the middle man smell contain more than a few forwarding methods, i.e. more than 2, and have a forwarding ratio greater than 50 percent.

3.7 ICEBERG CLASS

Encapsulation can be misused and create middle man classes. It can also be forced in another bad direction: a class takes a lot of responsibility. Typically a major part of the complexity is hidden in private methods. Micheal Feathers reported this smell¹ in 2005. He calls such classes “iceberg classes”.

Most complexity of an iceberg class is *below the water surface*, like for a real iceberg. This situation is illustrated in figure 3.3. The public complexity above the water surface is ruled out by the huge private complexity under water.

At first look, an iceberg class is well encapsulated. However, it does break the single responsibility principle of Martin [MM06]. Further, an iceberg class is hard to test. Most of its complexity is hidden in private methods, no unittest can directly and isolated specify their behavior. It is also impossible to reuse this complex behavior. Other classes cannot call these methods.

The detection strategy “iceberg class” makes use of this striking characteristic: private complexity is a multiple of the public complexity. Significant iceberg classes have multiple private methods (e.g four methods). The complexity is measured using

¹<http://www.artima.com/weblogs/viewpost.jsp?thread=125574> accessed on 07.01.2010

McCabes cyclomatic complexity on private methods and using “Weighted Method Count”, i.e. cyclomatic complexity summed over all methods.

3.8 LAW OF DEMETER VIOLATION

Controlling couplings and hiding and restricting information is necessary to keep a design maintainable [LHR88]. Lieberherr et al. describe a guideline for methods that should simplify modifications and complexity of programming [LHR88], called the “law of demeter”. The law of demeter restricts the set of types that can be called by a method. This results to some extent in the desired simplification of programming and modifications [LHR88].

Methods may not call arbitrary types, according to the law of demeter. Instead, a method *M* of a type *C* may call another type that:

1. is contained in type *C*,
2. is the type of a parameter of *M*,
3. is the type of a field contained in type *C*,
4. is instantiated by any method of type *C*.

This restriction should ensure that the software stays as modular as possible [LHR88]. Methods know only the immediate structure of their enclosing type. Any further structures of arguments or other types are hidden. This limitation controls the coupling of methods, because the number of used types is limited. Further, the law of demeter enforces information hiding by keeping structures private. This hiding makes modifications of methods simpler [LHR88]. The information needed to understand, modify and bugfix a method is localized. Only *close* structures are known. The localization reduces the complexity of programming [LHR88].

Methods that violate the law of demeter retrieve elements contained in substructures of accessed types. The concern of navigation these structures is thus scattered through the system. All calling methods have knowledge of it. It makes the methods more complex, less reusable and harder to change [LHR88]. As a consequence, also the structure of retrieved elements is also harder to change.

The detection of violations makes use of the above stated rules that form the law of demeter. All method calls within an analysed method are checked. The types containing these called methods are compared to the list of allowed types. If one accessed type is not allowed to be called, the law of demeter is violated.

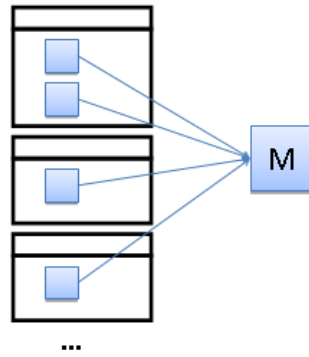


Figure 3.4: Shotgun Surgery

3.9 SHOTGUN SURGERY

Object oriented designs require collaboration among objects, having possibly several methods calling the same method. Nevertheless, the number of incoming method calls should not exceed a certain limit. Too many incoming calls damage the maintainability of a design and increase instability. They are one potential reason of the “shotgun surgery” smell [Fow99, page 80]. Lanza and Marinescu select this reason and define a detection strategy based on the amount of incoming calls and number of different calling types.

Figure 3.4 illustrates a situation of a method *M* exposing shotgun surgery. The method *M* is called by many different other methods. The method calls form “used by” relations between the calling methods and the method *M*. These calling methods are part of many different types.

A suggested limit for the number of incoming calls is the capacity of human’s short term memory. However its exact size is still debated. Early results by George Miller in 1956 revealed the fact that a maximum capacity exists. Miller gave a rough estimate of seven plus or minus two, which can also be referred to as “magical seven” [Mil56]. Marinescu uses this magical seven. Recent research by Nelson Cowan suggests an even lower capacity of four *chunks of memory* [Cow01].

Whenever a method is called by too many other methods, any change to such a method ripples through the design. Such changes are likely to fail when the number of to-be-changed locations exceeds the capacity of human’s short term memory [LM06, page 134]. The more different classes are involved, the more risk is introduced into this situation. Thus shotgun surgery is especially dangerous whenever the calling methods are dispersed among many classes [LM06, page 134].

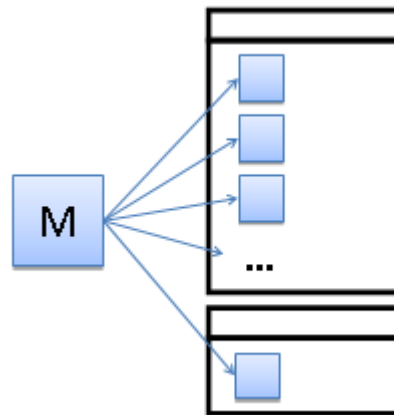


Figure 3.5: Intensive Coupling

3.10 INTENSIVE COUPLING

Some methods grow not only in size, but start to call many different methods. Methods that intensively call other methods, dispersed only over a few types, are affected by “intensive coupling” [LM06, page 120].

Figure 3.5 shows such a situation. The shown method *M* calls many other methods that are part of only two types. The dispersion of method calls is low while the calling intensity is very high. The involved calls form a “uses” relation between the method *M* and the called methods.

Intensively coupled methods have an excessively chatty communication. This verbosity binds the method strongly to the related types [LM06, page 120].

The detection strategy for intensively coupled methods takes two variants into account. Intensively coupled methods may call more methods than the capacity of the short term memory. These called methods are part of few types, the dispersion is below 50 % [LM06, page 122]. Intensively coupled methods may also call only slightly more than a few methods, part of very few classes. The coupling dispersion is below a quarter in such cases.

The metric “Coupling Intensity” is used to measure the number of called, different methods. The dispersion is computed by the metric “Coupling Dispersion”. It measures the number of used classes divided by the coupling intensity.

Typically, initializers and configuration methods, e.g. concerning graphical user interfaces, are intensively coupled methods. Marinescu and Lanza add a further condition, testing the maximum nesting level of intensively coupled methods. Nesting levels below one or two are excluded. This has the desired effect to exclude initializers and configuration methods, however it is done implicitly. The approach of this thesis uses defined structures to exclude these special methods explicitly (see 6.2).

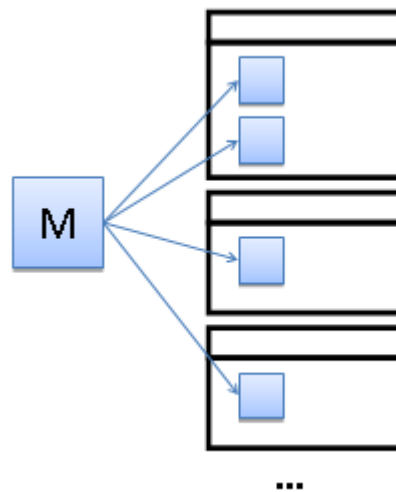


Figure 3.6: Dispersed Coupling

3.11 DISPERSED COUPLING

Complementary to intensive coupling, methods that call many other methods on many different types. Such methods have “dispersed coupling” [LM06, page 127].

Figure 3.6 shows a method *M* that has a dispersed coupling. The method *M* calls many methods. These methods are part of many different types. The method calls form a “uses” relation between *M* and the called methods.

Dispersed coupling introduces potential ripple effects into a design [LM06, page 127]. Whenever a called method changes, the method with dispersed coupling is very likely to change also, causing a ripple effect.

The related detection strategy checks if a method’s coupling intensity is beyond the capacity of short term memory. Further, methods with dispersed coupling have a coupling dispersion greater than 50 %.

Similar to intensive coupling, initializers and configuration methods, are typically affected by dispersed coupling. Marinescu and Lanza add again a condition, testing the maximum nesting level of intensively coupled methods. The approach of this thesis uses defined structures to exclude these special methods explicitly.

3.12 REFUSED PARENT BEQUEST

Back and Fowler report a smell describing subclasses that deny using their parent’s bequest, called “Refused Bequest”. They conclude that this smell is normally not a big problem, only if the class gets complex [Fow99, page 87]. Refused bequest hampers the understanding of an analysed subclass, because its relationship to the overall hier-

achy is unclear. If such a subclass adds substantial services to the interface, it neglects the parent's interface. Beck and Fowler regard this situation as more problematic and propose refactorings in this case[Fow99, page 88]. Lanza and Marinescu report two distinct design flaws and related detection strategies for both cases [LM06, page 145].

The detection strategy for "Refused Parent Bequest" detects classes that use or specialize less than a third of its parent class. Refused parent bequest is problematic, because it hampers the understanding of a class with regards to its inheritance hierarchy and makes the class less cohesive and coherent [LM06, page 145].

The detection strategy first tests classes for ignoring bequest, while having significant size and complexity. The ignorance of bequest is either computed by low rates of bequest usage or low rates of specialization. A parent with substantial bequest has more than a few methods to specialize, e.g. protected methods in Java. Child classes ignore, instead of use, this bequest, if their "Baseclass Usage Ratio" is less than a third. Child classes do not specialize this bequest, if their "Baseclass Overriding Ratio" is less than a third.

Relevant subclasses have a total complexity above average, compared to the study of Lanza and Marinescu [LM06]. Total complexity is measured with the metric "Weighted Method Count". The average complexity of its methods is above average, measured with the "Average Method Weight" metric. Further, the number of methods is also above average.

3.13 TRADITION BREAKER

Inheritance should be used to form specialization hierarchies [Rie96, page 81]. The interface of a class should increase smoothly down the inheritance hierarchy [LM06, page 141 and 152]. If a child uses only little bequest and adds excessively new services to the interface (compared to the parent), this child is a tradition breaker.

The affected inheritance hierarchy is potentially wrong, when tradition breakers appear [LM06, page 152]. Beck and Fowler suggest to remove such subclasses from the hierarchy and use forwarding instead [Fow99, page 87].

The detection strategy for tradition breakers exploits the fact that such classes excessively add services to the interface. A child class can only break a tradition, that exists and is significant. Thus the parent class is analysed for several properties. Its method should have a complexity above the average. The total complexity should amount to more than a half of the child's complexity. Further, the parent class should have more than half of the child's methods.

A tradition breaker is only a problem, if the analysed class has a substantial size and complexity. Thus the number of methods, weighted method count and average method weight are computed. The complexity of a method should be, on average, above the typical value stated in Lanza and Marinescu's study [LM06, page 16]. The total complexity of a substantial tradition breaker is at least very high, distributed to more than a high number of methods. These thresholds are also derived from the cited study [LM06, page 16].

Excessive increase of a classes interface is measured with the metrics "Number of Added Services" and "Percentage Number of Added Services". Tradition breakers add more services to an interface in a hierarchy than a class has methods on average. Further, these added services dominate the child class. Their share is above two thirds.

3.14 SUMMARY

This chapter explained the detection of significant smells. These smells can be detected with the help of metrics or structural detectors. All metrics and thresholds have been motivated by popular and widely understood design heuristics. Further, the impact of those smells to the design quality of a system has been studied. It is evident, that some situations are exceptions to the heuristics and thus to the detected smells. Identified smells are false positives in these situations. This reasoning is studied further in chapter 5.

RESEARCH QUESTION

This chapter presents the remaining problems present in the current state of the art in smell detection and floss refactoring. These remaining problems are used to phrase the underlying hypotheses of this thesis. The hypotheses are used to give concrete answers to the main research question: **“How can context-sensitivity improve smell detection for floss refactoring?”**.

4.1 REMAINING PROBLEMS IN THE RELATED WORK

Floss refactoring needs efficient smell detectors [MHB08b] [MHB08a]. Still, several problems remain in the current state of research. The current approaches do not support floss refactoring very well, because they neglect the context of smells. Further the relation of created structures (and design decisions) and smells is not investigated to reduce false positives during the detection of smells.

The cited approaches of Bravo et al., van Emden and Moonen and the approach of Slinger rely on the detection of structural patterns in abstract syntax trees to detect smells [Bra03] [EM02] [Sli05]. Beck, Fowler state that smells are mostly of “fuzzy” nature while Marinescu concludes that there are also fuzzy design heuristics [Fow99] [Mar04]. Marinescu bases his approach solely on metrics [Mar04]. Some design heuristics present them selfs as sharp rules, defining structural properties.

An example of such a sharp heuristic is the law of demeter [LHR88]. The law of demeter characterizes allowed method calls from an object’s point of view. Violations of this principle are variants of the “Message Chains” smell. A violation of the law of demeter may indirectly lead to a higher outgoing coupling (efferent coupling). While the latter can be measured by coupling metrics, the presence of chained method invocations can only be detected by exploitation of the method body’s structure.

As a consequence efficient smell detection needs a combination of metric based detection and structural analysis. Logic meta programming provides the expressive power and ease to formulate them, metric definitions, detection strategies and structural smell detectors [KHR07] [Bra03].

Radiu et al. use historic information to measure stability and persistence of a smell. This information is used to judge a smell’s relevance. Changes taken into account

are the addition or removal of methods in classes. However, not measuring changes within methods leaves out a lot of important changes. Frequent subtle changes are in the context of certain design flaws especially important. Shotgun surgery, dispersed coupling and feature envy are problematic as they force the developer to make a lot of small, sometimes equivalent, changes frequently [LM06]. This is also called the “ripple effect”. As Ratiu leaves out small changes, frequently changed classes suffering from the ripple effect are still evaluated as stable by his method. Such frequent changes, however, do have a relevance to the project’s team. Computing the stability and persistence of a design flaw only for versions with added or removed methods does not represent the whole range of an real artifact’s relevance. Efficient smell detection thus needs to evaluate fine grained historic information.

The cited available tools, excluding the Ambient View of Murphy-Hill and Black, do not fit well with floss refactoring and the guidelines of Murphy-Hill and Black.

CheckStyle does not measure design problems but problems with code conventions. FindBugs analyses project to find common patterns of possible bugs. Both tools are not able to detect smells.

XRadar evaluates problems in the design of a whole project, however its information about single elements is again a collection of metric values. These values have to be interpreted manually by the user. Such an approach does not scale well to detect smells and refactor them [Mar04] [WL08]. Further it is not aligned with the guidelines of Murphy-Hill and Black. The information of XRadar is not context-sensitive and does not fulfill the scalability requirements. The user has to start the detection procedure manually, so it does neither follow the availability guideline.

CodeNose and JDeodorant provide both smell detection as plugins for Eclipse. Both search abstract syntax trees for small patterns to identify smells. The detection is expressed in visitors that traverse the abstract syntax tree [Sli05] [FTC07]. The extension of the Refactoring Browser in VisualWorks Smalltalk by Bravo et al. uses logic meta-programming to express the search for patterns to identify smells [Bra03]. CodeNose and JDeodorant do not use any metrics at all while Bravo et al. do not use some consequently. These approaches lack a expressive combinations of metrics, detection strategies and structural detection. Further the guidelines for efficient smell detectors are not implemented. The user of these tools has to trigger detection manually, results are not context-sensitive and overload the user with information.

The Ambient View of Murphy-Hill and Black is an implementation of their guidelines [MHB08b], however it does not support extensibility to add many needed smell detectors for an industrial evaluation.

The current state of art in smell detection does not exploit context information to present relevant, understandable and easy accessible information about design problems. Thus it is more difficult to integrate these approaches into floss refactoring.

The concepts of a task context and relevance of an element with regards to a task [Ker07] are missing. Further the concept of an element's relevance compared to the history of the enclosing system is also missing. To implement the context-sensitivity and scalability guidelines both contexts may be needed. With the help of temporal relevance and task relevance, the amount of detected smells could be reduced so that the user is not overloaded with information.

Complex design problems are not existing in isolation [WL08] [Mar02, page 85]. Theoretical and empirical study of smells also supports that relations between smells exist [MVL03]. To understand them, the relation between smells and the affected elements has to be visualized and presented to the user. This implements the relationally guideline of Murphy-Hill, however has to respect the context-sensitivity and scalability guidelines.

Finally, several structures and design decisions, e.g. design patterns, are the root cause of certain smells. Such instances of smells are intuitively considered as false positives. The relation between structures and smells is studied in depth in chapter 5. None of the current approaches exploits this relation to reduce false positives.

To summarize, efficient smell detection needs information about the context of smell instances and the affected elements. Current approaches do not use this context information. Further the relation of structures and smells is not used.

4.2 HYPOTHESES

The research question, how floss refactoring can be efficiently supported, has to be investigated. This thesis adds context information to smell detection. Its main hypothesis is that adding context information makes smell detection more efficient in supporting floss refactoring:

H1 Context sensitive smell detection eases the identification of refactoring opportunities during floss refactorings.

This main hypothesis is subdivided into three smaller hypotheses. These are related to different problems and contexts that have been analysed before.

The relation of structures and smells is used to build a structural context of a smell. This context enables the approach to decide if the smell is a consequence of a defined structure. Such smells are then removed as false positives. Other approaches consider

persistently removing single instances of a smell [Sli05]. This does not solve the root cause of false positives and cures only a symptom. The structural context is thus expected not only to remove false positives. It is also expected to do so in a, for the user, scalable and adaptive way.

H2 A structural context exploiting relations between defined structures and smells reduces false positives in a adaptive way.

The overloading with information is a major obstacle, preventing efficient smell detection during floss refactorings. Kersten showed that filtering for task relevance improves the navigation in integrated development environments [Ker07]. The usage of history information provided, to some extend, also already relevance information [RDGM04]. Using the concept of relevance in a programming task and computing a similar relevance from the history of a project builds a relevance context. This context is expected to remove the information load during floss refactoring and present only relevant smell instances to the user.

H3 Using relevance contexts (from task and history information) decreases the information load during smell detection and makes it manageable for floss refactoring.

To understand a design problem, its context in the sense of co-located smells and affected program elements is of interest again [WL08]. The relations of smells and relations between affected elements is visualized with an interlinking context. Again relevance is important. Only relations with importance in a floss refactoring should be presented. This visualization of interlinkings is expected to provide a better explanation and understanding of a design problem.

H4 Visualizing the context of a smell and its interlinking with other smells helps to understand design problems during floss refactoring.

In the following chapters, a new approach to provide efficient smell detection for floss refactorings is outlined. First the relations between structures and smells are investigated. Further, the different contexts, their concepts and implementations are described. Finally the research question and hypotheses are explored in a case study, covering an industrial project.

RELATION OF STRUCTURES, SMELLS AND FALSE POSITIVES

This chapter investigates the impact of structures on the design quality in terms of known smells and design flaws. Structures of interest are design patterns, architectural decisions, application programming interface (API) design, embedded domain specific languages¹, unittests and elements of presentation layers. Structures have an impact on the existence of smells. Detecting such smells is in general considered a false positive. An improved detection approach should exclude them. Several well known structures are inevitably bound to discussed smells and design flaws. This relation is investigated in the following sections.

5.1 DESIGN PATTERNS

Design Patterns have positive, but also negative consequences, that can be design flaws. Some flaws are already stated as consequences in [GHJV95] while others show up only in certain variants or after careful analysis.

5.1.1 *Decorator*

The middle man and refused bequest smells are likely in instances of the decorator pattern. They are inevitably bound to decorators in a certain variant of the pattern. Such decorators inherit from an abstract decorator which takes the responsibility of method forwarding to the decorated type.

The decorator pattern is used to add additional responsibilities to another object dynamically. A common implementation (e.g. in [GHJV95] and JHotdraw² 6) of forwarding is to put the responsibility to forward method calls to the decorated object into an abstract decorator. Concrete decorators are subclasses of this abstract decorator. As a positive consequence, concrete decorators only need to override methods, to which they add behavior while they do not have to care about forwarding all other methods.

¹EDSLs, also named Internal DSLs [Fow07] or Fluent APIs [EE05]

²<http://www.jhotdraw.org/>

Abstract decorators suffer from the middle man smell. An abstract decorator delegates all responsibility while he himself does not take any. This is a clear indication for the middle man smell, because the abstract decorator does delegate more than half of his methods. Whenever an abstract decorator is used, middle man smells are inevitable within them.

Any concrete decorator that inherits from such an abstract decorator and overrides only a small amount of the methods is subject to the refused bequest smell. The abstract decorator is used to free any concrete decorator from the forwarding responsibility. It is also used to write small, compact concrete decorators that only take care of their extension to the decorated element. The result may be a small number of overridden methods. These concrete decorators suffer from refused bequest whenever the decorated element, and thus the abstract decorator, provide a substantial amount of methods for specialization. Compact concrete decorators that do not need to take care of forwarding inevitably refuse bequest.

The decorator pattern exposes middle man inside abstract decorator classes and refused parent bequest in concrete decorators.

5.1.2 *Composite*

The composite pattern has a structure similar to the decorator pattern while having a different intention. Refused parent bequest is an inevitable smell for composites.

The composite pattern is used to decompose objects into a tree structure, while clients are not aware of a separation between single objects and composed objects. Within composed objects, method calls are forwarded to all contained children. A usual approach is to maximize the interface of operations that components and composite have in common. Default implementations are part of the component class [GHJV95].

Any leaf element that inherits from such components will refuse parts of the bequest. The maximization of the components interface contradicts with the principles of designing class hierarchies [GHJV95]. The refused parent bequest smell is then inevitable, because not all operations in the interface of a component that are useful to a composite are useful to implement in a leaf.

The maximization of the component's interface is opposed to the principles of inheritance in object orientation. Due to this contradiction, refused parent bequest is an inevitable smell coupled with the composite pattern.

5.1.3 *Visitor*

In some designs data and behavior do not evolve in parallel. The visitor pattern is used to encapsulate behavior on a whole structure of objects. Behavior and data are splitted, the result are concrete visitors that envy the data of the visited object structure.

The foundation of object orientation is to keep data and related behavior in one place. However, sometimes this basic assumption is endangered when behavior evolves faster than the underlying data. The visitor pattern capsulates such behavior in one place, separated from the data. This makes the addition of new operations easier [GHJV95]. To accomplish the behavior's goals, the behavior has to access the separated object structure frequently. The feature envy smell is an expression of this contradiction with basic object oriented principles [Fow99, page 81]. The classes taking the role of concrete visitors envy the data of the concrete elements in the visited object structure.

Further the separated object structure may present itself as dataclass. Within the visitor pattern behavior is separated from the data to ease the addition of new behavior. Then, all behavior has moved completely into the classes fulfilling the role of concrete visitors. The classes of the visited object structure may remain without any reasonable behavior and are thus dataclasses.

5.1.4 *Strategy*

The strategy pattern is, besides the visitor pattern, another pattern that mainly helps to evolve behavior. Similar to the visitor pattern, the separation of behavior and related data forms the basis. This approach results either directly in the feature envy smell or is a major contributor to it.

To evolve a set of algorithms working on the same stable data and interface, the strategy pattern splits the algorithm's behavior from the related data into strategies and context. As for the visitor pattern, this split is a major contributor to feature envy within implementations of the pattern. Possibly the pattern itself causes the detection of the feature envy smell [Fow99, page 81].

5.1.5 *Facade*

Facades are designed to provide an interface for a whole set of interfaces in a subsystem to ease the usage of the subsystem. Within facades, several smells and design flaws

are compulsory [GHJV95]. A facade's responsibility is the reason for middle man and intensive and dispersed coupling.

The first responsibility of a facade is to know to whom a request (i.e. method call) has to be delegated [GHJV95]. As a consequence, facades may consist of method forwarding only. In such facades the middle man smell is imminent.

In more complex facades and subsystems, the facade's methods call many methods and classes of the underlying subsystem [GHJV95]. The number of calls and called types may exceed limits for reasonable object oriented design. Intensive coupling and dispersed coupling are the consequence. However, as a facade is explicitly designed, the smells are a necessary trade-off.

Facades possibly degrade to middle men or are intensively and dispersly coupled to the subsystem's classes. These smells are the required trade-offs of the facade pattern.

5.1.6 *Adapter*

Whenever a type should be used in the place of another interface, both object interfaces have to be adapted. Adapters provide a one way adaptation between an adaptee type and a target interface. Because of their nature, an adapter might also be a middle man.

If both interfaces are very similar the adapter likely just forwards to methods with possibly same signature. This forwarding is subject of the middle man smell. For such similar interfaces, an adapter is introduced, whenever the adapted type is not under the developer's control. In such situations, becoming a middle man is a necessary trade-off of the adapter pattern.

5.1.7 *Mediator*

Mediators are build to modularize the communication between different colleague classes. In contrast to the adapter pattern, mediators provide a bidirectional adaption of interfaces and behavior. Further, they provide a decoupling between the different colleagues [GHJV95]. Simple mediators may expose the middle man smell. More complex mediators have intensive or dispersed coupling to the mediated colleagues.

A mediator is an abstraction of the way how a set of colleague objects work together. The mediator pattern simplifies their communication protocol. The more complex this protocol is, the more complex and coupled a mediator is. Mediators might couple intensively to certain colleagues or couple to many (dispersed) colleagues when their number increases.

Further, the mediator pattern decouples colleague objects. Mediators build primarily for this reason tend to become middle men. Their mediation consists mainly of forwarding method calls. Thus they expose the middle man smell. As the decoupling is an explicit decision, the smell is a consequence one has to accept.

Explicitly introduced mediators may evolve into both extremes. Either they are simple middle men or they are intensively coupled to the mediated colleagues.

5.1.8 *Creation Methods*

Several patterns support the encapsulation of how to create an object, separated from the class itself. Patterns like Builder, Abstract Factory and Factory Method may be used to package complex creation logic, to create a whole family of depending objects or allow subclasses to decide on the concrete created object [GHJV95]. Kerievsky outlines their common denominator as “Creation Methods”, a method that creates instances of a class [Ker04, pp. 57].

Such creation methods are tightly coupled to the class they create instances of. Because objects are created in terms of data that is instantiated, creation methods may access a lot of this data. In this case, feature envy is identified. Any creation method with or without feature envy is conceptually coupled to the created class. The negative effects of feature envy have a minor weight compared to the effects of conceptual coupling.

5.2 PROCEDURAL COMPONENTS

To favor procedural programming in certain components of a system is a decision that is part of the overall architecture. Reasons range from the implementation of remoting components to simplified modeling of not overly complex domains. Such decisions should be taken carefully and trade off the simplicity for potential maintainability and change problems in a later evolution.

Related problems are the presence of dataclass and feature envy flaws. Within objects and methods of a procedural component there is no other chance than to envy the datastructures of input and output - feature envy is inevitable. The datastructures are by design exhibiting the dataclass smell (see also dataclass flaw 3.2, page 31).

In some components the datastructures are stable and only functions are added. These are best represented with procedural services and dataclasses [Mar08, page 97]. For domains having a moderate complexity, even the whole domain may be

modelled with services and dataclasses. Fowler calls this pattern the “Transaction Script” [Fow02, pp. 25]:

A Transaction Script([definition beginning on page] 11) is essentially a procedure that takes the [structured] input from the presentation, processes it with validations and calculations, stores [structured] data in the database and invokes any operations from other systems. It then replies with more data to the presentation, perhaps doing more calculation to help organize and format the reply. [Fow02, page 25]

The Transaction Script pattern implies dataclass smells at the datastructures and feature envy smells within the transaction scripts. This trade-off is well known for this pattern and approach to model domains. Modern remoting technologies like the webservises standard³ also force a procedural style at the system boundary communication with other systems. Such remoting services are defined in terms of operations, that are called from remote, and messages, that are passed over the wire.

Several decisions about architecture and design have an impact on the presence of smells like feature envy and dataclass. In the context of transaction scripts, datastructures and remoting services, these smells are the price that has to be paid.

5.3 EMBEDDED DOMAIN SPECIFIC LANGUAGES

Domain specific languages are small, usually declarative programming languages that are created and used with a narrow focus on a specific domain. The special focus on a domain is exploited to be able to express solutions in the level of abstraction of the domain itself [vDKV⁺00]. One way to implement a domain specific language is to use the mechanisms of an existing host language to define an embedded language, so called “embedded domain specific languages”. Several styles of this implementation approach are in contradiction with before presented smells and design flaws.

5.3.1 *Method Chaining*

In order to implement an embedded domain specific language, a common⁴ style is the use of method chaining [Fow07]. This style exposes several smells and design

³<http://www.w3.org/TR/ws-arch/> accessed on 07.03.2010

⁴Found in popular frameworks like JUnit 4.5 (www.junit.org), Google Guice (<http://code.google.com/p/google-guice>), Hamcrest (<http://code.google.com/p/hamcrest>) and StructureMap (<http://structuremap.sourceforge.net>). All accessed on 09.11.2009

flaws. Every domain specific language provides a form of grammar and syntax to build sentences with it. To achieve this goal within a host language, its mechanisms are used. In case of Java, these are classes, interfaces, methods, etc. There is at least one entry point (also called “Expression Builder”) to start sentences while the grammar is translated to constructs like method chains and nested functions [Fow07]. A method participating in such a method chain is modifying the state of the underlying builder or translates directly to a traditional API. Further, such methods also return another type instance, possibly the enclosing expression builder. To constraint the set of valid callable methods according to the grammar, another type may be returned.

In figure 5.1, taken from Google Guice⁵, bindings between abstract types and concrete types or concrete instances are configured. The listing 5.2 is a simplified version of the underlying EBNF. In the here presented, simplified version, a binding consists of a Java type and a bound Java type or instance. The expression builder is the base class `AbstractModule` with the method `bind(...)`. This method returns an instance of the class `LinkedBindingBuilder` as only calls of the method `to(...)` and `toInstance(...)` are allowed in the grammar of Google Guice.

Listing 5.1: Simple example of method chaining in a DSL

```
class MyModule extends AbstractModule {
    protected configure() {
        bind(Service.class).to(ServiceImplementation.class);
        bind(Service.class).toInstance( new ServiceImplementation() );
    }
}
```

The name of this implementation approach hints directly at a smell: “method chains”. To achieve the above presented readability of the code, chains of methods are used by the clients of such an embedded domain specific language. Fowler mentions method chains as a smell. They couple the client code to a number of types and methods and to the navigation between those types. This coupling makes the client code instable. Any change in this navigation chain renders the client code invalid [Fow99]. This is a before-known trade off, when using such an API. Such embedded domain specific languages that are implemented using method chains expose the method chain smell. However, it does not cause as many harm as other, accidental instances of method chains.

Listing 5.2: Simplified EBNF for the DSL shown in sample 5.1

```
Binding = ``bind`` JavaType To;
```

⁵<http://code.google.com/p/google-guice> accessed on 09.11.2009

```

To = ToType | ToInstance;
ToType = ``to`` JavaType;
ToInstance = ``toInstance`` JavaTypeInstance;

```

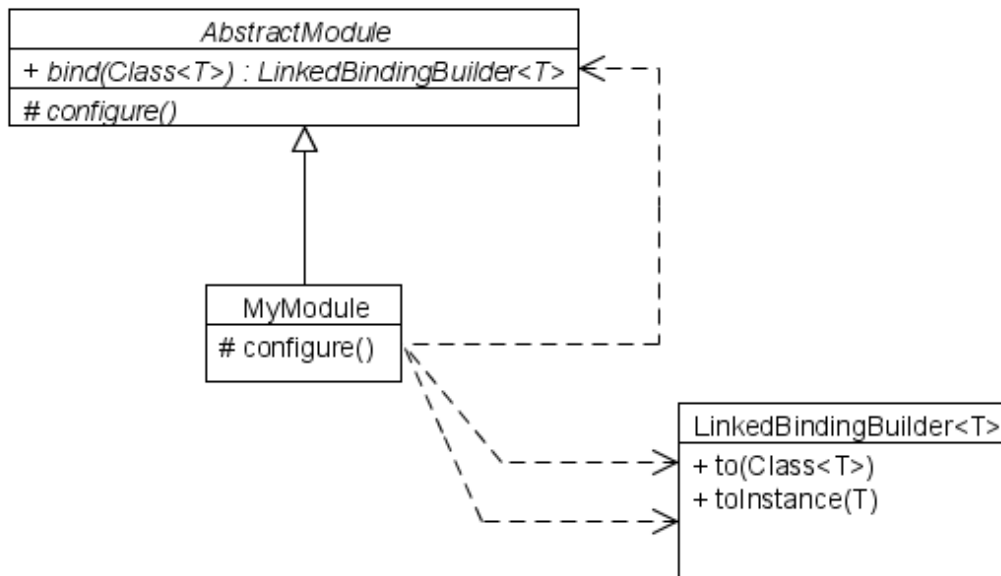


Figure 5.1: Classdiagram with dependencies in the sample 5.1

Further, having a chain of methods is a violation of the law of demeter [LHR88], which is problematic for the same reasons as given for method chains (see also 3.8). According to this rule, a method *M* may only call methods on the enclosing class, on objects created by *M*, on arguments passed to *M* and on instance variables of the enclosing class. Calling methods on objects returned by methods of unrelated classes does violate the law of demeter. The method chain approach does make frequent use of calling methods on returned objects. The use of an embedded domain specific language that is implemented with the method chain approach violates the law of demeter in a controlled way.

Methods that are part of such an embedded domain specific language modify the internal state of the expression builder. At the same time they also return instances of types. This breaks the “Command Query Separation” of Bertrand Meyer. According to Meyer, a method should either modify state, called a “Command”, or return a value, called a “Query”, but should never do both [Mey00]. The definition of an embedded domain specific language with method chaining violates this heuristic as a consequence of its nature.

Finally, Marinescu’s dispersed coupling design flaw is present in embedded domain specific languages that are implemented with method chains or nested functions.

Grammars of typical complexity, e.g. the “Binding API” of Google Guice, have many small types with two to five methods. Using such an embedded domain specific language causes the coupling intensity and coupling dispersion to exceed defined limits. Thus, dispersed coupling design flaws are present. Because of the many types to translate the complex grammar with a few methods, a client method calls a lot of different methods and types. As a result, the coupling intensity, i.e. the number of called distinct methods, is higher than good object oriented design should expose on average, i.e. the capacity of short term memory. As the involved types are small, the number of called distinct types is also higher than usual.

Embedded domain specific languages that use the method chaining approach to implement their grammar expose a number of well known smells and design flaws: method chains, law of demeter violations, intensive and dispersed coupling and violations of the command query separation. These problems in design are a trade-off that one should be aware of, when deciding to use or even implement such an embedded domain specific language.

5.3.2 *Nested functions*

Another approach to implement an embedded domain specific language is the use of nested functions, often in conjunction with method chaining [Fow07]. This approach may suffer from the intensive coupling design flaw which is again the trade-off for higher readability and a level of abstraction close to the problem domain. An example of the nested functions approach is the assertion language of JUnit 4.5, build on top of the Hamcrest library. Instead of chaining method calls, the API is build in a way that functions can be stacked according to the domain specific languages’ grammar. This is shown in the following sample 5.3.

Listing 5.3: Simple example of nested functions in a DSL

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
class MyTest {
    @Test
    public void someTests() {
        assertThat(1, is(1));
        assertThat(1, is(not(0)));
        assertThat(Arrays.asList(1,2,3), hasItems(2,3));
    }
}
```

Again an expression builder, the class `Assert`, is the entry point to build the sentences valid according to the grammar. In Java the functions can be realized as static methods of the expression builder class that is statically imported. To ease the import, multiple functions of the assertion language are centralized in the classes:

1. `org.junit.matchers.JUnitMatchers`
2. `org.hamcrest.CoreMatchers`

Certain styles of unit testing may call a lot of methods from these two classes. As a consequence, the coupling dispersion with regard to the domain specific language is low, while the coupling intensity is high. As a result the intensive coupling design flaw may be present.

5.3.3 *Examining the use of Google Guice 2.0*

It has been shown, that several smells are inevitable when using an embedded domain specific language. They are also found in real world projects, as a small investigation of a project of SOPTIM AG shows. The project uses the Google Guice framework to wire-up concrete service implementations and abstract service contracts for injection into other services to achieve loose coupling between all components of the system. This is also called “dependency injection” [Mar08, page 157]. The `XyModule` in sample 5.4 configures one module of the system, depending on the runtime configuration of the system. Several smells and design flaws are detected in the sample; however, they are a trade-off for better readability and declarative dependency configuration provided by the Google Guice framework.

Listing 5.4: Usage of Binding DSL of Google Guice in a real project at SOPTIM AG

```
class XyModule extends AbstractModule {
    /* ... */

    private void bindSimulation() {
        bind(String.class)
            .annotatedWith(Names.named("xmlFileData"))
            .toInstance(SimulationStarter.DATA_XML);

        if (ArrayUtils.contains(args, "-AutoRefresh")) {
            LOG.info("++AUTOREFRESH++");
            Data data = SimulationStarter.generateData();
```



```
        SimulationStarter.writeDataToXml(data,
                                         SimulationStarter.DATA_XML);
    }

    bind(XyService.class).to(XyServiceImplSimulation.class);
}
}
```

The following design flaws can be detected in the method `bindSimulation()`:

1. Method chains
2. Law of demeter violation
3. Dispersed coupling

The method chains and law of demeter violations are an obvious and direct consequence of using the embedded domain specific language to configure the framework. A more subtle design flaw is the presence of dispersed coupling (see also figure 5.2). The method `bindSimulation()` has a coupling intensity of nine called distinct methods. The coupling dispersion is 0.77. This is a perfect match for the dispersed coupling detection strategy. Within the project, this result is however viewed as a false positive.

Most method calls and called types that contribute to these detected design flaws are part of the used domain specific language. Five out of the nine called methods and four of the seven called types are part of it. As the smells are an expected trade-off when using this domain specific language, excluding them is a reasonable modification. The coupling intensity drops to four while the coupling dispersion decreases only to 0.75. Because of the reduced coupling intensity below the small memory capacity, this is no more a match for the dispersed coupling detection strategy. Also method chains and violations of the law of demeter do not remain.

The described design flaws and smells of embedded domain specific languages or not only theoretically present, but also found in a real world project. As the problems are a well known trade-off when using such domain specific languages, the exclusion of participating methods and types looks promising to reduce false positives.

5.4 APPLICATION PROGRAMMING INTERFACE DESIGN

The development and design of an application programming interface (API) especially takes interface stability, versioning and change management of the interface into

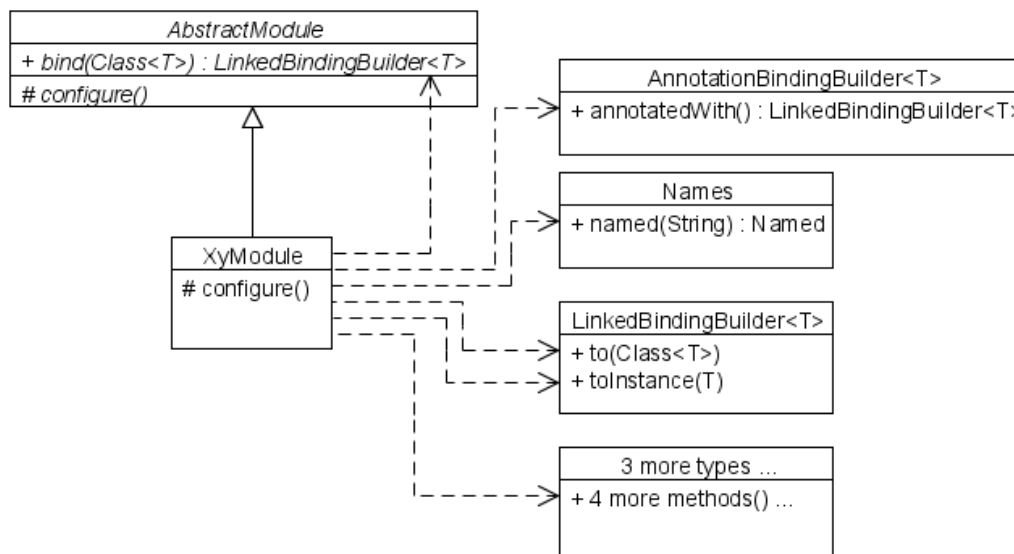


Figure 5.2: Classdiagram of dependencies in sample 5.4

account. As soon as an API is published it is very hard to change it. It is only possible to add or remove parts of it. Removing parts of the API is done by declaring this change in advance, i.e. “deprecating” the to-be-deleted element of the API. Whenever such an API is also used within the same project, e.g. in the Eclipse Project, the widespread use of such a component is subject of certain detected smells that do not take the explicit stability into account.

Robert C. Martin defined the instability metric to express the relative amount of work required to make a required change [MM06, page 426]. The external clients of an API are unknown and not under the control of the API developers. Thus it is hard to make a required change and the instability of an API is very low. Marinescu defines a detection strategy for the shotgun surgery smell, to find locations where a change has significant impact onto the system [Mar02, page 71]. As an API is used by many clients within a project, it is used by many types and methods expressed in the changing classes and changing methods metrics. An example (5.5) is the “Term API” in the Cultivate Project to construct terms out of atoms and more terms. The involved methods are subject of detected shotgun surgery in the system. However, the stability is explicitly stated as high, independent of the measures of Robert C Martin. Possibly the API should be extracted into its own class, to clearly separate between domain code and API code in the given sample.

All explicitly stated application programming interfaces may be subject of the shotgun surgery smell. However, their structure and special position within the system

take care⁶ of the problems related to shotgun surgery. The detection of shotgun surgery is thus expected and considered a false positive.

Listing 5.5: Term API in Cultivate Project

```
public final class Term {
    /**
     * Creates an atom term (a term that has no child terms).
     */
    public static Term atom(String name) {
        return new Term(name);
    }

    /**
     * creates a prolog term with an arbitrary number of
     */
    public static Term term(String name, Term... children) {
        return new Term(name, children);
    }

    /* ... */
}
```

5.5 PRESENTATION LAYERS

Classes in the presentation layer of an enterprise application are exposing several smells and design flaws. Some classes, typically the views, are tightly coupled with the underlying user interface framework while others, typically the controller classes, have a strong coupling to views and the underlying model. Some architectures, e.g. the Cultivate Project and several projects at SOPTIM AG, make use of an additional separation between the domain model and the (presentation) model used in the presentation layer. Typical problems are feature envy, dataclasses as well as intensive and dispersed coupling.

Typical architectures in presentation layers make use of the “Model View Controller” pattern [BMR⁺96, page 125] or one of its several variants like “Model View Presenter” [ACE⁺06] or “Model View ViewModel” [Wik09]. All of them are based on an idea dating back to the Smalltalk platform in the 1970s. The idea says that model, view

⁶e.g. with the help of Eclipse PDE API Tools: <http://www.eclipse.org/pde/pde-api-tools/> monitoring the development of an API and all changes done to it

and controller should be separated to be able to evolve and maintain them separately. Views that are written in the host language, i.e. not in any external domain specific language such as XHTML, Java Server Faces⁷, etc., are necessarily coupled with the underlying user interface framework. Flaws like intensive coupling and dispersed coupling are inevitable.

Controller classes are coupled with the view and the model. Depending on the chosen pattern for user interface architecture, feature envy may be present. In the Model View Presenter variant, the Presenter class mediates between the view and the model. This is a case where feature envy is likely while the patterns Model View Controller (MVC) and Model View ViewModel (MVVM) are unlikely to expose this flaw. The View of MVC has access to the model, so the Controller does not necessarily access all the data of the model. The pattern MVVM uses the databinding technology of the underlying user interface framework and delegates all mediation of data to the underlying framework.

Typically, the model of the user interface architecture is not the same as the domain model. In this case, the model of the user interface is likely to become a dataclass. Its sole responsibility is the transfer of data between the presentation layer (and views) and the services of domain model and the application layer.

Several smells and design flaws related to coupling are typical for current user interface architectures. As such they are considered as false positives and have little additional meaning to a project that choose the cited patterns for user interface architecture.

5.6 UNITTESTS

Unittesting reveals a strong relation between tests and the units under test. This coupling is subject of the design flaws intensive coupling, dispersed coupling and feature envy. Further, these relations contribute to the detection of shotgun surgery in the unit under test.

Good unittests should be readable, which means having clarity and simplicity. It is also recommended to have only one assertion per test [Mar08, pages 124 and 130]. Still, complex scenarios for a test require multiple interactions with a unit under test. The number of called methods may exceed the limits set in the intensive coupling detection strategy and when multiple types are involved also the limits of the dispersed coupling detection strategy. However, as long as the test has clarity and is readable,

⁷<http://java.sun.com/javaee/jaserverfaces> accessed on 10.11.2009

these multiple couplings will not do any harm. As such the detected intensive coupling and dispersed coupling is a false positive.

On the side of the unit under test, the before mentioned couplings may exceed the limits of the shotgun surgery detection strategy. As more tests enter the system, the number of calling methods and calling classes increases and possibly goes beyond manageable limits for non-API classes. The coupling introduced by unittests is, however, inevitable. As a consequence they do not contribute to the real intent of the shotgun surgery smell.

Three design flaws are directly associated with unittesting. Within the unittest itself, intensive coupling and dispersed coupling are inevitable for complex szenarios. These unittests contribute at the same time to potential shotgun surgery in the units under test.

5.7 SUMMARY OF THE STRUCTURE-SMELL-TRADEOFF

Structures can be defined in terms of roles and interactions and relations between those roles [GHJV95] [Rie00a]. Important relations include “uses”, “used by” and “forwarding” relationships. This chapter has shown that several structures contribute to the existence of smells. Subsets of their roles and relations are the root cause of detected smells. These subsets are used in the following chapter (see 6.2) to formalize and model the effect of structures on the existence of smells.

The following table (5.1) summarizes all discussed structures and related smells, giving a complete overview of the results.

Structure	Role	Relation	Possible Smells
Decorator Pattern	Abstract Decorate Concrete Decorator	forwards inherits	Middle Man Refused Parent Bequest
Composite Pattern	Component	inherits	Refused Parent Bequest
Visitor Pattern	Concrete Visitor Element	uses uses	Feature Envy Dataclass
Strategy Pattern	Concrete Strategy	uses	Feature Envy
Facade Pattern	Facade Facade Facade	forwards uses uses	Middle Man Intensive Coupling Dispersed Coupling
Adapter Pattern	Adapter	forwards	Middle Man
Mediator Pattern	(simple) Mediator (complex) Mediator (complex) Mediator	forwards uses uses	Middle Man Intensive Coupling Dispersed Coupling
Creation Methods	Creation Method	uses	Feature Envy
Procedural Components	Service Datastructure	uses uses	Feature Envy Dataclass
Embedded DSL	Client Client Client Client	 uses uses	Method Chain LoD Violation Intensive Coupling Dispersed Coupling
APIs	API methods	used by	Shotgun Surgery
Model View Presenter	Presenter View View	uses uses uses	Feature Envy Intensive Coupling Dispersed Coupling
Unittests	Test Test Test	uses uses uses	Feature Envy Intensive Coupling Dispersed Coupling

Table 5.1: Overview of Structures and Smells

APPROACH: SMELL DETECTION IN CONTEXT

In the previous chapters, the link between structures and smells has been drawn theoretically and requirements for efficient smell detection have been set. The following chapter describes an approach to meet these requirements and exploit the nature of structures to reduce false positives. In order to achieve an efficient smell detection several forms of context are used.

Smells and design flaws are detected by implementations of Marinescu's detection strategy approach. The approach is implemented as an addon to the static analysis system "Cultivate". Detection strategies, the underlying metrics and structural detection are implemented with logic meta-programming.

These detection strategies and metrics are enriched with a **structural context**. The structural context describes well defined structures to reduce false positives. Descriptions of such structures include their relation and influence to smells and their detection.

The **relevance context** reduces the amount of presented information and informs the user only about smells that are relevant to the user's current task. This prevents an information overload of the user. The relevance context considers the relevance of elements in a programming task or compared to the history of the whole system.

An **interlinking context** is used to present the context of a smell instance. This includes relations between smells and relations between elements relevant to a programming task. The interlinking context should help the user of this approach to understand a complex design problem more easily.

6.1 IMPROVING STATE OF THE ART IN CULTIVATE

6.1.1 *Architecture of Cultivate*

Cultivate is a platform developed on top of the Eclipse Platform¹ that integrates natively with the Java Development Tools (JDT)² inside the Eclipse IDE. Cultivate provides the concepts of a metric and a smell detector together with automatic caching

¹<http://www.eclipse.org/platform/> accessed on 28.12.2009

²<http://www.eclipse.org/jdt/> accessed on 28.12.2009

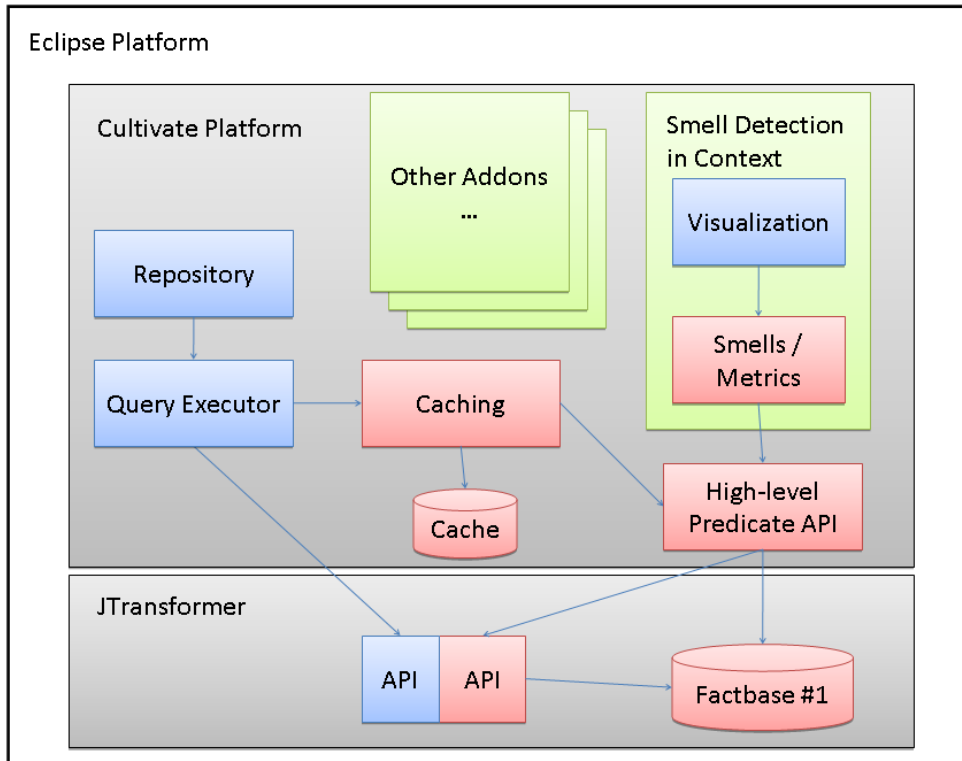


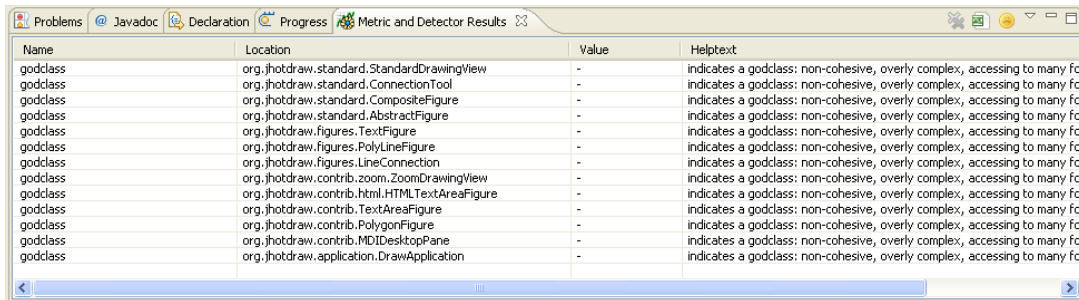
Figure 6.1: Architecture of Cultivate. Green boxes are Cultivate Addons, blue boxes are components written in Java, red boxes are written in Prolog.

and computation, as well as means to present results in the result view and as markers inside the Eclipse IDE in the Java Editor and lists of warnings. Further it provides an API to draw diagrams based on metric results.

The computations are based upon the abstract syntax tree (AST) model of JTransformer. JTransformer is a plugin for Eclipse to build a model of the AST from sourcecode. This model is represented by logical facts inside the Prolog language. The set of these facts is called a “factbase” (see figure 6.1). JTransformer provides an application programming interface (API) to query and transform factbases [KHR07]. This API is provided as a set of prolog predicates and Java classes. Thus prolog is used as a database for the AST model and as a language to work on this model.

All computations in Cultivate are realised as predicates written in the Prolog language. The predicates:

- `smell_(SmellName, Element)`
- `metric_(MetricName, Element, Value)`
- `query_scope_(QueryName, Scope)`
- `query_description(QueryName, Text, RangeAndTheshold)`



Name	Location	Value	HelpText
godclass	org.jhotdraw.standard.StandardDrawingView	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.standard.ConnectionTool	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.standard.CompositeFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.standard.AbstractFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.figures.TextFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.figures.PolyLineFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.figures.LineConnection	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.contrib.zoom.ZoomDrawingView	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.contrib.html.HTMLTextAreaFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.contrib.TextAreaFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.contrib.PolygonFigure	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.contrib.MDIDesktopPane	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc
godclass	org.jhotdraw.application.DrawApplication	-	indicates a godclass: non-cohesive, overly complex, accessing to many fc

Figure 6.2: Result View of Cultivate, showing Godclass Smells

are used to define smell detection and metric computations. Further attributes like a description, a range and default threshold for metrics and the scope of the element (package, class, method, etc.) can also be specified. These smells and metric computations are queried with the predicates “smell” and “metric” that handle automatically the caching of computed results. Smell detectors and metric computations are defined within so called “Addons” that plug into the Cultivate Platform (green boxes in figure 6.1). These addons are realized as Eclipse Plugins, so Eclipse takes care of managing them technically (loading, starting them, etc.). Cultivate provides a high level API to easily describe metric computations and smell detectors. This high level API is based upon the JTransformer prolog API and used the created factbase.

All visualizations and presentations of computed results are written in the Java language. The classes `BaseQuery`, `MetricQuery` and `SmellQuery` provide abstractions for the querying of arbitrary queries (like dependency queries), metrics and smells within the visualizations. A blackboard architecture together with a publisher-subscriber notification is used to start computations and notify interested parties like diagrams, the result view and markers of the computed results. The responsible component is the `Repository`. The `Repository` also provides a second level caching for the results to minimize roundtrips to prolog. All executions of queries prolog are delegated to a `QueryExecutor`, that uses the JTransformer API to issue the queries to prolog and the prolog caching component of Cultivate.

Further the Cultivate Platform provides the already described `Repository` and also a `RepositoryJob` to run computations in the background. This component is invoked whenever the factbase in JTransformer is changed after a build has occurred inside the Eclipse IDE. The `RepositoryJob` clears all caches and recomputes metrics and smells that have subscribed clients. After recomputing them, the results are published to the subscribers by the `Repository`.

The “Generic UI” component is also part of the Cultivate Platform. It provides basic visualizations of metrics and smells with the “Result View” and Eclipse markers.

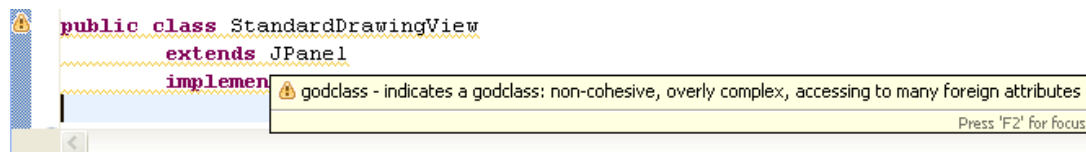


Figure 6.3: Cultivate using Eclipse Markers to indicate a Godclass

The result view is a tabular presentation of detected smells and computed metric values. In figure 6.2 the result view is showing detected godclasses of the JHotDraw 6 framework. The table of the result view presents the name of the metric or smell, the corresponding location and a helptext. The column “value” is used to present values of computed metrics. The description given by the smell definitions is used as helptext. Each row represents one detected instance of a smell or a location where a metric was computed. A double click with the mouse on a row opens the Java editor of Eclipse and navigates to the corresponding location. The toolbar on the right hand in figure 6.2 provides several actions. The first one (grey cross) disables the detections or computation of all smells and metrics. The second action (document of a common office application) exports the results shown in the table into a “Comma Separated Values” (CSV)³ file. The action triggered by the button with the yellow smily enables a default set of smells to detect. The triangle symbol next to this action provides the menu of the result view. This menu allows the activation and deactivation of specific smells and metrics for detection and computation.

Cultivate adds also Eclipse markers at the locations of smells. The user can configure which smells are used to create such markers. Figure 6.3 shows a “warnings” marker in a file of JHotDraw. Eclipse adds a warning symbol at the bar on the left side of the editors. Further the location is underlined in yellow. When the user points at the marker with the mouse, a description of the marker is shown. This description is set by Cultivate. Again the description of the corresponding smell definition is used.

Upon the Cultivate Platform, domain specific addons can be build (green boxes in figure 6.1). These are realized as Eclipse Plugins. An addon can extend the Cultivate Platform with metric and smell definitions, visualizations and other related features. To provide these extensions, the addons depend the components of the Cultivate Platform and make use of the high level Prolog API.

Several addons exist already. They provide analysis and visualization of structures, the analysis of concepts, method refactorings, architecture checks and more. The described approach “Smell Detection in Context” is also realized as an addon to the Cultivate Platform.

³http://en.wikipedia.org/wiki/Comma-separated_values accessed on 06.01.2010

6.1.2 *Architecture of Smell Detection in Context*

The prototype of this thesis consists of a “Histogram Addon” and the “Smell Detection in Context Addon”. The Smell Detection in Context Addon uses Histogram Addon to build the temporal context (green boxes in figure 6.4).

The Histogram Addon contributes a model that represents historical information about an analysed project. The addon is shown in the bottom left corner of figure 6.4. Based upon this model, temporal and historic metrics can be computed. The design of this component is described in further detail in section 6.4.

The Smell Detection in Context Addon provides several contexts to enrich classic smell detection. A task context, also called relevance context, a temporal context and an interlinking context. The task and temporal context extend the result view of the Generic UI Addon (figure 6.4). Both provide special filters to focus the querying of smells and metrics on a set elements. These filters are plugged into the result view. The interlinking context is visualized by the “Smell Context View”. This view uses the task context to focus itself (top right corner of figure 6.4). The smell detection is implemented with detection strategies. These are defined in prolog and use metrics that are also defined in prolog. All, but metrics of cyclomatic complexity and number of statements, are part of the Addon. Metrics for cyclomatic complexity and number of statements are part of the Cultivate Platform. The metric and smell definitions in the Smell Detection in Context Addon are enriched with structural information, that is also available als prolog predicates. These structure definitions and the metric definitions make use of the high level API of Cultivate (figure 6.4).

6.1.3 *Improving Detection Strategies*

Marinescu bases detection strategies solely upon metrics and thresholds. However, for some smells and design flaws, this approach can still only compute the symptoms and not the disease - to phrase it with Marinescu’s words.

Moha et al propose the use of structural properties in addition to metrics [MGL06]. The law of demeter [Lie96] characterizes allowed method calls from an object’s point of view. Some violations of this principle are also variants of the “Message Chains” smell. A violation of the law of demeter may indirectly lead to a higher outgoing coupling. While the latter can be measured by coupling metrics, the presence of chained method invocations, and other forbidden methods calls, can only be detected by exploiting the structure of the method body.

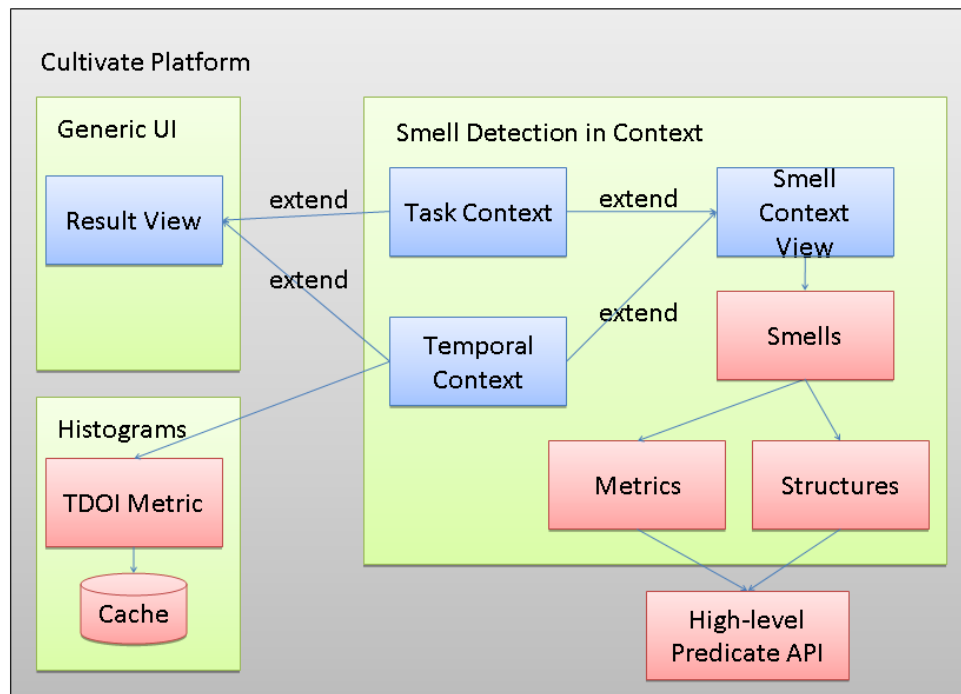


Figure 6.4: Architecture of Smell Detection in Context

The architecture of Cultivate makes it possible to implement detection strategies based upon metrics and also based upon structural detections. Both can also be combined in a single smell detector. The message chains detection strategy implemented in the Smell Detection Addon makes use of this as well as the law of demeter detection strategy that is already part of Cultivate.

6.2 STRUCTURE CONTEXT TO REDUCE FALSE POSITIVES

The structural context uses well defined structures to reduce false positives during the detection of smells. Descriptions of such structures consists of the different roles and relations between them. These descriptions also include the relationship of structures to smells and their influence on the detection. Further, roles can be directly attached to smells.

The structural context hooks into the querying of serveral relations and detection strategies. This is done to recognize smells that are introduced by relations or roles of well defined structures. These smells are classified as false positives and removed from result output.

To be able to check, if a relation or a smell participates in a structure, these structures have to be defined. The prototype provides the means to define roles, attached smells and relations that contribute to smells. Several structures are already declared in

the prototype. Among them are embedded domain specific languages, application programming interfaces and the patterns Decorator, Visitor and Strategy.

6.2.1 Friend Relations - A Model for Structures and Smells

The approach “Smell Detection in Context” provides concepts to model the relationship between structures and smell. These models are expressed as logical facts within the Prolog language. Structures are modelled of roles and relations between them. The relations are used within the computation of metrics for detection strategies, while roles are used within the computation of a detection strategy to identify a smell.

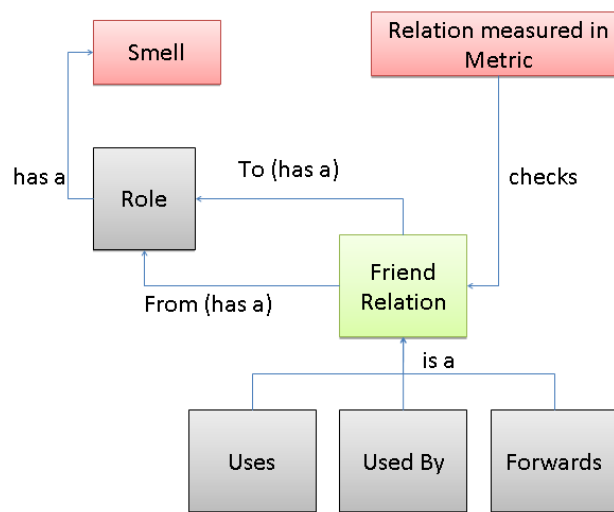


Figure 6.5: Model of Structures

Roles may exhibit a smell and thus the model allows a structure to define roles that are directly related to smells. These “friend smells” can be queried directly within the detection strategy. Friend smells make especially sense for design flaws in the identity category. Smells like dataclass are candidates for friend smells, because no relation between roles (and classes) is involved in their definition.

Listing 6.1: Structure definition for embedded DSLs and APIs

```

structure(embedded_dsl) .
role_uses(embedded_dsl, dsl_client_method, embedded_dsl_method) .

structure(api) .
structure_inevitably_has_smell(api, api_element, shotgun_surgery) .

```

The listing 6.1 shows the definition of the structure *embedded_dsl*. The structure consists of methods that are part of an embedded DSL and methods that use an

embedded DSL. This “uses” relation is expressed with the `role_uses` fact. The listing also shows the definition of the *api* structure. Any element that is declared as an element of a published, controlled API is subject to shotgun surgery. This fact is represented by the `structure_inevitably_has_smell` fact.

The relations “uses”, “used by” and “forwards to” are used to define so called “friend relations”. These are relations that are introduced into a system by a well defined structure. Such friend relations should not negatively contribute to the detection of a smell. These three relations check if the participating elements are part of the same structure and if a friend relation is defined between them. Metrics and detection strategies that make use of the “uses” / “used by” and “forwards” relation exclude friend relations this way from their detection.

Listing 6.2: Uses relation and friend uses constraint

```
uses(method, attribute, MethodId, AccessedAttribute) :-
    . . . . ,
    not(friend_usage(method, attribute, Method, AccessesAttribute)).

friend_usage(method, attribute, FromMethod, ToAttribute) :-
    structural:role(Structure, UsingRole, Instance, FromMethod),
    structural:role(Structure, UsedRole, Instance, ToAttribute),
    role_uses(Structure, UsingRole, UsedRole).
```

The fact *structural:role (Structure, Role, StructInstance, RoleInstance)* is attached to all elements that are part of a described structure. Several instances of a structure are distinguished with the *StructInstance* argument. The role is attached to an element with identifier *RoleInstance*. These facts can be provided by a design pattern detection tool or any other method to annotate structures in sourcecode. The only requirement for extension is that these tools have to provide the Prolog facts in the described form.

6.2.2 Providing Instances of Structures

To study several examples of structures and to verify the hypothesis within a case study, a simple mechanism to annotate sourcecode with structures was needed. The prototype provides a component called “Structural Annotation Processor”, that creates the *structural:role* facts for a system.

The processor is not a structure detection tool nor a real pattern detection tool. It simply creates a tree of roles, starting from an entrypoint that the user defines. These user provided entry points are hints to locations of structures. The user defines these

hints either in terms of full qualified names of the instances itself or with a full qualified name of a Java annotation that marks all instances.

Listing 6.3: Hinting the system at instances of a structure

```
annotation_role(controlled_api, api_element,
    'org.cs3.cultivate.core.PublicApi').
fqn_role(embedded_dsl, embedded_dsl_package,
    'com.google.common.inject').
fqn_role(embedded_dsl, embedded_dsl_package,
    'com.google.common.inject.binder').
```

The sample 6.3 shows the hints for the embedded domain specific language in “Google Guice”⁴ and an annotation that marks the controlled API in the component “Cultivate Core”.

The structure definition within the prototype does not only provide roles and relations but also rules how to build the tree of derived roles, starting from the entrypoints that the user provides. The sample 6.4 describes a rule to create the derived role *embedded dsl type* from the role *embedded dsl package* based on the variables for instance, parent and the new child (*DslType*). The rule to derive a new role is based on the high-level predicate API that is part of the Cultivate Platform.

Listing 6.4: Rule to build a derived role in the embedded dsl structure

```
%type in embedded_dsl_package is a dsl type
process_leaf_(embedded_dsl, embedded_dsl_type,
    embedded_dsl_package,
    Instance, Parent, DslType, (
    package_contains_type(Instance, DslType)
    )).
```

The structural annotation processor starts with the hints given in listing 6.3. The user of the prototype has specified that the packages “com.google.common.inject” and “com.google.common.inject.binder” contain an embedded dsl. Figure 6.6 illustrates the situation. First the hints are used to annotate both packages as *embedded dsl packages*. Next, the rule presented in listing 6.4 is processed and all types in both packages are annotated as *embedded dsl types*.

In the sample shown in figure 6.6, only the types “AbstractModule” and “Linked-BindingBuilder” are presented. Both provide the minimal functionality of the embedded domain specific language in Google Guice. Similar to the derived rule in listing 6.4, further rules exist. These rules are used to derive annotations for *embedded dsl*

⁴<http://code.google.com/p/google-guice/> accessed on 05.01.2010

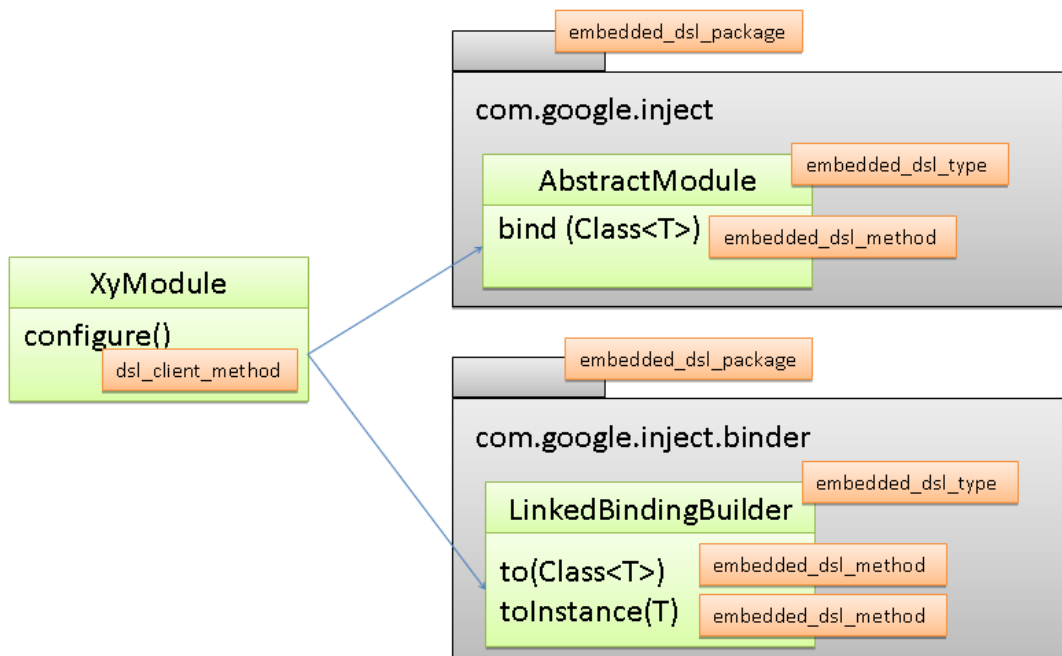


Figure 6.6: Extract of the Google Guice DSL with Structural Annotations

method and *dsl client method*. All methods in types of the DSL are annotated, as shown in the figure. Based on these annotations, all methods using these *embedded dsl methods* are annotated as *dsl client methods*.

This flexible approach is used in the prototype of this thesis. Implemented sample structures contain embedded domain specific languages, managed application programming interfaces and the patterns: strategy, decorator and visitor. During the case study presented in chapter 7, further structures were developed. These model APIs of the standard Java Development Kit⁵ and iText⁶, which is a framework to generate Adobe PDF documents.

The effects of the structural context have been validated against commonly analyzed projects. These include JRefactory and JHotDraw. In the following, the results of this validation is reported.

6.2.3 Case: Visitor Pattern in JRefactory 2.6.24

JRefactory⁷ 2.6.24 contains a large instance of the visitor pattern: the “SummaryVisitor” hierachy. One of its largest concrete visitors, the class “PrintVisitor”, is complex

⁵http://en.wikipedia.org/w/index.php?title=Java_Development_Kit&oldid=335764961 accessed on 05.01.2010

⁶<http://itextpdf.com/> accessed on 05.01.2010

⁷<http://jrefactory.sourceforge.net/> accessed on 24.11.2009

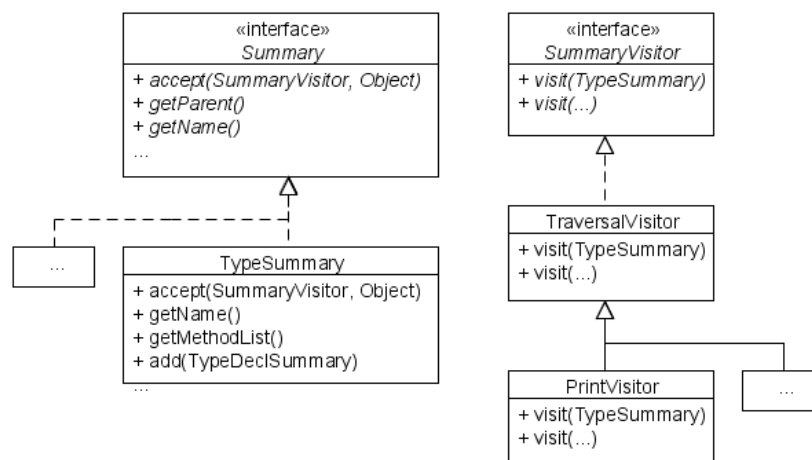


Figure 6.7: SummaryVisitor in JRefractory 2.6.24

enough to study it and is a common implementation of the visitor pattern for an investigation with regards to smells.

JRefractory is a tool that provides refactorings for the Java language. Further, it has components for code pretty printing, uml diagramming, duplication analysis, and metric computations.

The whole pattern instance “SummaryVisitor” consists of 13 concrete visitors in a visitor hierarchy with a depth up to five. This visitor instance operates on the Summary hierarchy. The PrintVisitor class is among the most complex concrete visitors within the hierarchy. Two of its methods visiting elements in the Summary datastructure expose the feature envy smell.

One of the envied datastructures is the TypeSummary class. This class is part of the Summary hierarchy. It is envied by the method `visit(TypeSummary, Object)`. Envied attributes of the TypeSummary class are its name, if it represents an interface and its parent. The visitor method does not use any local attributes, i.e. attributes of the concrete visitor. There are three accesses to foreign data. The locality of attribute usage is zero and the used foreign data is provided by only one class. Thus the method exposes feature envy.

The method `visit(ImportSummary, Object)` envies the classes `ImportSummary` and `PackageSummary`. It envies the type and package attribute of the `ImportSummary` and the name attribute of the `PackageSummary` class. No local attributes are used. Because three foreign attributes and no local ones are used while the foreign data is provided by two classes, this method has feature envy.

Both visiting methods, for `TypeSummary` and `ImportSummary`, have instances of the “uses” relation between methods in concrete visitors and attributes of the visited

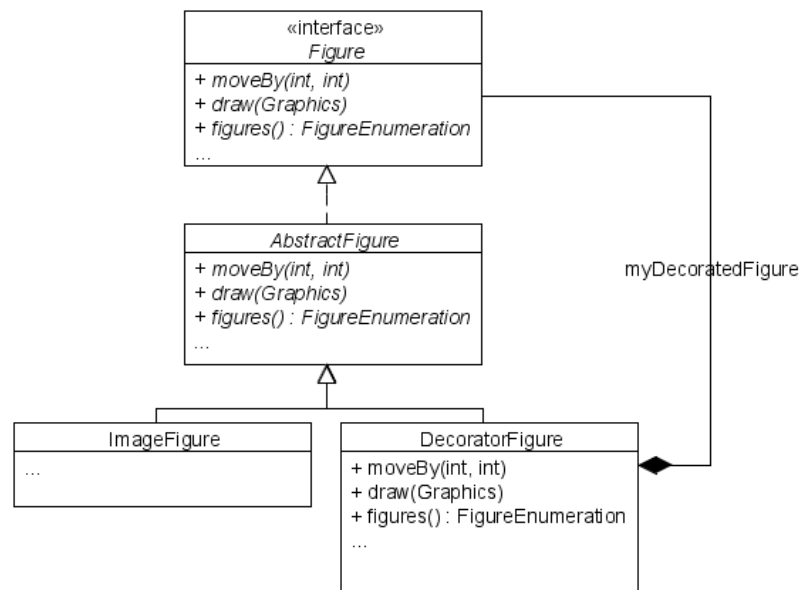


Figure 6.8: DecoratorFigure in JHotdraw

structures. As already stated, the feature envy smell is a known trade-off that is part of the visitor pattern. All envied attributes are part of the overall visited structure. No other attributes are envied. Both identified methods are false positives.

The relation between concrete visitors and visited elements is expressed in the *visitor pattern* structure. More precisely, a “uses” relation is defined between methods of a concrete visitor and attributes of visited elements. This definition results in the friend relation “uses”. It modifies the results of the computed metrics that are part of the feature envy detection strategy. The usages of attributes inside both described methods are checked for using attributes of the visited elements. Such usages are an inherent part of the pattern and thus not taken into account during the computation of the involved metrics. As a consequence the number of accesses to foreign data is zero and the false positive is removed.

6.2.4 Case: Decorator Pattern in JHotDraw 6

JHotDraw⁸ is a GUI framework for structured Graphics that was originally developed as an exercise by Erich Gamma and Thomas Eggenschwiler. Inside JHotdraw, the concept of a figure is a central one [Rie00a]. Some of the concepts involved with figures are using the Decorator and Composite pattern. One decorator of JHotDraw 6, SVN Revision 267, is subject of this study.

⁸<http://www.jhotdraw.org/> accessed on 26.11.2009

The figure concept is expressed in the interface named “Figure” supplying 42 different operations in the public interface. The abstract class “AbstractFigure” provides default implementations for the “Figure” interface. The whole hierarchy consists of 51 classes and the Figure interface on top. There are three classes that dynamically extend a Figure, implemented as decorators: “AnimationDecorator”, “BorderDecorator” and “RefusingDecorator”. The responsibility of handling the technical aspects of the decorator pattern, e.g. forwarding, are focussed into an abstract decorator called “DecoratorFigure”.

The DecoratorFigure class is, at first sight, an instance of the middle man smell. Most default implementations of the AbstractFigure class are overridden by the DecoratorFigure class (37 methods). The majority of these 37 methods only forwards the method call to the decorated element. Ignoring the enclosing context, this majority (21 methods) cause the middle man smell in the whole class. Only 16 methods take more responsibility than forwarding.

All methods of the DecoratorFigure class that implement the behavior of the Figure interface are part of “forwards to” relations. These relations are defined by the Decorator pattern between the methods of the decorators and the methods of the decorated element, whether the decorator is abstract or concrete. Especially when the forwarding responsibility is pushed from concrete decorators into an abstract decorator, these forwarding relations contribute to the middle man smell.

The definition of the Decorator pattern as a structure inside Cultivate annotate these “forwards to” relations as a friend relation between the involved roles. The metric “number of delegating methods” works on the “forwards to” relation. By considering the friend relations as part of the “forwards to” relation between methods, the detection strategy for middle man does not detect the DecoratorFigure and the false positive is removed.

6.2.5 Case: Strategy Pattern in JHotDraw 6

JHotDraw also provides means to build editors for structured graphics. Figures can be connected to other figures with a special “ConnectionFigure”. The connection points on connected figures are located by different strategies, so called “Connectors”. The “ChopBoxConnector” is one interesting case among the concrete strategies of this strategy pattern instance.

The central operations provided to locate connection points on a figure are the methods `findStart(ConnectionFigure)` and `findEnd(...)`. The context

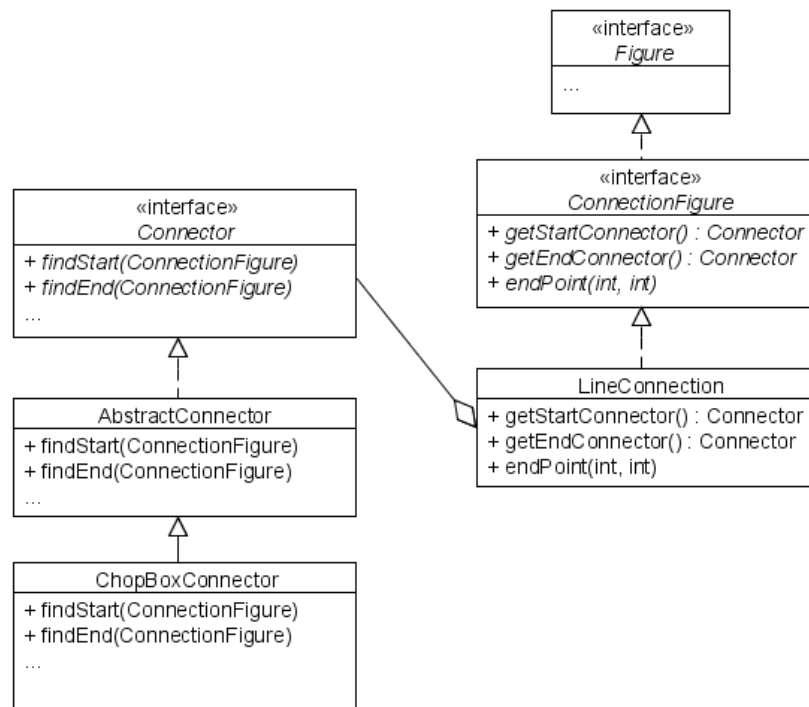


Figure 6.9: Strategy pattern in JHotdraw

of this strategy is the `ConnectionFigure` class. The `ChopBoxConnector` implements both operations in a way that they suffer from feature envy.

At first glance this envy arises from the access to the methods `getStartConnector()` and `getEndConnector()` of the `ConnectionFigure`. However, the `ConnectionFigure` itself is an interface and not a class. Thus it is not clear, whether it is decidable if these methods are accessor methods and thus represent an attribute.

Both methods still suffer from feature envy. The reason is the `Rectangle` object and its four attributes `x`, `y`, `width` and `height`. The methods `findStart` and `findEnd` both access all four attributes. As a consequence the number of accessed foreign data is four and the detection strategy for feature envy will catch both methods.

The structure of the strategy pattern defines a “uses” relation between the methods of a concrete strategy and the fields of its related context. Applying the structural context has no effect, as the context is not envied in this case.

6.2.6 Case: Public Api in Cultivate Core, Revision 2264

The Cultivate Platform contains a module that provides central functionality for the whole Platform, called “Cultivate Core”. This module defines for example the concept

of a prolog term and prolog atoms, which can be used to construct queries. Inside, two instances of the shotgun surgery smell are present.

A term has a name and possibly other terms as children. An atom is a named term without any further children, i.e. other terms. The functionality to construct a term out of other terms and atoms is part of the `Term` class. The creation methods are `term(Term...)` and `atom(String)`. Both return the term type. These methods are an integral part of the public application programming interface of the Cultivate Platform.

Still, both creation methods expose the shotgun surgery smell. All defined concrete query classes make use of them to construct their predicates for querying. The core of cultivate itself defines thirteen queries, thus the number of changing classes is thirteen. Every query has to define the method `getTerm()`. Thus the number of changing methods is also thirteen. The respective thresholds inside the definition of shotgun surgery is the short memory cap (Marinescu et al. use seven to eight [LM06]). As a result both creation methods are instances of shotgun surgery.

During the design of an application programming interface, versioning and change management are explicitly handled. These are locations where a change has significant impact onto the system. The developers are aware of the fact. The shotgun surgery smell is not a problem in this case but a sign for the usage of the application programming interface. Any change to it is handled with care and as a consequence, exceeding the shortmemory cap is not a problem at all.

6.3 RELEVANCE CONTEXT: TASK RELEVANCE

The relevance context provides context-sensitive detection of smells and scalable presentation of results. Kersten's concept of task context is used to model relevant smells in a programming task. This reduces the amount of files for detection, yielding less smells to detect and present. Only smells that are relevant in a programming task are detected.

6.3.1 *Task Relevance*

Murphy-Hill et al. claimed that scalable, context-sensitive smell detection is needed to support floss refactoring. Detection of smells has to focus on the current working task of a developer. Focussing detection presents relevant smell instances. Further focus prevents an approach to overload a developer with information.

Gail Murphy et al. showed in [MKRC05] that 90 percent of all changes done by developers involve more than one file. Despite modularity mechanisms of current languages and programming environments, developers touch elements across a code base to perform a required change [MKRC05]. Murphy et al. showed further that most changes involve less than seven file.

Smell Detection in Context uses the concept of a “Task Context” to focus smell detection on the current task of a programmer. The concept was introduced by Mik Kersten [Ker07, page 16]. The task context models information that a developer needs to complete a task. Every element within a task context has an attached weighting to define its relevance. This relevance is expressed by a “degree of interest” (DOI). All interactions of a developer with the integrated development environment are tracked and form an interaction history. The degree of interest is computed out of this interaction history [Ker07, page 16].

Interactions with an element increase its degree of interest in the current task context. Interactions with other elements decrease its degree of interest. An element is relevant if the degree of interest is positive. Elements with negative degree of interest are removed from the task context.

Smell Detection in Context uses Kersten’s task contexts to form a relevance context for smell detection. This relevance context focusses the detection on interesting elements in the current task context of a developer. Smells are detected only in those files that are part of the current task context. Smells are not detected globally on all elements or only on a single element, like classic approaches do. It makes the detection context-sensitive and follows the associated guideline (context-sensitivity) of Murphy-Hill et al. [MHB08b].

A sideeffect of the approach is an increase in performance. As Gail Murphy et al. showed, only a small set (one to seven) of files are part of most tasks. Smells have to be only detected in this reduces set of files.

The reduced set of files is also the measure how smell detection is made scalable in Smell Detection in Context. Murphy-Hill et al. stated that efficient smell detection should not overload the user with information [MHB08b]. The focused detection does also limit the result set that is presented to the user, i.e. developers. During the case study presented in this thesis in chapter 7, 1470 instances of eight different smells were detected. Using the relevance context focussing on two files, only six smell instances remain.

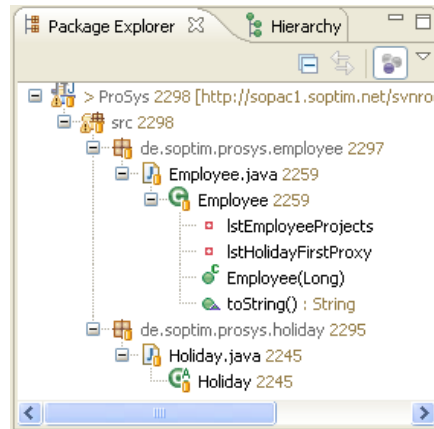


Figure 6.10: Focused Eclipse Package Explorer using Eclipse Mylyn

6.3.2 Integrating Cultivate with Eclipse Mylyn

In order to implement the relevance context in Cultivate, it is integrated with Eclipse Mylyn. The ideas of Kersten et al. are provided by the Eclipse Mylyn⁹ project and can be used by any plugin of the Eclipse Platform. Relevant files are extracted from the mylyn task context. Smell detection is performed on these relevant files.

Figure 6.10 shows the Eclipse Package Explorer for Java, focused on an active task context. Only elements with a positive degree of interest are relevant and shown in this view. Mylyn provides degrees of interest on all levels, from methods to packages of the Java language.

The Smell Detection in Context add-on for the Cultivate Platform provides a filter for the result view of Cultivate. The result view is part of the Cultivate Platform and was extended to support a filtering of the queries and the result set. The filter to focus on the current task can be triggered with a button that is added to the window of the result view. In figure 6.11, this button is activated and shown on the left of the toolbar.

Upon activation of the relevance context, the relevance context is added as a filter to the result view. It is also registered as a listener at Eclipse Mylyn. This listener is notified by Mylyn of all changes of a task context. Possible changes are the activation, deactivation and modification of a task context. A task context is modified, whenever an element is added or removed because of a change in its degree of interest.

The provided filter decorates all queries for subscription with a predicate to detect smells only for a given set of files. The queries for the result view are created by a factory. The standard factory for the result view creates `SmellQuery` and `MetricQuery` instances. The provided filter acts as such factory and creates `FileFilteredSmellQuery` (for metrics respectively). A `FileFilteredSmellQuery`

⁹<http://www.eclipse.org/mylyn/> accessed on 04.01.2010

Listing 6.6: Definition of File Filtering in Prolog

```

file_filtering(Path, Name, Participant) :-
    member(QueryType, [smell, metric, relation_smell, relation_metric]),
    query_scope(Name, QueryType, Scope), !,
    scoped_file_filtering(Path, Scope, Participant).

```

Figure 6.11 shows the detected smells for the task context shown in figure 6.10. The task context consists of the files “Holiday.java” and “Employee.java”. All smells are activated and detected.

6.4 RELEVANCE CONTEXT: TEMPORAL RELEVANCE

During a project, the development of a system causes an evolution of it. Not all artifacts are equally relevant to the project’s team. Design flaws that appear in artifacts without recent changes during the project evolution can be considered irrelevant. They become relevant again, if the artifact is changed again during the project’s evolution. An approach to detect smells in a project has to respect these different relevances.

The relevance context uses temporal relevance to model the historically based relevance of an element. This relevance is computed with respect to the complete history of the whole project.

6.4.1 *Temporal Relevance*

The approach of Ratiu et al. is a good step in the direction of historically relevant smells [RDGM04]. Their granularity of changes does leave out important, smaller, changes. These changes have to be considered to describe the relevance of an element in front of the history of a project.

The here presented temporal context is constructed to represent the relevance of an artifact in the current project evolution. The relevance is expressed as “temporal degree of interest” (TDOI) taking all changes of an artifact in the project’s source code management system into account.

In the temporal context the metric TDOI is used as an indicator of an artifact’s relevance. The temporal degree of interest is the adaption of Mik Kersten’s idea of “degree of interest” (DOI) to the area of source code management systems. Mik Kersten uses in [Ker07] the degree of interest to represent the relevance of an element during a programming task of a developer.

From two succeeding versions of an element, the change is computed as the difference of both versions. As shown in listing 6.7, all changes in the repository of a sourcecode management system (SCM) are interpreted as interaction events. Recent changes increase the temporal degree of interest while changes in other artifacts of the project decay the temporal degree of interest. Any interaction with elements having negative degree of interest causes a reset of its degree of interest.

Listing 6.7: TDOI algorithm

```
double doi(File file) {
    int start = minimumRevision(file);
    double interest = 0;
    for (Revision revision : eventsForElement(file)) {
        interest += 1;

        if (Interest < decay(start, revision)) {
            interest = 1;
            start = revision;
        }
    }
    return interest - decay(start, lastRevisionInProject()); //DOI
}
```

Stable artifacts are represented by low or even negative temporal degrees of interest. Instable artifacts have a high positive temporal degree of interest. All artifacts with a negative measurement are considered irrelevant.

This relevance is to be interpreted with the whole project in mind. Recently changed artifacts are the places where the project's current evolution takes place. They have a high relevance for the project's team. Other artifacts that did not change have a low relevance. The current evolution is expressed in the last revisions that have been stored in the SCM's repository. Recently changed artifacts, i.e. changes near this last revision, have a decay that is lower than the interest. Thus they have a positive TDOI measurement and are interpreted as relevant. Artifacts that have not been changed for many revisions up to the last project's revision will have a low TDOI, negative values are also possible. The TDOI algorithm ensures that small single changes (e.g. small bugfixes) have little impact on the relevance - the decay dominates here.

Frequent small changes, e.g. resulting from Shotgun Surgery, will add up to more relevance, decay will not dominate if frequency is high enough. Irrelevant artifacts have a negative measurement as decay dominates over the interest. In summary the

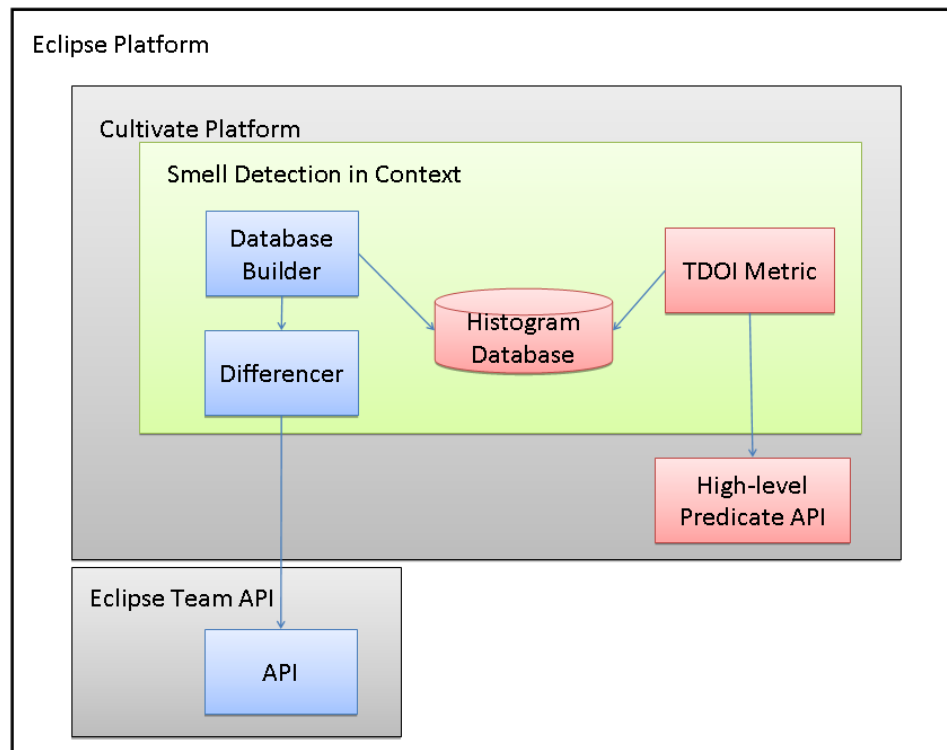


Figure 6.12: Architecture of Cultivate Histogram Component

temporal degree of interest represents the relevance of an artifact and tolerates small bugfixes.

6.4.2 Computing Temporal Degree of Interest

The temporal degree of interest is computed as a metric that is implemented in the Histogram component of this thesis' prototype.

The Histogram component is an addon to the Cultivate Platform. The addon that provides a model and a metric to represent the historic changes and temporal relevance of an artifact. It consists of the so called "Differencer", a "Database Builder", a "Histogram Cache" and the temporal degree of interest metric. The Differencer uses the Eclipse Team API¹⁰ to build change histograms for all files in a project (see figure 6.12). A change is computed of the difference of two succeeding versions of a file.

The Database Builder stores uses the Differencer to trigger missing computations and stores the results in the Histogram Cache. This cache is persistent and saved to disk when the Eclipse IDE shuts down. During startup the cache is loaded again for each project, if available.

¹⁰<http://www.eclipse.org/eclipse/platform-team/> accessed on 02.11.1009

A change is represented by a number of added, changed or deleted lines between two subsequent versions of a file. These differences are stored as prolog facts, using the following template:

Listing 6.8: Fact definitions for the histogram database in Prolog

```
addition(File, Revision, Author, Timestamp, AddedLines) .
change(File, Revision, Author, Timestamp, ChangedLines) .
deletion(File, Revision, Author, Timestamp, DeletedLines) .
```

The temporal degree of interest metric is implemented in prolog. The metric is exposed for use by other addons in the Cultivate Platform. The TDOI metric accesses change facts to compute the temporal degree of interest. Every interaction with a file, i.e. addition, change or deletion of lines, increases the artifact's interest. Interactions with other artifacts in the project result in a decay of the TDOI by a scaling factor (0.27 in the implementation). This constant ensures that the degree of interest decays fast enough for few changes.

The algorithm further ensures that whenever the DOI is negative, another interaction starts from zero again. It ensures that another interaction with an irrelevant artifact makes this artifact relevant again. The algorithm to compute temporal degree of interest uses all interaction events with the project's repository and computes a single value to represent the artifact's relevance considering the complete history of the project.

The Histogram Addon provides another filter for the result view of Cultivate. The filter decorates the queries of the result view with similar mechanisms as the filter for task relevance in section 6.3.2. This temporal filter acts as a feactory for smell and metric queries. The `SmellQuery` is decorated by instances of `TemporalFilteredSmellQuery`.

The decorator uses the predicate `temporal_filter` to apply a filtering based on the before explained temporal relevance. The predicate is given in listing 6.9 as a simplified, but working, version. First, the original smell detection is computed. If the instance is an internal type, package or method or a field of an internal type, the temporal degree of interest is computed. Internal elements are defined by sourcecode of the analysed project, in contrast to external elements provided by frameworks or libraries in binary form. Only relevant elements are passing this filter. As no history information is available for external elements, those elements also pass the filter.

Listing 6.9: Definition of Temporal Filtering for Smells in Prolog

```
temporal_filter(smell(Name, Participant)) :-
    smell(Name, Participant),
    (
```

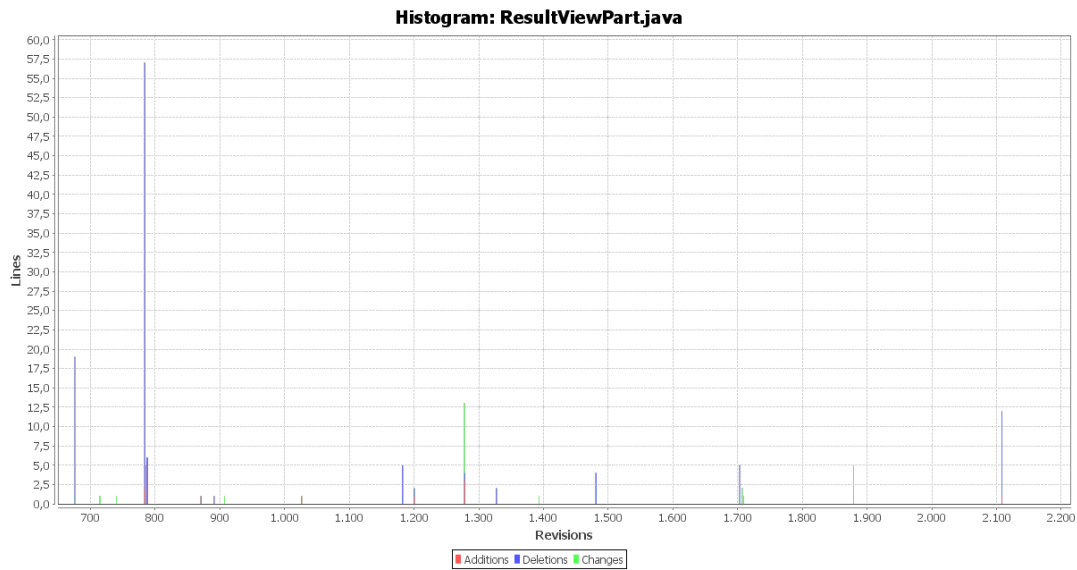


Figure 6.13: Change Histogram of class ResultViewPart

```

(internal_type(Participant) ,
 metric(doi, Participant, DOI) ,
 DOI >= 0)
;
not(internal_type(Participant))
) .

```

6.4.3 Case

Two cases of the Cultivate project itself illustrate the temporal degree of interest for a stable artifact and an artifact under current development. The development of the Cultivate Platform is currently at revision 2108. Recent evolution has taken place around the results view feature. Problems related to it are relevant to the current evolution of the platform. The class “ResultViewPart”, responsible for the presentation logic of this feature, has 2.02 as temporal degree of interest, because recently it has been changed very often (see figure 6.13). The diagram drawing feature has already evolved into stability. Potential problems are not relevant to the current evolution of the platform. The class “AbstractDiagramConfiguration” has a TDOI value of -3.75 and is thus not considered for smell detection in context. (see figure 6.14).

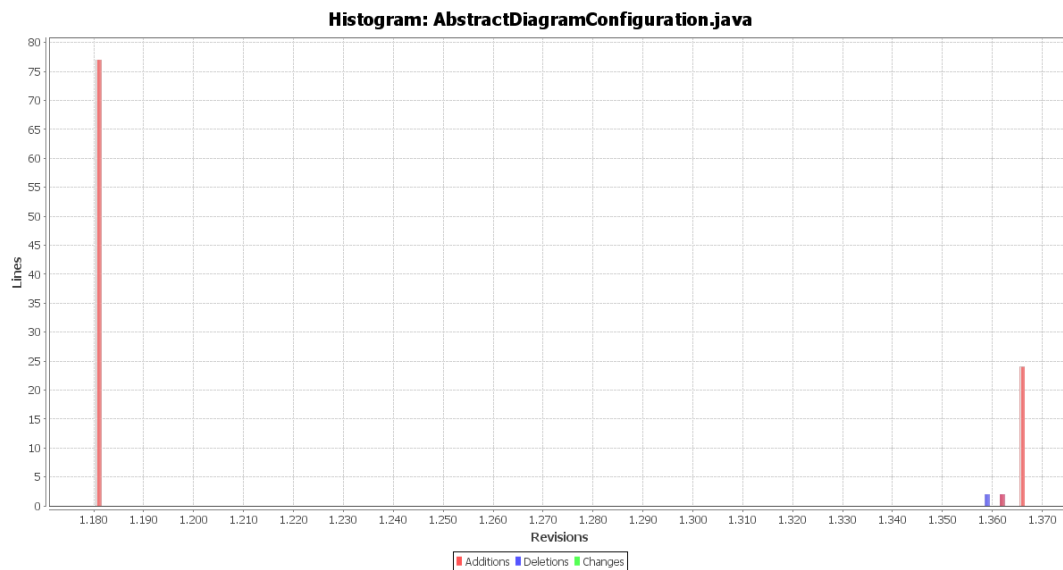


Figure 6.14: Change Histogram of class AbstractDiagramConfiguration

6.5 INTERLINKING CONTEXT

Smells are mostly not isolated problems [WL08]. Research shows that instances of a smell are accompanied by other smells. These smells form relationships [MVL03]. The interlinking context expresses these relationships as context-sensitive graph. Important details to understand presented smells are also presented. The interlinking context should help the user of this approach to understand a complex design problem more easily.

6.5.1 Interlinking of Design Flaws

The study of Mäntylä et al. shows that relations between smells exist. They are not only pure theory. Smells reasoning about coupling problems are of special interest. Smell Detection in Context provides means to define relationships of interest for a smell.

Such a “smell relation” defines a connection between a “smell element” and another element of the analysed program. These elements are called “related elements”. The relation between the smell element and the related element has also a direction. Possible directions are out-going, in-going and parent relations. The direction depends on the analysed relations. For method calls, the direction is out-going. In-going directions are used for “called/used by” relations. The parent direction represents analysed relations concerning the direct inheritance of a class.

Some smells do not work on relations, like godclass or middle man smells. These smells do not define any relationships.

Further details concerning a smell are also of interest. Two types of details exist. On the one hand, details for a single smell are of interest. On the other hand, also details concerning a defined relationship are also important.

Details of a smell are called “first order details”. These details are defined for a smell at a concrete element. A first order detail consists of a name, a corresponding location in the sourcecode and a textual representation of that location. Further a detail may also contain a value computed by a metric. For the godclass smell it is, for example, interesting to know which methods have a high complexity. The first order detail for the godclass smell at a concrete smell instance has a detail to list all of these complex methods. For each method, the location of the method in sourcecode, its complexity and the method’s name are returned by the detail.

For relations between elements defined by a smell, “second order details” are defined. Second order details are defined for a smell relation between an smell element and a related element. Similar to first order details, also a name is specified. Further the details location (e.g. a field or method) and a textual representation are returned. For metric computations on the relation a detail value may be returned (or zero otherwise).

Within the approach, relations defined by smells are stated between a method and related types or between the enclosing type of the method and its parent type. This limitation is used to build a very local, context-sensitive presentation of smells. This local smell context provides detailed information about smells at the method and type level during floss refactoring.

The elements and relations form an annotated graph, called “smell context graph”. This graph gives the smells at an element and resulting smell relations to related elements. Any smells at related elements are also part of this graph.

6.5.2 Examples: Feature Envy and Shotgun Surgery

Given the feature envy smell, several relations and details are important. First, it is important which types are envied. This is expressed as smell relation, to visualize this question as an edge in the smell context graph. The relation is defined as an out-going relation. It is also important, which fields are envied. This is defined as second order detail on the relation between the method having feature envy and the envied types. To refactor a feature envy problem, it is especially important if the envied types are dataclasses. The approach does also consider the smells exposed by related elements, thus they are available for a user.

Considering the shotgun surgery smell, other relations and details are of importance. Shotgun surgery exists at methods that are called by a very high number of other methods. The relation of importance are types that contain these method calls, to present an overview of the situation. This smell relation is an in-going relation. The second order detail for this relation uncovers which concrete methods in a type call the shotgun surgery method.

The godclass smell is defined in terms of method complexity and number of foreign data accesses. No smell relations are thus defined. The important details are the complex methods, the overall complexity of the type and the methods that access a lot of foreign data. These details are expressed as first order details.

The table 6.1 gives an overview of the different relations and details for each smell that was considered in the Smell Detection in Context approach. If a value is given for a detail, the short name of the used metric is stated in braces.

6.5.3 Implementation of Smell Relations and Details

The definition of concrete smell relations and details is done as predicates in prolog. They accompany the definitions of smells in Cultivate.

Listing 6.10 shows the template for the predicate that is used to define smell relations. For each Smell, detected at an Element, the list of RelatedElements is enumerated in a defined Direction. The textual representation is *returned*¹¹ in the variable RelatedElementName.

Listing 6.10: Template for a Predicate defining a Smell Relation

```
smell_relation_(Smell, Element, RelatedElement,
               RelatedElementName, Direction)
```

Additionally, also first and second order details are specified for the smells of the Smell Detection in Context prototype. In listing 6.11, the template of the predicate for first order details is shown. Similar to the smell relations, first order details are defined for a Smell at an Element. The predicates return a name for a detail (DetailName), its location in the sourcecode (DetailLoc) and a textual representation (DetailText). They are allowed to return a value, for example computed by a metric (DetailValue).

Listing 6.11: Template for a Predicate defining a First Order Smell Detail

```
smell_detail_(Smell, Element, DetailName,
```

¹¹Logic predicates do not return values, but they unify free variables with values. Whenever *return* is used, it is a short form of unifying a certain free variable with a value.

Smell	Relations	First Order Details (Value)	Second Order Details (Value)
Godclass	-	Complex methods (CYCLO) Method w. foreign access (AFTD)	-
Brainclass	-	Brainmethod (CYCLO) Complex methods (CYCLO)	-
Dataclass	-	Public fields accessors	-
Iceberg Class	-	Total complexity (WMC)	Private complexity (WMC)
Middle Man	Delegated types	Number of delegated methods Delegating methods	Delegating methods
Brainmethod	-	Method complexity (CYCLO) Nesting (NEST) Number of variables (NOAV)	-
Feature Envy	Envied type	-	Envied attribute
Intensive Coupling	Used types	-	Used methods
Dispersed Coupling	Used types	-	Used methods
Shotgun Surgery	Used by types	-	Used by methods
Refused Parent Bequest	Super type	-	Not specialized methods
Tradition Breaker	Super type	% added services (PNAS) added methods	-

Table 6.1: Smell Relations and Details

```
DetailLoc, DetailText, DetailValue)
```

Second order details are an extension of the first order detail definition (listing 6.12). They are defined for a smell at an element, in relation to a `RelatedElement`. Again a name for the defined detail is given. Further variables return the location, a textual representation and possibly a value (zero otherwise).

Listing 6.12: Template for a Predicate defining a Second Order Smell Detail

```
smell_detail_(Smell, Element, RelatedElement,
    DetailName, DetailLoc, DetailText, DetailValue)
```

The feature envy smell works on the “uses” relation between methods and attributes. The accessed to attributes may be direct or indirect through accessor methods. This “uses” relation is expressed as smell relation between the feature envy instance, i.e. a method, and the types containing envied attributes. The concrete envied fields provide a detail on this smell relation.

The described relation and details of the feature envy smell are defined as follows. The feature envy detection strategy uses several metrics, that expose predicates to work on the “uses” relation between a method and foreign data. The smell relation predicate shown in listing 6.13 uses these predicates of the involved metrics to extract all accessed foreign data (attributes). The involved types are inferred and duplicates removed. These types are the related elements of the smell relation defined for feature envy. The names of the related elements are used as textual representation. The used predicates `field_is_in_type` and `name_of_element` belong to the high level API of the Cultivate Platform.

Listing 6.13: Smell Relations of Feature Envy

```
smell_relation_(feature_envy, Element,
    RelatedElement, RelatedElementName, out) :-

    all_accesses_of_method(Element, Attributes),

    findall(Type,
    (
        member(Attribute, Attributes),
        field_is_in_type(Attribute, Type)
    ),
    Types),

    list_to_set(Types, TypesWithoutDuplicates),
```

```
member(RelatedElement, TypesWithoutDuplicates),
name_of_element(RelatedElement, RelatedElementName).
```

The details of the feature envy smell are defined for the relation between the smell instance and envied types. Thus it is a second order detail. This detail returns the envied attributes of the related envied types that are accessed by the feature envy instance (i.e. a method). In listing 6.14, the definition of smell detail predicate for the feature envy smell is shown. The attributes of the envied type (`EnviedType`) that are accessed by the feature envy instance (`Participant`) are collected by using the “uses” relation between methods and attributes. The accessed attributes are the locations of the details. A textual representation is constructed using the name of the attributes and the name of their type.

Listing 6.14: Smell Detail of Feature Envy

```
smell_detail_(feature_envy, Participant, EnviedType,
    'envied', DetailLoc, DetailText, 0) :-

    findall(Attribute,
    (
        uses(method, attribute, Participant, Attribute),
        field_is_in_type(Attribute, EnviedType)
    ),
    Attributes),
    list_to_set(Attributes, Details),

    member(DetailLoc, Details),

    field_type(DetailLoc, FieldType),
    name_of_element(DetailLoc, FieldName),
    string_concat(FieldName, ' : ', Tmp),
    string_concat(Tmp, FieldType, DetailText).
```

The godclass smell does not include any definitions of smell relations. Instead, two first order details are given to provide insight into the design problem. On the one hand, a list of complex methods along with their cyclomatic complexity is defined as detail in listing 6.15. On the other hand, methods accessing an exceeding amount of foreign data are listed by the predicate in listing 6.16.

Complex methods are listed by the predicate shown in listing 6.15. For each method in the given godclass (`Participant`), the cyclomatic complexity of the method is measured. If this complexity exceeds the set limits, it is an detail of interest. The

limit is based on the thresholds used in the godclass detection strategy [LM06] and the average number of methods in a class, provided in the study of Lanza and Marinescu. The cyclomatic complexity is provided as value of the detail, the name of the method is used as textual representation. The predicate `type_contains_method` is part of the Cultivate Platform.

Listing 6.15: Smell Detail of Godclass

```
smell_detail_(godclass, Participant, 'method complexity',
    DetailLoc, DetailText, DetailValue) :-

    type_contains_method(Participant, DetailLoc),
    metric(cyclomatic_complexity, DetailLoc, DetailValue),
    DetailValue > 3,
    name_of_element(DetailLoc, DetailText).
```

Methods with an exceeding access to foreign data are collected by the smell detail presented in listing 6.16. For each method, the number of accesses is measured. If this value exceeds the set limits, this method is also a detail of interest. The limit is again based on the godclass detection strategy [LM06] and the study of Lanza and Marinescu [LM06]. The number of accesses to foreign data is used as value of the detail. The name of the method provides a textual representation of the detail.

Listing 6.16: Smell Detail of Godclass

```
smell_detail_(godclass, Participant, 'foreign accessing m.',
    DetailLoc, DetailText, DetailValue) :-

    type_contains_method(Participant, DetailLoc),
    metric(method_access_to_foreign_data, DetailLoc, DetailValue),
    DetailValue > 2,
    name_of_element(DetailLoc, DetailText).
```

6.5.4 *Implementation of the Smell Context View*

The prototype of Smell Detection in Context adds a new view to the Cultivate Platform. This view is called “Smell Context View”. It provides a local, context-sensitive presentation of the smell context graph. The smell context view links itself to the currently activate Java editor in the Eclipse IDE. The smell context graph is presented for the currently edited method, that is selected in the Java editor. Whenever the user

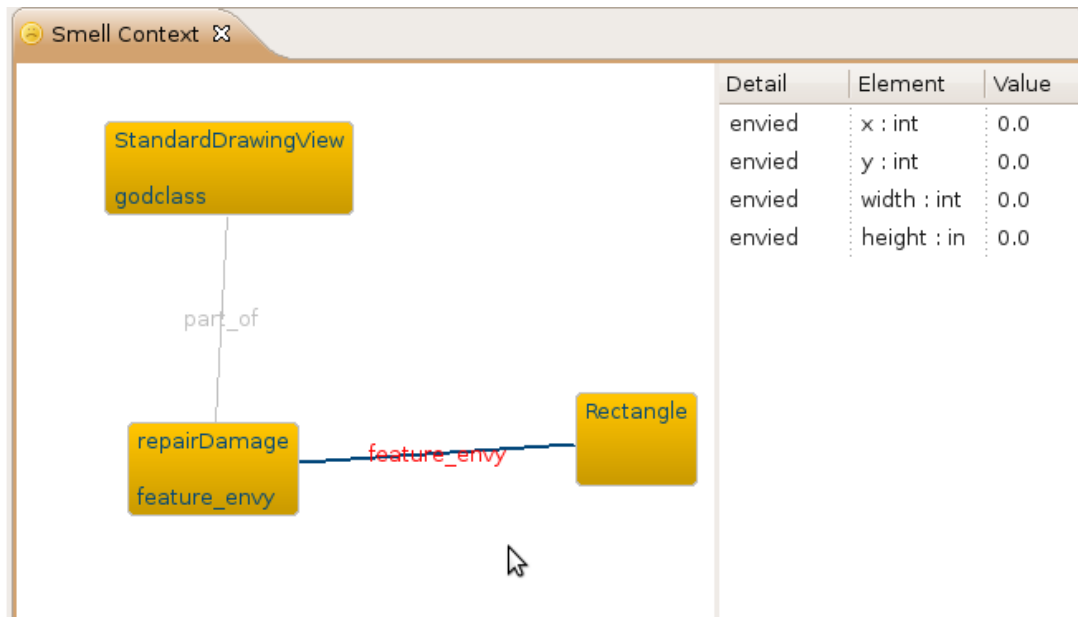


Figure 6.15: Smell Context View showing Feature Envy in JHotdraw

selects another method by navigating with the mouse or keyboard, a new smell context graph is created and presented.

Figure 6.15 shows the smell context graph of the method “repairDamage” in the class “StandardDrawingView”. The class is a central part of the JHotdraw 6 framework. The repairDamage method is a part of the StandardDrawingView - this relation is presented with the “part_of” edge shown in the graph. Further, the StandardDrawingView is a godclass.

The method repairDamage is affected by the feature envy smell. It accesses attributes of the “Rectangle” type, shown by the edge between both nodes. This edge was selected (the line is bold), and the details are presented in the table on the right side. The before described smell details of the feature envy smell are shown. RepairDamage envies four attributes in the Rectangle class: x, y, width and height. All of them have the primitive type “integer”. This way the smell context view presents out-going relations and second order details.

The smell context view can also present first order details, whenever the user selects a node. Figure 6.16 shows the smell context graph for the method “insertFigures”. This method is also part of the StandardDrawingView class. In the figure the node is selected that represents the StandardDrawingView class. As mentioned before, this class is a godclass. In the table on the right hand, both types of details defined by the godclass are presented. Every method that exceeds the complexity threshold is listed together with the complexity. Further, every method that uses to much foreign data is also listed. These methods are accompanied by the number of foreign data that is used.

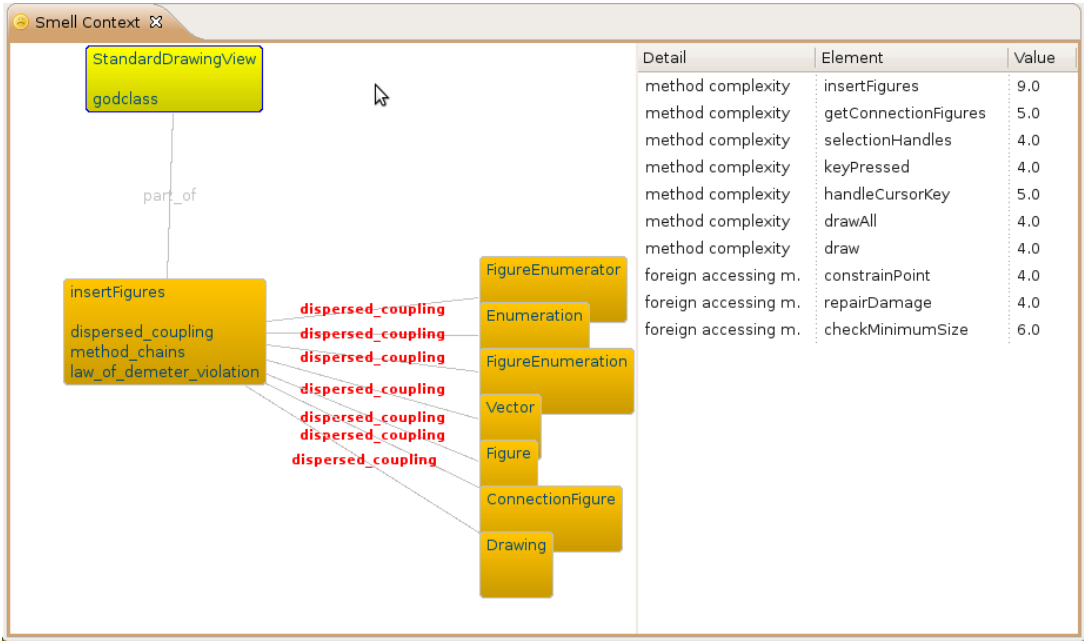


Figure 6.16: Smell Context View showing First Oder Details of a Godclass in JHotdraw

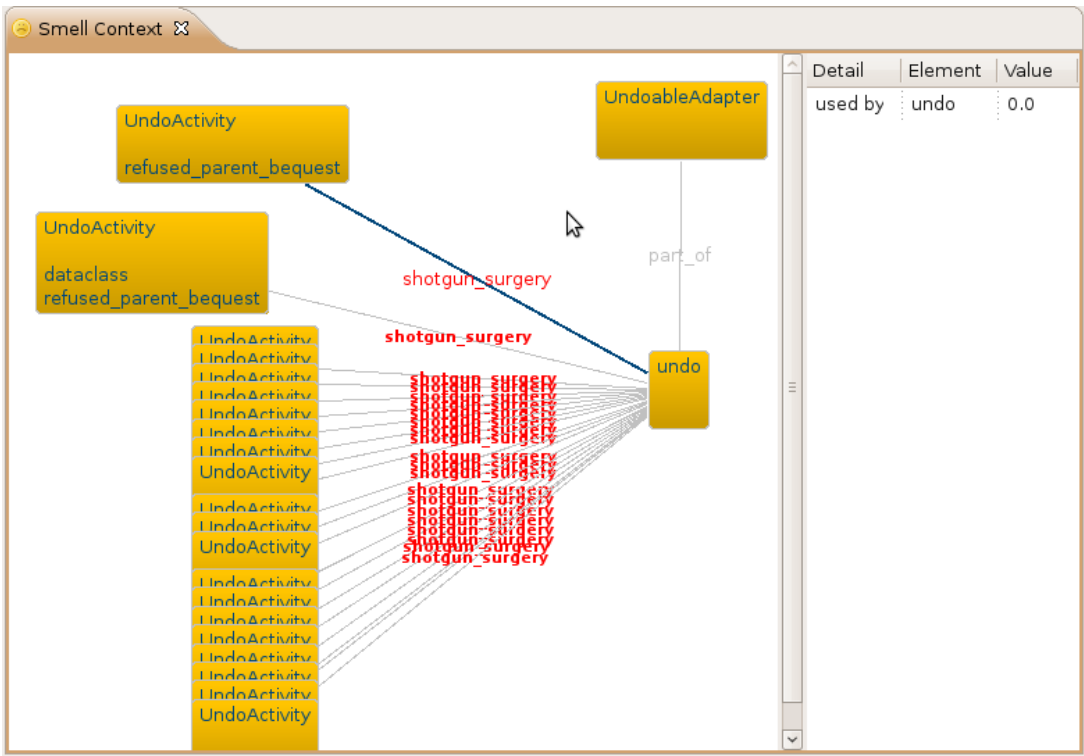


Figure 6.17: Smell Context View showing Shotgun Surgery and Second Order Details

In-going relations and parent relations are also visualized by the smell context view. In figure 6.17, the smell context graph of the method “undo” is presented. The method is part of the “UndoableAdapter” class in the JHotdraw 6 framework. The method is affected by the shotgun surgery smell. On the left a lot of types calling this method are visualized. Some of them have further design problems like refused parent bequest and dataclass. These types have the same names, because they are inner types defined in different features that want to support undo functionality in JHotdraw.

One of the relations defined by shotgun surgery is selected in the figure. The second order detail is listed in the table on the right hand. It is shown, that a method also named “undo” in the class “UndoActivity” calls the visualized instance of a shotgun surgery method.

VALIDATION OF THE APPROACH

7.1 THE CASE STUDY METHOD

Case study is an empirical method to investigate contemporary phenomena within their real-life context [RH09]. It is one of the five classes of research methods that are classified by Easterbrook et al for usage in empirical software engineering research [ESSD08]. Runeson et al conclude that case study is suitable for software engineering research [RH09]. To study the research questions and hypotheses of this diploma thesis in real situations, the case study research method is used [ESSD08] [RH09].

Characteristic properties of the case study method are lack of experimental control [Yin02] and an unclear boundary between the studied phenomena and its context [RH09]. Further, case study research typically has a flexible design, i.e. key parameters may be changed during the course of the study [RH09]. It can also involve quantitative and qualitative data and analysis. To increase the precision, triangulation should be used [Yin02, page 97]. Runeson et al summarize four different forms of triangulation: using multiple data sources, multiple observers, combining different data collection methods and using alternative theories [RH09].

Areas like sociology and political science are common subjects of case study research. Runeson and Höst compare software engineering to such areas and conclude that case study is a reasonable method for software engineering research [RH09]. Given that software engineering is a multidisciplinary area, it involves these common subjects of case studies. As a consequence case study research is suitable for research questions in software engineering.

Case studies incorporate qualities like scale and complexity present only in real-life situations. case studies as such are very suitable for industrial evaluation of software engineering tools [WRH⁺00]. The amount of information exposed during smell detection depends on the context of an industrial project. Size and complexity influence the number of smells, relevance and navigation through artifacts and smells. In general, users have to *synthesize and analyse large collections of data (code)* [MCSW07] during refactorings. As such the case study method suits the exploration of this thesis' propositions.

Carrying out a case study involves multiple steps. First, careful planning of the study, which includes setting up a research design, has to be done. Within the case study protocol, the research questions, data collection techniques, sources of evidence and selection of the cases and subjects are laid out. This states the general procedures to apply during the course of the study. During the subsequent steps, this initial design may however be changed when needed. Evidence is collected from multiple sources of data. This evidence is finally analysed within the selected context. For case study designs involving multiple cases by replication, an analysis that compares the cases, called “cross case analysis”, is performed [Yin02].

The chapter is organized as follows: first the design of the case study is presented. That involves the selection of research questions, cases and subjects as well as procedures of data collection and analysis. The research design finally covers procedures to strengthen validity of this case study. Second, the context of this case study is presented. In the following, a cross case analysis covering both cases is performed and reported. Finally threads to the study’s validity and its limitations are presented.

7.2 CASE STUDY DESIGN

7.2.1 *Research Questions*

In connection with the hypotheses of this thesis, raising questions for three areas looks promising. First of all the usage of context-sensitive smell detection and the usage of classic smell detection, i.e. smell detection that is not aware of contexts, is an area that was not covered before. Second, the usage of context-sensitive smell detection during floss refactoring may reveal the usefulness of specific contexts. Finally, interlinking should help to understand a design problem as a whole and also in its details. To study this third area, it can be observed how developers in an industrial project study a design problem in depth, while they refactor.

From these three areas in connection with the hypotheses, the following research questions for the case study have been derived:

1. How is context-sensitive smell detection used, compared to classic smell detection?
2. How does context-sensitive smell detection help identifying problems during floss refactoring?
3. How are design problems studied as a whole and also its details?

Name	Location	Value	Help text
godclass	de.soptim.prosys.year.gui.model.YearOverviewTableModel	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.salesaccount.gui.model.SalesGroupTrea...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.salesaccount.gui.controller.SalesAccou...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.salesaccount.gui.SalesAccountEvaluati...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.salesaccount.SalesAccountService	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.subproject.SubProject	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.subproject.gui.controller.Over...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.packet.PacketService	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.packet.Packet	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.gui.controller.ProjectEvaluation...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.gui.controller.ProjectAssignme...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.ProjectService	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.project.Project	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.order.orderposition.gui.controller.Orde...	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.order.orderposition.OrderPosition	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.order.filter.FilterOrder	-	indicates a godclass: non-cohesive, overly complex, accessing to many for
godclass	de.soptim.prosys.order.OrderService	-	indicates a godclass: non-cohesive, overly complex, accessing to many for

Figure 7.1: Godclass smell listed in the result view of the classic prototype

4. How is the structural context used to remove false positives?

7.2.2 Case and Subjects Selection

To investigate the research questions of the case study, a representative context has to be chosen. An industrial project provides typical constraints regarding budget, time and quality. Further, the quantities size, complexity and existence of multiple design problems should be typical for an industrial project.

The project “ProSys” is a system providing project and order management, time tracking and project controlling. It is developed and used internally at SOPTIM AG. The project has evolved over several years and exposes typical characteristics. Thus it fits the stated requirements for case selection.

This case study is designed as multiple-cases study to derive theories from the comparison of classic smell detection and smell detection in context. Case and subject replication is also used to strengthen external validity [Yin02, page 36], i.e. whether the study’s findings are generalizable.

During the first case, a prototype of classic smell detection is used. That means that Cultivate is used to detect smells presented in this thesis; however, without any form of task and relevance context, interlinking or any form of structural context. All smells described in chapter 3 are available for detection. Within figure 7.2.2, Cultivate and the Eclipse IDE are shown, while using it to analyse ProSys. The result view in the right-bottom corner displays all detected instances for selected smells.

During the second case, the prototype of this thesis is used containing all three types of context. As shown in table 7.1, interlinking is available with the “Smell Context View”, relevance context is represented by “Link To Editor” and “Focus On Mylyn Task” features. The structural context is build by the “Structural False Positive

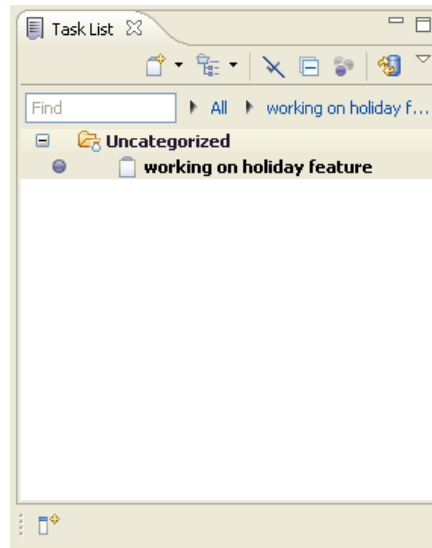


Figure 7.2: Elements in the current Eclipse Mylyn task “Working on holiday feature”

	Case	Case
Feature	ProSys + Classic	ProSys + SDIC
Result View	X	X
Smell Context View		X
Available Smells	All in chapter 3	All in chapter 3
Link Result View with Editor		X
Focus Result View on Mylyn Task		X
Temporal Result View Filtering		X
Structural False Positive Reduction		X
Eclipse Call Graph	X	X
Eclipse Reference Search	X	X

Table 7.1: Features of Prototypes in both Cases

Reduction” feature. Figure 7.2.2 shows the Result View of Cultivate with context sensitive smell detection. The result view presents smell instances only relevant, when the task context is activated. The task context is a combination of “Link To Editor” and “Focus on Mylyn Task” features, as listed in table 7.1. The “Link To Editor” feature allows the Result View to follow the currently opened editor, showing only smells for the class and its methods that are edited in this editor. The “Focus on Mylyn Task” features links the Result View with the Eclipse Mylyn Plugin, that allows to filter views to show only elements relevant to the current task. The Eclipse Mylyn task context corresponding to figure 7.2.2 is shown in figure 7.2.2. The feature “Structural False Positive Reduction” provides the structure context and the interlinking of smells is presented in the “Smell Context” view (see table 7.1).

ProSys is currently developed by two software developers, called Developer A and Developer B in this case study. Both developers are the unit of analysis. Each of

Figure 7.3: Smells relevant to the current task listed in the SDIC prototype

Before the start of the case study, both developers are trained for one hour on the topic of smells. This training includes all smells presented in this thesis, as they are part of the accompanying prototype.

During both cases, data is collected from more than one source to improve the reliability of the study. The first source of evidence is the observation of both developers in each case. An observation guide is used to structure the observations. The second source of evidence are interviews, done at the end of each case.

The developers are highly aware of the fact that they are observed. Apart from questions to support a “think aloud” protocol, there is no interaction between developer and observer. An observation with a “think aloud” protocol is an observation where the

researcher ask questions like “What is your strategy?” and “What are you thinking?” [RH09]. Thus the researcher is only seen as researcher and not as a member of the team, to prevent interference and reduce biasing. The observation guide is available in full detail as appendix.

After using each prototype, the developers are interviewed using a semi-structured interview. This type of interviews allows improvisation and exploration during the course of it. Semi-structured interviews are in line with our research questions as this type allows a mix of open and closed questions to describe and explore how individuals experience the phenomena. Still, this form of interview requires preparation and guidelines to minimize threats to validity [RH09]. An interview guide is available as appendix and part of this research design.

After a period of time, in this case two weeks, the participating developers are interviewed again in a “late interview” to ensure that interviews and observations before are not biased by the subjective novelty of the approach.

7.2.4 *Analysis Procedures*

Basing a case upon theory is a standard measure described by Yin to increase the external validity of a case study [Yin02, page 34 and 111]. Each case is itself based on theoretical propositions drawn in 4.2, that are the foundation to select both cases and the units of analysis. This theory is again used to analyse the collected data. In order to provide internal validity [Yin02, page 36], the propositions of a case study have to be linked to the collected data and evidence. During analysis the hypotheses are triangulated across the units of analysis to provide reliable explanations [RH09]. Pattern matching is a powerful technique to analyse data and strengthen the internal validity. The pattern matching technique requires a set of patterns that are matched against the empirically found patterns [Yin02, page 116]. Further, also rival theories have to be checked against the findings and explanations [Yin02, page 112]. Rival explanations can also be setup in form of patterns and used in pattern matching [Yin02, page 118]. Finally, the findings of both cases are used in a cross-case analysis to setup a comparing case study [RH09].

This case study is based on the propositions stated in 4.2 and 7.2.1. In table 7.2 these propositions are linked to collected data. Having false positive smell indications that are caused by a structure, especially when they annoy the user, indicate that the structural context is useful. Whenever the observed developer uses the structural context to remove these false positives, this strengthens the hypothesis H2 about the usefulness of the structural context.

Data	Proposition
Structures cause false positives	H2 - Structural context removes FPs
False positives annoy the user	H2 - Structural context removes FPs
Structural context removes FPs	H2 - Structural context removes FPs
Amount of smell indications	H3 - Relevance reduces information load
Smells or elements missing	H3 - Relevance reduces information load
Usage of smell context view	H4 - Interlinking helps understanding
Favoring smell context view	H4 - Interlinking helps understanding
Integrating it in refactoring procedurs	H4 - Interlinking helps understanding

Table 7.2: Linking Data To Propositions

An excessive amount of smell indications is a first step towards hypothesis H3, i.e. the relevance context makes the information load of smell detection manageable. Of special importance is this fact, if the amount of smell indications distracts the developer from digging deeper into his current refactoring opportunity. However the relevance context should not remove too many elements, especially no elements that are needed by the developer to study a refactoring opportunity.

The usage of the “Smell Context View” in the prototype indicates that interlinking has a use for the studied developer. Favoring or even fully integrating smell interlinking into the developer’s process to identify refactoring opportunities is a step of special importance towards hypothesis H4.

From these links several patterns for explanations and patterns for rival explanations are derived.

With regards to the area of structures and false positives, two patterns of interest have been identified. The first one (P1) is directly derived from hypothesis H2. Pattern P2 deals with the fact that information overload possibly impacts the developer’s sensitivity for false positives and structures.

P1 Structural Context is used to remove false positives when looking in-depth at a complex design problem, where parts of the problem are related to a structure.

P2 false positives caused by structures are cared less about when not focussing smell detection on the current task.

This information overload is also expressed in pattern P3 on the need of task focus during smell detection as an integral part of floss refactoring. The strategy of using the relevance context in floss refactoring is described in pattern P4.

P3 Without task focus, a developer is affected by information overload during smell detection as part of floss refactoring.

P4 During a floss refactoring, developers use the relevance context to reduce the amount of information.

Related to the focus on a smaller amount of information are the patterns P5 to P7. The usage of smell interlinking to understand all details of a design problem is expressed in the pattern P5. The second pattern for this area, P6, depicts that a developer does not use other sources than interlinking while pattern P7 is a softer formulation with special regards to call graphs and reference search inside the development environment.

P5 Smell Context View is integrated into the developer's refactoring procedure to understand a design problem in-depth.

P6 Smell Context View and Mylyn Task Focus present all, to the developer, relevant elements.

P7 Smell Context View is favoured over Eclipse Call Graph and Eclipse Reference Search.

Several rival explanations are also possible and are formulated as patterns for pattern matching in the rest of this section. First, pattern R1 is expression of the possibility that developers might not care about false positives and the structures related to them during smell detection. Second, caring about false positives might also not an explicit deciding, but subject to the information load a developer is facing in general. Third, the features of this thesis' prototype, especially the interlinking presented in the Smell Context View, might be used, just because they are *new*.

R1 Most false positives caused by structures are not cared about at all by developers.

R2 Information overload (occurring without task focus) prevents most in-depth analysis of problems.

R3 Smell Context View is used because it is a *new* feature.

7.2.5 *Validity Procedures*

The validity of a case study designates the credibility of its results and to what extend these are generalizable. Yin classifies criteria to judge the quality of a case study design [Yin02, page 33]. Runeson et al. also refer to this classification scheme

[RH09]. This scheme consists of the aspects construct validity, internal validity, external validity and reliability.

The construct validity is an aspect that indicates the operational measures taken are reasonable to answer the given research questions [RH09]. According to the checklist of Höst and Runeson [HR07] as well as Yin [Yin02, page 34], construct validity is established by the use of multiple sources of evidence, a clear chain of evidence and having key informants to review the draft report. This study uses multiple sources of evidence and establishes a chain of evidence during analysis.

Internal validity is of importance whenever causal relations are examined and inferences are being made during data analysis [Yin02, page 36] [RH09]. Yin proposes several analysis tactics to strengthen this aspect of validity. This study employs pattern matching and addressing rival explanations (also with pattern matching [RH09]).

The aspect of generalizability beyond a single case is concern of the external validity. For a single case study, analysis is based on theory to support that kind of validity. More cases are selected by replication logic to construct a multiple case study. Further analysis across all cases constitutes a so called “cross case analysis” [Yin02] [RH09].

The final validity test is reliability. Any researcher following the same procedures as described and conducting the same case should conclude the same results. For this reason, any procedures taken have to be documented in the “case study protocol” and stated in the final report. Doing the same case does not mean using replication logic and doing another case study [RH09, page 37].

7.3 CONTEXT: SOPTIM PROSYS

The company “SOPTIM AG” uses and develops its own project management, controlling and time tracking system, called ProSys. It is developed and used for internal purposes only. After several years of development it has grown to a complex enterprise application of typical size. Further, it is affected by problems related to the procedural paradigm and “historically” grown design problems.

ProSys is build on top of the Java SE platform¹ and consists of two components. A server component is responsible to deliver mail notifications to the users of the system. The client is build as a two-tier client/server architecture. Within the client, layering is used to group presentation logic, domain model and persistence.

¹<http://java.sun.com/javase/> accessed on 22.12.2009

The domain model follows the procedural paradigm. It is made of datastructures and procedures, grouped into classes to so called “Services” within the project. These datastructures and services are grouped by domain aspects into modules (technically: Java packages). No further aggregative structures or sub-module facades exist on top. All data is stored inside a RDBMS and is mapped with an object-relational mapper to the datastructures of the domain model. The graphical user interface is constructed with Java Swing ². The presentation logic is intended to be decoupled with the Model View Controller pattern [BMR⁺96, page 125].

ProSys seems to suffer from the current size. The high number of methods (8516) distributed among only 976 types is above the average that was collected by Lanza and Marinescu in [LM06]. This indicates the already stated missing of aggregative structures and facades. The grown size has also created an excessive level of complexity, that can be measured on average per line and compared to the study of Lanza and Marinescu. ProSys has a average complexity per line of 0.27, which is above the high-threshold (0.24) of average Java projects [LM06]. All 8516 methods make 55935 calls to other methods. This results in an average coupling intensity of 6.568, which is twice the high-threshold (3.2) for typical coupling intensity. Combined with the very high number of shotgun surgery, especially on accessor methods, results in rigidity - changes are not easy to be made. Similar subjectively stated problems have been raised during the interviews with both developers before and during the case study.

7.4 CROSS CASE ANALYSIS

During the first case, features were added into ProSys and the classic prototype was used during floss refactorings, to identify refactoring opportunities. During the second case, the prototype of “Smell Detection in Context” was used to find refactoring opportunities while adding features to ProSys. The matched patterns indicate that Smell Detection in Context is useful approach for developers of complex, grown enterprise applications, especially during floss refactoring.

Several structures were observed in the first case that caused false positive smell indications (table A.3: Ignoring FPs). This was also reported by the interviewed developers (table A.4: Structures causing FPs). Reported structures included services, datastructures, controllers and external APIs. The developers simply ignored those. Within the second case, such structures were defined within the prototype to remove their influence on smell detections (table A.5: Removing FPs by Structures). False

²<http://java.sun.com/javase/6/docs/technotes/guides/swing/index.html> accessed on 22.12.2009

positives were then especially visible, because the set of analysed locations was smaller (table A.6: FPs become relevant when looking at details). Comparing both cases, the structural context is clearly used to remove false positive smell indications when looking in-depth at a design problem (see pattern P1).

No evidence is given for pattern P2. false positives seem more important when only few smells are presented; however, during the interviews no indication was given. A rival explanation (pattern R2) may be the exceeding amount of information that had to be taken into account without relevance context (table A.6: FPs become relevant when looking at details). The number of presented smells seem to have an impact, but it is unclear how. Possibly the understanding for smells has changed during both cases and interferes with this influence.

During the first case the list of identified smells was filtered manually by the developer (table A.4: Painful manual smell filtering). During the interviews the developers talked about information overload without relevance filtering (table A.6). This matches with the pattern P3: developers are overloaded with information, when there is no automatic filtering.

The situation changed during the second case, where the relevance filtering was used to reduce the amount of presented smells. Navigation and detail analysis was focused (table A.5: Task Focused Exploration+Details). This matches pattern P4: developers eagerly use such filtering to reduce the amount of presented information. Similar results are concluded by Kersten in [Ker07, page 83].

Smell details were needed in the first case and manually investigated by iterating over all callers and references shown in the Eclipse Call Graph and Eclipse Reference Search (A.4: Smell Details Important, Painful manual smell filtering). Compared to this situation, the developers used the Smell Context View to quickly analyse the details of a design problem and integrated it into their procedures (tables A.5 and A.6). This evidence matches pattern P5, on the integration of the Smell Context View.

Developers also still used the call graph. Thus pattern P6 does not match. Currently, it can not be validated, if the interlinking approach does present all relevant elements to understand a design problem (pattern P6).

Contrary, pattern P7 does match. In the interviews the developers stated that a lot of their manual work done with the Eclipse Call Graph is now easier to achieve in the Smell Context View. The Eclipse Call Graph did not give a good overview of a design problem (table A.6).

No evidence was collected that hints at developers not caring about false positives caused by structures (pattern R1). As such this rival theory can not be proven in both cases.

The approach to detect smells with a tool and further combining this detection with context sensitivity is new for both developers in the study. Consequently there is a possibility that the observed behavior and the statements during the interview are biased because it is “a new feature” that both are eager to use. For this reason, another interview has been done. This “late interview” was held two weeks after studying both cases in the field. The developers state again, that smell detection with contexts is useful to them. There were opportunities to use this tool during further floss refactorings. However, lack of time prevented a further adoption of the prototype (table tab:LateInterviewsAfterCaseStudy). The third rival pattern (R3) cannot be matched, and there is no evidence that the behavior and interviews are biased because of novelty.

Most patterns can be matched against the collected data. The patterns P2 and P6 could not be matched, as well as the rival patterns R1 and R3. There is no evidence that the interlinking approach used in the Smell Context View exhaustively presents all needed information. Further, we can not conclude that the sensitivity for false positives varies depending on the amount of presented smells. Instead, developers in complex enterprise projects do care about false positives. Summing up, the results indicate that Smell Detection in Context is useful developers doing floss refactorings inside a complex, grown enterprise project.

7.5 THREADS TO VALIDITY

Validity is considered in the research design of this case study. Still, there are some possible threads to validity. Of special importance is the short period of time that was considered during both cases. Furthermore, only a single observer and analyst conducted this case study.

All data was collected by the author of this diploma thesis which is also the developer of the accompanying prototype Smell Detection in Context. The collected data was also only analysed by this author, although all collected data and all conclusions were reviewed by the participants of the case study, developers A and B.

The data for both cases was collected during two days on site. One day was spend for each case. Possibly the experience with smells changed during this period of time was not constant enough, although there was a training about smells and refactoring opportunities before the first case. This understanding of smells and possible refactorings might interfere with the sensitivity for false positives. Thus, the pattern P2 was not matched during the analysis.

7.6 LIMITATIONS OF THIS CASE STUDY

Three major directions of future work can be identified based on this case study. First of all this study considered a typical project. Extreme cases might be also of interest and have to be studied. Second, long term effects of the approach Smell Detection in Context should be studied in further experiments and case study. Finally, investigating the effects of using the approach from the first day of a project, so called “greenfield projects”, is promising.

The conducted case study considered an enterprise project of typical size, history and complexity. The stated theories can not be generalized to other types of projects. Extreme cases like very large systems, possibly also of heterogeneous nature, and very small systems should be analysed during future case studies.

As this case study took only one day of development per case into account, it cannot be reasoned about long term effects. Developers might learn to cope with deficiencies of the classic prototype or reach more efficient levels of usage of the SDIC prototype. To exclude such factors, a long term study has to be conducted in the future.

The context of both cases reported here is a complex application that has grown. Some might call it a “legacy application” [Fea04]. The analysed data is thus generalized to theories in the context of such legacy applications. That leaves open the question, whether and how the approach Smell Detection in Context does also help for projects that start from scratch.

This study has set the grounds to research further questions and directions within the area of context-sensitive smell detection. Greenfield projects as well as extreme cases have to be investigated to generalize the stated theories to more types of projects. Long term effects of the approach are currently also missing and have to be understood to reason further about areas and effects of its usage.

CONCLUSIONS

8.1 CONCLUSIONS

The underlying research question of this diploma thesis was “How can context sensitivity improve smell detection for floss refactoring?”. The measures taken in the approach called “Smell Detection in Context” give several answers. Overall, smell detection for floss refactoring is improved by the combination of several contexts.

The structural context of Smell Detection in Context provides a model to express the impact of structures on the detection of smells. Friend relations and friend smells are used to explicitly define this impact. Structures and smells form a complex relationship. The theoretical analysis of the relationship revealed that smells are a necessary trade-off in many structures. Possibly this impact should be documented more directly in current pattern languages. The four cases of chapter 5 show that friend relations and friend smells reveal false positives in cases taken from real world applications. Using friend relations, the structural context can differentiate between false positives and true positives also in difficult cases. The analysis of the case study in chapter 7 studied the industrial application of the concepts. Instead of handling the symptoms of false positives, the root causes were cured. Instead of suppressing each false positive smell instance individually, structures were defined to cope with the problem. This provides better scalability for the handling of false positives. The hypothesis H2 is corroborated by these facts and inferences.

The case study presented in chapter 7 showed that usage of the relevance context reduces the information load during floss refactoring. The amount of smell instances is shrunk to small sets of relevant design problems. The guidelines “Context-Sensitivity”, “Availability” and “Scalability” of Murphy-Hill and Black are successfully implemented in the relevance context of Smell Detection in Context. Further, the guidelines have been proven useful again in the case study of chapter 7. These results partially support hypothesis H3. The usefulness of temporal information during floss refactoring could not be validated in the case study. Kersten’s task contexts are in practice small (around five to seven files [Ker07]). The resulting set of smell instances can possibly be filtered manually for historically irrelevant problems. The use of temporal relevance seems more useful during root canal refactorings.

Furthermore, the case study of chapter 7 showed also that the interlinking context improves the understandability of detected instances. The context implements the guideline “Relationality” of Murphy-Hill and Black. The relations between a smell and other elements and smells is disclosed. Without the interlinking context, significant manual work is needed in the study to understand a smell instance. Reference search and call graphs are used multiple times. The study revealed that the interlinking context improves this situation, but does not replace all manual work. Still, the improvement supports the hypothesis H4.

Overall, context sensitivity made the identification of design problems during floss refactoring easier. The approach Smell Detection in Context attacks the root cause of false positives in smell detection. The amount of presented information is manageable, due to the use of relevance filtering. Less manual work is required to understand a smell, as its context is disclosed by the interlinking context. In total, the three contexts of Smell Detection in Context make smell detection during floss refactoring easier.

8.2 SUMMARY OF CONTRIBUTIONS

This diploma thesis makes several contributions. In summary, contexts are modeled to provide efficient smell detection for floss refactorings.

Most important is the **formal model of friend relations and friend smells**. The model expresses the impact of structures on smell detection and metrics. Using this model in the **structural context** handles false positives that are introduced by structures.

The impact in a wide range of structures has been investigated. All studied structures have smells as negative consequences. The **detailed analysis of smells as consequence** of structures is a further contribution.

The conducted case study shows how context sensitive smell detection is used in floss refactorings of legacy systems. Smell Detection in Context improves the usability of smell detection. The case study **lays the ground for further empirical analysis** of efficient smell detection in floss refactorings. Furthermore, the applied usability guidelines proved useful for smell detection in an industrial project.

Another contribution is the **concept of smell interlinking**. The interlinking context provides the means to define relations and details of smells, that are needed to understand the design problem at hand.

This diploma thesis contributes the application of Kersten’s task context to the area of smell detection. The measurement “degree-of-interest” provides the **relevance of**

a **smell** within a programming task. This concept is further adopted to compute a **temporal relevance**, using the history of a project.

All concepts, the structural, relevance and interlinking context, were implemented as a prototype. This prototype is available as an addon of Cultivate.

8.3 FUTURE RESEARCH

The presented approach and the case study still have several limitations. Context sensitivity can be applied to other, larger scales. The approach might also be useful for other forms of refactoring. The case study is a first step towards empirical study of context sensitive smell detection. Replication and selection of further, extreme, cases is needed.

The limitations of the conducted case study have already been identified in chapter 7. Both cases had a length of one day each. Long term effects could not be studied. Such effects should be analysed during a significant longer case study that replicates the here presented one. Extreme cases regarding the size are also missing. Very small and, more important, very large projects might show different effects during a replicated case study. Finally, the selected cases of the study in chapter 7 considered a project with a significant history. Using Smell Detection in Context for greenfield development might reduce design problems and foster refactorings right from the beginning. Thus, greenfield development has also to be studied in another case study.

The annotation of concrete structures relies on manual management of these by the user. The definition of structures is based on assumptions, that might change and invalidate these annotation. An integration with the area of design pattern detection and, more general, structure detection is needed. Otherwise the annotations based on specific assumptions and the reality of the system might go out of sync. Derived reduction of false positives is in such cases wrong and might reduce acceptance of the users.

Smell Detection in Context applies context sensitivity to the scope of design flaws and smells. These reported flaws and smells work on the levels of classes and methods. However, design problems exist also on larger scales. Design problems are also known for architecture and whole systems, also called AntiPatterns. The idea of context sensitivity might also work for these scopes.

Finally, the approach “Smell Detection in Context” focused on the improvement of floss refactoring. The usage of contexts for root canal refactoring is also of interest. Especially the temporal relevance of the relevance context seems to provide a good starting point.

In summary, this diploma thesis explored the usage of contexts in smell detection and studied in depth the impact of structures on the existence of smells. Further investigation of context sensitive smell detection is needed. More study of its applicability in industrial projects is also needed.

APPENDIX

A.1 METRIC DEFINITIONS

This appendix gives an overview of all referenced metrics. All provided sourcecode of metric definitions is written in prolog. If not stated otherwise, the complete definition is given in [LM06]. For each metric, the precision (integer or floating point) and range is given with the pattern: precision(range from, to).

CYCLO - McCabe's Cyclomatic Complexity [McC76] int(0, infinity) McCabe's cyclomatic complexity computed over the controlflow graph of a method.

```
metric_(cyclomatic_complexity, MethodId, CYCLO) :-
    internal_method(MethodId),
    build_controlflow_graph(MethodId, Edges, Nodes).
CYCLO is Edges - Nodes + 2
```

LOC - Lines of Code int(0, infinity) Counts all lines of code, excluding comments.

NOM - Number of methods int(0, infinity) Counts all methods that are part of a given class.

WMC (CYCLO) - Weighted Method Count (CYCLO) int(0, infinity) Sum of all method complexity of a given class. Complexity is measured with the CYCLO metric.

```
metric_(weighted_method_count, ClassID, WMC) :-
    named_internal_type(ClassID),
    findall(Complexity,
        (
            type_contains_method(ClassID, MethodID),
            metric(cyclomatic_complexity, MethodID, Complexity)
        ),
        Complexities),
    sumall(Complexities, ComplexitySum),
    WMC is ComplexitySum.
```

AMW (CYCLO) - Average Method Weight (CYCLO) float(0.0, infinity) Cyclomatic complexities of all method of a given class averaged over the number of methods in the given class.

```
metric_(average_method_weight, ClassID, AMW) :-
    named_internal_type(ClassID),
    findall(Complexity,
        (
            type_contains_method(ClassID, MethodID),
            metric(cyclomatic_complexity, MethodID, Complexity)
        ),
        Complexities),
    length(Complexities, NumberOfMethods),
    sumall(Complexities, ComplexitySum),
    divide(ComplexitySum, NumberOfMethods, AMW, 0).
```

TCC - Tight Class Cohesion float(0, 1.0) TCC measures on pairs of methods that are part of a given class. A method accesses fields: $f(x) = a, b, c, \dots$. A pair of methods x, y is interesting, if both methods access the same field: $f(x) \cap f(y) \neq \{\}$

$$TCC = \frac{|\text{Pairs of methods accessing the same fields}|}{|\text{All possible pairs of methods}|}$$

ATFD - Access To Foreign Data int(0, infinity) ATFD is the number of distinct foreign data that a given method accesses. Foreign data are fields that are not part of the method's enclosing class. Accesses to such data are direct field accesses (read, write) and use of accessor methods.

FDP - Foreign Data Providers int(0, infinity) FDP is the number of distinct classes, that enclose the data accessed by a given method (see also ATFD).

LAA - Locality of Attribute Access float(0, 1.0) LAA is the ratio of accessed local data to all accessed data.

$$LAA = \frac{|\text{Accessed local data}|}{|\text{Accessed local data}| + |\text{Accessed foreign data}|}$$

NOPA - Number of Public Accessors int(0, infinity) NOPA is the number of public accessor methods of a given class.

NOPAttr - Number of Public Attributes int(0, infinity) NOPA is the number of public fields of a given class.

MaxNesting - Maximum Nesting Level int(0, infinity) MaxNesting is the maximal depth of nested blocks. Nested blocks are introduced by if, for, while, etc. statements.

NFM - Number of Forwarding Methods int(0, infinity) NFM is the number of forwarding methods of a given class. A method is forwarding, if it calls a method with the same signature on a field of the given class. This field may also be hidden behind an accessor method.

MFR - Method Forwarding Ratio float(0.0, 1.0) MFR is the ratio of NFM to all methods of the given class.

```
metric_(method_forwarding_ratio, ClassId, MFR) :-
```

```

internal_type(ClassId) ,
metric(number_of_forwarding_methods, ClassId, NFM) ,
metric(number_of_methods, ClassId, NOM) ,
divide(NFM, NOM, MFR, 0) .

```

CINT - Coupling Intensity int(0, infinity) CINT is the number of distinct methods that are called by the given method.

CDISP - Coupling Dispersion int(0, infinity) CDISP is the number of distinct types, that enclose the methods called by the given method (see CINT).

BUR - Baseclass Usage Ratio float(0.0, 1.0) BUR is the ratio of used protected members to all protected members of the given class' parent in the inheritance hierachy.

BovR - Baseclass Overriding Ratio float(0.0, 1.0) BovR is the ratio of overridden methods to all methods of the given class' parent in the inheritance hierachy.

NAS - Number of Added Services int(0, infinity) NAS is the number of methods of a given class that do not override a method of the class' parent.

PNAS - Percentage of Added Services float(0.0, 1.0) PNAS is the ratio of NAS to all methods of the parent of a given class. Constructors and abstract methods are excluded from the counts.

A.2 OBSERVATION GUIDE

Observations during a single case

1. Number of different refactorings, simple vs complex refactorings
2. Procedure to identify refactoring opportunities
3. Which views are used?
4. Used Task Context
5. Which structures seem to expose smells (and which ones)?
6. How does the developer cope with this "false positives" ?
7. Identified Problems

Cross-Case observations

1. How is the Smell Context View used?
2. How often is the Smell Context View used?

3. How is it integrated into the above described procedure?
4. Does it replace parts of the above described procedure?
5. Did the structural context help to remove the false positives? Did that help the developers in understanding their design problems?

A.3 INTERVIEW GUIDE

Interviews before the Case Study

1. What is your experience with static code analysis tools?
2. What is your experience with smells, smell detection and refactorings?
3. What is your experience with smell detection tools?
4. What is your experience with task relevance, e.g. Eclipse Mylyn Plugin?
5. What is your subjective judgement of the current structure and possible design problems in the project?

1st-Level Questions

1. What did you especially refactor? What was the most complex, difficult refactoring you did?
2. How did you research the design problem and its implications?
3. How did you research the relation between different smells in a complex design problem?
4. How did you research the possible implications of your refactorings?
5. What seems to work best for you?
6. Do you know structures in the project that reveal smells?
7. Do you consider them as false-positives?
8. Does removing them make sense for you? Is it easier to understand the real design problem then?
9. How do you decide which smells are relevant during your development task?

10. Are the measures taken in the prototype useful for you?
11. Were artifacts missing that you needed to understand a design problem?
12. Did your judgement about the overall structure and possible design flaws in the project change?

2nd- and 3rd-Level Questions

1. Was the Smell Context View useful?
2. Was removing the false-positives with the structures smell intuitive and reasonable?
3. Did the relevance context filter enough?
4. Did the relevance context filter too much?

Late Interview

1. Would you use a smell detection tool that is context-sensitive in the future? Why?
2. Did you use the context-sensitive prototype in the meanwhile? How and why?
3. Have you encountered possibilities where the prototype would have been useful? When and Why?

A.4 CONDUCTED DATA

This section presents a condensed version of all conducted data. A more precise version is kept closed to protect the identities of the participating developers.

Experiences	Developer A
Static Code Analysis	Findbugs, Checkstyle (before checkin)
Smell Detection Tools	None
Refactoring Opportunities	Intuitive
Eclipse Mylyn (Task Relevance)	None
Current Project Structure	Some problems obvious, point to start missing

Table A.1: Interviews Before the Case Study with Developer A

Experiences	Developer B
Static Code Analysis	Eclipse Warnings (all), Findbugs, Checkstyle
Smell Detection Tools	None
Refactoring Opportunities	Experience, Intuition
Eclipse Mylyn (Task Relevance)	Basic Knowledge
Current Project Structure	Specific problems known: LoD, method chains, procedural not scaling

Table A.2: Interviews Before the Case Study with Developer B

Group	Observations
Refactorings	Extract, Move Method in godclasses and heavy shotgun surgery
Almost Manual Procedure	Look for relevant locations Look at all smells there in ResultView Look at Code for spec. smell Dev.B: look at call graph
Result+Coupling Views	Result View, Dev.B also call graph, warnings
Multi-File Tasks	Dev.A: 2 classes, Dev.B: 6 classes
Ignoring FPs	Services have feature envy Utilitily classes have shotgun surgery Usage of external API: dispersed+intensive coupling both ignore these FPs when looking into details FPs not detected within Result View

Table A.3: Observations during Case: ProSys + Classic

Group	Developers Quotes
Smell Details Important	“What is envied?”, “What is dispersely coupled?”
Manually identifying relevant relations	“Manually filtering call graph” “Reading the code” “[...]browsing in ResultView”
Automatic detection is good first step	“Faster, easier than manually doing the detection” “Less time needed during a refactoring”
Structures causing FPs	“Services - Datastructures, Controllers, External APIs” “Because of the flood of information, it may make sense [to remove these FPs]”
Painful manual smell filtering	“Continually adapted the set of locations, constantly adding more locations” “Iterating over all callers and references in the hierarchy and lists” “Painful to do it by hand”
Some new insights into project’s state	“Shotgun Surgery at utility classes, but did not expect it for the rest” “Feature Envy, Godclass, Dataclass was expected”

Table A.4: Interviews After Case: ProSys + Classic

Group	Observations
Refactorings	Extract, Move Method in godclasses and heavy shotgun surgery
Task Focused Exploration+Details	Adding locations to Mylyn context Browsing (small set) of smells Checking relevant methods in Smell Context View
Focused Result and Detail Views	Result View (focused), Smell Context View Result View used for navigation, Smell Context View used to study details
Multi-File Tasks	Dev.A: 3, Dev.B: 3
Removing FPs by Structures	External API and String are not considered in dispersed coupling Calls inside the same inheritance hierachy not considered
Smell Context View needs dual display setup	Eclipse on one display, Smell Context View undocked on another Smell Context View used less, on single display setup

Table A.5: Observations during Case: ProSys + SDIC

Group	Developers Quotes
Focused Navigation and Details	<p>“Navigated in that [Mylyn] context [...] (did that yesterday manually)”</p> <p>“Identified methods in ResultView, than looked at Smell Context View”</p>
Details are important	<p>“Special look at feature envy: what is envied (in Smell Context View)”</p> <p>“Smell Context View used for in-depth look”</p>
Result View gives overview	“Result View only for quick overview”
FPs become relevant when looking at details	<p>“feature envy due to copying of data from model to view”</p> <p>“The size of the result set (small due to task context) has a direct impact on that.”</p> <p>“Coupling was especially important. Removed false positives with structure”</p>
Information overload without relevance filtering	“Relevance filtering is useful. The size of all problems was too striking.”
Interlinking and Filtering replaces a lot of manual work	<p>“The combination of Result View and Smell Context View works best for me”</p> <p>“Open Call Hierachy might also work in some way, but it is more manual work”</p> <p>“It [Call Hierachy] does not give a good overview of the problem”</p> <p>“Now this manual work is made easier with the Smell Context View that helps to navigat”</p>
Automation is not at its end	“Giving proposals of refactorings was missing”

Table A.6: Interviews After Case: ProSys + SDIC

Group	Developers Quotes
Floss Refactoring needs Smell Detection in Context	<p>“A smell detection tool is useful”</p> <p>“without context-sensitivity problematic to use during floss refactorings”</p>
Lacking Time for Adoption	“Used the eclipse I am used tool, mainly because of lack of time to dig deeper”
Might have helped, however didn't adopt	“Would have helped in several ocasions, however, to understand that problematic looking structures are really problematic”

Table A.7: Late Interviews after the case study, two weeks after the study

BIBLIOGRAPHY

- [ACE⁺06] Micah Alles, David Crosby, Carl Erickson, Brian Harleton, Michael Marsiglia, Greg Pattison, and Curt Stienstra. Presenter first: Organizing complex gui applications for test-driven development. In *AGILE '06: Proceedings of the conference on AGILE 2006*, pages 276–288, Washington, DC, USA, 2006. IEEE Computer Society.
- [BMMIM98] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc. New York, NY, USA, 1998.
- [BMR⁺96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, 1 edition, August 1996.
- [Bra03] Francisca Muñoz Bravo. A logic meta-programming framework for supporting the refactoring process. Master’s thesis, Vrije Universiteit Brussel, 2003.
- [BW02] Lionel C. Briand and Jürgen Wüst. Empirical studies of quality models in object-oriented systems. In *Advances in Computers*, pages 97–166. Academic Press, 2002.
- [Chi06] Ravindra Chilaka. A comparative study of static software analysis tools with a special focus on software visualization. Master’s thesis, University of Bonn, 2006.
- [Cow01] Nelson Cowan. The magical number 4 in short-term memory: a reconsideration of mental storage capacity. *Behavioral and Brain Sciences*, 24(1):87–185, February 2001.
- [EE05] Martin Fowler Eric Evans. Fluent Interface. <http://martinfowler.com/bliki/FluentInterface.html>, 2005. Accessed on 06.11.2009.
- [EM02] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *in Proceedings of the 9th Working Conference on*

- Reverse Engineering. IEEE Computer*, pages 97–107. Society Press, 2002.
- [ESSD08] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. pages 285–311. 2008.
- [Fea04] M. Feathers. *Working effectively with legacy code*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2004.
- [Fow99] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.
- [Fow07] Martin Fowler. Implementing an Internal DSL. <http://martinfowler.com/dslwip/InternalOverview.html>, 2007. Accessed on 06.11.2009.
- [FS00] Claus Lewerentz Frank Simon, Frank Steinbrückner. Metrics based Refactoring. Technical report, Software Systems Engineering Research Group Technical University Cottbus, Germany, 2000.
- [FTC07] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Ideodorant: Identification and removal of feature envy bad smells. In *ICSM*, pages 519–520. IEEE, 2007.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.
- [HR07] M. Höst and P. Runeson. Checklists for software engineering case study research. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, pages 479–481. IEEE Computer Society, 2007.
- [Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.
- [Ker07] Mik Kersten. *Focusing knowledge work with task context*. PhD thesis, University of British Columbia, 2007.

- [KHR07] Günter Kniesel, Jan Hannemann, and Tobias Rho. A comparison of logic-based infrastructures for concern detection and extraction. In *LATE '07: Proceedings of the 3rd workshop on Linking aspect technology and evolution*, page 6, New York, NY, USA, 2007. ACM.
- [KLB05] Kristoffer Kvam, Rodin Lie, and Daniel Bakkelund. Legacy system exorcism by pareto's principle. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 250–256, New York, NY, USA, 2005. ACM.
- [LHR88] Karl Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. *ACM SIGPLAN Notices*, 23(11):323–334, 1988.
- [Lie96] Karl Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method*. PWS Boston, 1996.
- [LM06] 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, 1 edition, 9 2006.
- [Mar01] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of TOOLS*, pages 173–182. IEEE Computer Society, 2001.
- [Mar02] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnice University of Timisoara, 2002.
- [Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.
- [Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.
- [McC76] Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

- [MCSW07] Erica Mealy, David Carrington, Paul Strooper, and Peta Wyeth. Improving usability of software refactoring tools. In *ASWEC '07: Proceedings of the 2007 Australian Software Engineering Conference*, pages 307–318, Washington, DC, USA, 2007. IEEE Computer Society.
- [Mey00] Bertrand Meyer. *Object-Oriented Software Construction (Book/CD-ROM) (2nd Edition)*. Prentice Hall PTR, March 2000.
- [MGL06] N. Moha, Y.G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *ASE*, volume 6, pages 297–300, 2006.
- [MH09] Emerson Murphy-Hill. *Programmer Friendly Refactoring Tools*. PhD thesis, Portland State University, 2009.
- [MHB08a] Emerson Murphy-Hill and Andrew P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25:38–44, 2008.
- [MHB08b] Emerson Murphy-Hill and Andrew P. Black. Seven habits of a highly effective smell detector. In *RSSE '08: Proceedings of the 2008 international workshop on Recommendation systems for software engineering*, pages 36–40, New York, NY, USA, 2008. ACM.
- [Mil56] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63:81–97, 1956.
- [MKRC05] G.C. Murphy, M. Kersten, M.P. Robillard, and D. Cubranic. The emergent structure of development tasks. In *ECOOOP 2005—object-oriented programming: 19th European conference, Glasgow, UK, July 25-29, 2005: proceedings*, page 33. Springer Verlag, 2005.
- [MM06] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.
- [MVL03] Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 381, Washington, DC, USA, 2003. IEEE Computer Society.

- [RDGM04] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 223, Washington, DC, USA, 2004. IEEE Computer Society.
- [RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Professional, 5 1996.
- [Rie00a] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich, 2000.
- [Rie00b] Dirk Riehle. Method types in java. *Java Report*, 5:22, 2000.
- [SAK07] Daniel Speicher, Malte Appeltauer, and Günter Kniesel. Code Analysis for Refactoring by Source Code Patterns and Logical Queries. In *Proceedings of the 1st Workshop on Refactoring Tools held in conjunction with 21st European Conference on Object-Oriented Programming (ECOOP 2007)*,. Technical Report No 2007-8, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, ISSN 1436-9915, July 2007.
- [Sli05] Stefan Slinger. Code smell detection in eclipse. Master's thesis, Delft University of Technology, 2005.
- [SRK07] Daniel Speicher, Robias Rho, and Günther Kniesel. Jtransformer - eine logikbasierte infrastruktur zur codeanalyse. In *Proc. 9. Ws. Software-Reengineering (WSR 2007)*, pages 21–22, 2007.
- [TCC08] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *CSMR '08: Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering*, pages 329–331, Washington, DC, USA, 2008. IEEE Computer Society.
- [TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. *Software Maintenance and Reengineering, European Conference on*, 0:91, 2003.

- [vDKV⁺00] Arie van Deursen, P. Klint, J.M.W. Visser, Arie Deursen, Paul Klint, and Joost Visser. Domain-specific languages. Technical report, Annotated Bibliography. ACM SIGPLAN Notices., 2000.
- [Wik09] Wikipedia.org. Model View ViewModel. http://en.wikipedia.org/w/index.php?title=Model_View_ViewModel&oldid=323009259, 11 2009.
- [WL08] R. Wettel and M. Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 155–164. ACM, 2008.
- [WRH⁺00] C. Wohlin, P. Runeson, M. Host, C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, 2000.
- [Yin02] Robert K. Yin. *Case Study Research: Design and Methods, Third Edition, Applied Social Research Methods Series, Vol 5*. Sage Publications, Inc, 3rd edition, December 2002.

ERKLÄRUNG

Hiermit erkläre ich, diese Diplomarbeit selbständig durchgeführt zu haben. Alle Quellen und Hilfsmittel, die ich verwendet habe, sind angegeben. Zitate habe ich als solche kenntlich gemacht.

Bonn, März 2010

Sebastian Jancke