HELSINKI UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering

Software Business and Engineering Institute

Mika Mäntylä

Bad Smells in Software – a Taxonomy and an Empirical Study

Supervisor: Professor Casper Lassenius

Instructor: M. Sc. Jari Vanhanen

HELSINKI UNIVERSITY OF TECHNOLOGY | **ABSTRACT OF THE MASTER'S THESIS**

Author:	Mika Mäntylä		
Title of the thesis:	Bad Smells in Software – a Taxonomy and an Empirical Study		
Date:	May 8th, 2003	**Number of Pages:**	75

Department:	**Professorship:**
Department of Computer Science and Engineering	T-76 Software Business and Engineering

Supervisor: Professor Casper Lassenius

Instructor: M.Sc. Jari Vanhanen

In this work, the bad code smells are studied empirically and their relationship to source code metrics is evaluated. This work also presents an initial taxonomy for the bad code smell, which improves their understandability and feasibility

The bad code smells, presented by Martin Fowler and Kent Beck, are dissatisfactory structures in the source code of software that decrease software quality by making it less maintainable. The maintainability of software is important, because it is one of the factors affecting the cost of the future development activities.

The literature study looks at the concept of software maintainability, discusses how software maintainability can be measured, and provides motivation and migration techniques to achieve more maintainable software. Based on the literature study, this work proposes a taxonomy for the bad code smells and evaluates the measurability of each bad code smell with source code metrics.

A survey is used to collect the developers' opinions on the existence of bad code smells in particular software modules. The results of this survey show that the developers' opinions on a particular smell in a particular software module are not very uniform. The survey also provides more support to the theoretical taxonomy by showing that there are many strong correlations within the taxonomy's categories.

This study also compares the results of the smell survey to the source code metrics collected with automatic tools. The results show that developers' evaluations of the bad code smells do not correlate with the actual source code metrics. This means that the smell evaluations from developers are not very reliable and that there is a need for automatic smell measurement.

Keywords:	bad code smell, software maintainability, software metrics, software engineering

TEKNILLINEN KORKEAKOULU	DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Mika Mäntylä		
Työn nimi:	Ohjelmistojen pahat hajut – luokittelu ja empiirinen tutkimus		
Päivämäärä:	8.5.2003	**Sivumäärä:** 75	
Osasto: Tietotekniikan osasto		**Professuuri:** T-76 Ohjelmistoliiketoiminta ja -tuotanto	
Työn valvoja: Professori Casper Lassenius			
Työn ohjaaja: DI Jari Vanhanen			

Tässä työssä tutkitaan pahoja koodihajuja empiirisesti. Lisäksi arvioidaan niiden suhdetta lähdekoodimetriikoihin ja niille luodaan luokittelu, joka parantaa niiden ymmärrettävyyttä ja käytettävyyttä.

Pahat koodihajut, jotka Martin Fowler ja Kent Beck esittelevät, ovat ohjelmiston lähdekoodissa olevia huonoja rakenteita. Ne heikentävät ohjelmiston kokonaislaatua heikentämällä ohjelmiston ylläpidettävyyttä. Ohjelmiston ylläpidettävyys on eräs avaintekijä, joka vaikuttaa ohjelmiston jatkokehityksestä tuleviin kustannuksiin.

Kirjallisuuskatsauksessa tutustutaan käsitteeseen ohjelmiston ylläpidettävyys, esitellään tapoja, joilla ylläpidettävyyttä voidaan mitata ja tarkastellaan miten ohjelmiston ylläpidettävyyttä voidaan parantaa. Kirjallisuuskatsauksen pohjalta esitellään luokittelu pahoille koodihajuille ja tarkastellaan lähdekoodimetriikoiden avulla koodihajujen mitattavuutta.

Kyselytutkimuksella selvitettiin ohjelmistokehittäjien mielipiteitä koodihajujen esiintymisen määrästä valituissa ohjelmistomoduuleissa. Kyselytutkimuksen tulokset osoittavat, että ohjelmistokehittäjien mielipiteet hajujen määrästä eivät ole erityisen yhtenäisiä Kyselytutkimuksen tulokset tukevat myös aiemmin esitettyä teoreettista koodihajujen luokittelua, koska suurin osa vahvoista korrelaatioista on samaan hajuluokkaan kuuluvien hajujen välillä.

Lopuksi tutkimuksessa verrataan kyselytutkimuksella kerättyjä arvioita koodihajujen määrästä automaattisesti kerättyihin lähdekoodimittauksiin. Tehty vertailu osoittaa, että kehittäjien arviot koodihajuista eivät korreloi vastaavien lähdekoodimittausten kanssa. Tämä tarkoittaa, että kehittäjien arviot koodihajuista eivät ole kovin luotettavia ja että automaattinen koodihajujen mittaus vaikuttaa tarpeelliselta

Avainsanat:	pahat koodihajut, ohjelmiston ylläpidettävyys, ohjelmistometriikat, ohjelmistotuotanto

Acknowledgements

Doing this thesis work has truly been interesting and inspiring task and I feel that I have enjoyed almost every moment of it. I would like to thank the following individuals who have positively affected the outcome of this thesis.

First of all I like to thank my supervisor professor Casper Lassenius and instructor Jari Vanhanen for their help, guidance, and instructions during the making of this thesis. I would also like to thank Kristian Rautiainen for helping me in working with the case company. Juhana Hietala was great help in my first steps with SPSS and statistical data analysis. Project manager of SEMS, Jarno Vähäniitty, must also be thanked for handling many of the administrative tasks in the project, thus allowing me to concentrate on this thesis. I would also like to thank the rest of the SEMS research team, Juha Itkonen and Mikko Rusama, for their encouragement and the great working atmosphere we have.

I also want to express my gratitude to the case company, who wishes to stay anonymous, and its employees that have made this type of research possible.

Finally an especial thanks goes to my wife Terhi for the love and support she has given me during the making of this thesis.

Espoo, May 8th ,2003

Mika Mäntylä

Table of Contents

1 Introduction

1.1 Background

The mission of the *Software Business and Engineering Institute* (SoberIT) *is to improve the global competitiveness of the Finnish software industry by providing world-class education and research.* SoberIT is a part of the Department of Computer Science at Helsinki University of Technology (HUT). During the writing of this thesis, the author worked in SoberIT with the research project called *Software Engineering Management System* (SEMS), which aimed *at improving the profitability and growth opportunities of small and medium-sized software product companies by ensuring a fit between SW development and management practices and business models.*

Martin Fowler is the author of many well-known computer science books[1]. In his book Refactoring (Fowler 2000) Fowler presents 22 bad code smells and instructions for removing them, i.e., refactorings. The author of this study has previously applied refactoring to legacy software whose original developer was no longer available. Based on those experiences, the author wrote a seminar report for SoberIT's course Software Engineering Seminar.

The SEMS research project also had several industrial partners. One of the industrial partners, whom we shall call by the name BeachPark[2], had showed interest in refactoring. BeachPark had two software products that had been constantly developed for 4-5 years. During that time, some parts of the products had become quite complex and some people at BeachPark thought that refactoring the source code of the products would help to enhance the maintainability of their products.

1.2 Motivation

1.2.1 Personal Motivation

While working as a programmer in software industry and university, I have personally seen software that was difficult to maintain and develop further. Often the original developer of this kind of software had left the organization, leaving other developers to deal with the situation. I feel that detecting badly structured code will help preventing these problems and benefit the working environment of software developers, and also help organizations by making their software development more productive.

[1] *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, *Refactoring: Improving the Design of Existing Code, Planning Extreme Programming*, *Analysis Patterns: Reusable Object Models* just to name a few

[2] To see a description of the case company refer to Section 6.1.1

1.2.2 Research Motivation

The laws of software evolution were introduced in the nineteen seventies by researchers Belady & Lehman who studied how the IBM's operating system 360-series evolved. One of the key messages of software evolution is that software products need to be continuously modified in order to maintain the competitive edge. Another main point is that continuous modification makes software structure more complex, unless effort is made to reduce this complexity. One way to reduce this complexity is to apply a technique called refactoring, which aims to improve the software structure without changing the software's observable behavior. Good programmers have always modified their code to improve its structure, i.e., they have applied refactoring without knowing it. However, only in recent years has continuous refactoring been recognized as an important part of software development thus making it more popular.

In order to identify candidate spots for refactoring Fowler and Beck (Fowler & Beck 2000) give a list of bad smells that can exist in source code. Bad smells bear close resemblance to development level AntiPatterns (Brown et al. 1998), which also describe different problems the software structure can retain. Little academic work has been done to investigate these issues, which are widely recognized in software industry[3]. So I feel there is a need for this kind of academic research.

1.3 Research Problem

According to Fowler and Beck (Fowler & Beck 2000) a bad code smell[4] is a structure that needs to be removed from the source code by refactoring to improve the maintainability of the software. As examples of such structures I can mention classes that are too large, utilization of a switch statement instead of inheritance, and duplicate code. Since little research has been done on the bad code smell, I cannot elaborate an exact research problem that this thesis tries to answer. However, the goal of this thesis is to:

Study the bad code smells and software maintainability

The research goal is achieved by answering the following research questions:

1. How effectively can the different code smells by Fowler & Beck be measured by tools?
2. How can the smells be made more understandable?
3. Do software developers have a uniform opinion on the "smelliness" of the source code?
    a. How do developers' experience and capability affect the smell evaluations?

[3] The Yahoo!Groups refactoring mailing list, which discusses refactoring and bad code smells, was founded in 14.9.2001, and has today (1.5.2003) 1864 members. Martin Fowler's book Refactoring, which introduces the bad code smells, has a sales rank of 3512 at Amazon and 80 people have reviewed it. To put that information into a right context we can compare it to a true classic among software engineering books: Design Patterns by Gamma et al. has sales rank of 1376 at Amazon and 150 people have reviewed it

[4] See Section 4.4.3 for a detailed description of bad code smells

4. Do the developers' evaluations on code smells correlate with the appropriate source code metrics?

Fowler & Beck think that there is no substitution for informed human intuition when it comes to deciding whether a certain code smell should be refactored. I still feel that first research question is worth pursuing, because automatic smell measurement can help developers by providing them more information. In addition, if smells are ever to be used as a part of automatic smell measurement, their measurability must be addressed. Finally, I also need measures for bad code smells to be able to answer the fourth research question.

The second research question is motivated by the fact that currently Fowler & Beck only provide a single flat list of the smells. This flat list of 22 bad code smells is not ideal for understanding the smells, because it is too long for a human mind to understand. The flat list also fails to recognize the common aspects that some of these smells share and it fails to put the smells into a larger context. I think that a shorter taxonomy based on common concepts from a larger context will make them more understandable.

The motivation for the research question 3 comes from Fowler & Beck's idea that no precise criteria for evaluating code smells can be given. Since human judgment is in a significant role when evaluating smells, it is interesting to see if evaluations are uniform. The developers should have a common view, or otherwise the usability of bad smells as indicators of software maintainability is very questionable. The research question 3a deals with the idea that it is unlikely that developers with different experience and capability would evaluate the smells consistently.

In the last research question, it will be interesting to see, whether the smell evaluations on different modules correlate with the source code metrics for a particular smell. This is essential, because if the human evaluations and source code metrics do not correlate, it undermines the smell usability as a maintainability indicator.

1.4 Research Methods

To answer the first research question, I made a literature study on the source code metrics and the bad code smells. Based on what I had learned from the literature, I tried to propose possible measures for the smells and evaluate how measurable the smells are. The measurability would be based on my personal evaluation of how good a chance the measure has of detecting the smell correctly.

To answer the second research question, I utilized the information gathered from the literature and my personal knowledge as a programmer. Based on this I created a taxonomy that maps the 22 smells to 7 higher-level categories.

To answer the research questions 3, 3a, and 4, I conducted a web-based survey in which the developers of the case company participated. The survey contained 22 questions that asked each respondent to evaluate how much of each smell exists in the modules they had primarily worked with. The case company had 18 software developers and 12 of them participated the survey. To answer the research question 3, I used the developers' opinions on the bad code smells. Then I studied how uniformly the developers had evaluated the particular smells in the same software modules. In this effort, I used the standard deviation of the smell evaluation in a particular module. In research question 3a I tried to find differences in smell evaluations based on experience, knowledge of the software module, and role. I feel that the role and knowledge should give enough indication on the developer's capability. I compared the means and utilized the standard t-test to find the differences in smell evaluations. For the research question 4, I additionally measured the source code of the case company. After that I compared the source code metrics of the particular smells with the smell evaluations I had received. The idea here was to see whether high smell evaluations would also indicate high results from source code metrics.

1.5 Structure and Outline of the Thesis

This chapter has introduced the issues to be discussed in this work. The research goals and questions were described in Section 1.3.

The literature study of this thesis is introduced in the following chapters. Chapter 2 will discuss literature on software maintainability and some important issues around it. Chapter 3 introduces the research that has tried to measure software maintainability with approaches that are close to code level. Chapter 4 discusses the ways to keep source code maintainable and the benefits of maintainable source code. Chapter 4 also introduces the bad code smells as a measure of maintainability. Chapters 2, 3, and 4 motivate the research, present the most significant prior work, and present a larger context for it.

My own contribution comes in Chapters 5 and 6, which also provide answers for all research questions. Chapter 5 will provide a taxonomy and measures for the bad code smells. Chapter 6 contains the results from the bad code smell survey and the comparison with source code metrics. Finally, Chapter 7 contains the conclusions and ideas for further research based on this thesis.

2 Software maintainability

2.1 Introduction

This chapter discusses software maintainability. In order to understand what software maintainability is, we must understand what is meant by software maintenance. Software maintenance and maintainability are studied in Section 2.2. To better understand what is meant by software maintenance we also need to look at software life cycles. The effects of continuous software development and maintenance on software's source code are known as a phenomenon called software evolution. Software evolution, its laws, and software life cycles are studied in Section 2.3.

2.2 Software Maintenance and Maintainability

2.2.1 IEEE Definitions on Software Maintenance

The Institute of Electrical and Electronics Engineers (IEEE) defines software maintenance in their *IEEE Standard for Software Maintenance* (IEEE 1998) as follows:

> *"Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment."*

Basically, this means that any activity that modifies software product after its release is software maintenance. Further down the line the document defines three different types of maintenance tasks:

- *Adaptive maintenance* is defined as: *"Modification of a software product performed after delivery to keep a computer program usable in a changed or changing environment."*
- *Corrective maintenance* is defined as: *"Reactive modification of a software product performed after delivery to correct discovered faults."*
- *Perfective maintenance* is defined as: *"Modification of a software product after delivery to improve performance or maintainability."*

Those three maintenance activities listed above are the most widely known and they were actually first introduced by Swanson (Swanson 1976). The definitions of *adaptive* and *corrective maintenance* are pretty much self-explanatory. However, we should realize that the definition of *Perfective maintenance* covers all kinds of improvements like new features, not only performance issues.

There is also a fourth type of maintenance activity in addition to those three presented above. It is listed for example in the *IEEE Standard for Software Maintenance's* Annex A on 'Maintenance Guidelines, but not in the official part of the standard.

- *Preventive maintenance* is defined as: "*Maintenance performed for the purpose of preventing problems before they occur.*"

What this kind of maintenance might mean to software is still quite unclear even to software maintenance community. This confusion was discussed in a workshop titled *"Do we know what preventive maintenance is?"* (Chapin 2000). Because the term is somewhat ambiguous and not very much in use, I shall make no further references to it.

2.2.2 Other Definitions on Software Maintenance

According to Haikala and Märijärvi (Haikala & Märijärvi 1998), software maintenance is resolving customer's problems, fixing bugs, changing program's behavior while requirements change, and adding new features. The book also notes that software products often do not have a maintenance period in their life cycle. On the other hand, we could argue that for a software product everything after the first public release is maintenance. This is also supported by Glass and Noiseux (Glass & Noiseux 1981) who stated that essentially everything is maintenance.

Pigoski (Pigoski 1996) quotes several definitions and concludes that all maintenance activities occur in the post delivery stage of software. He is supported by Schach (Schach 2002) who says that all changes to a product after it has been accepted by the client are maintenance. Similar definitions can also be found in Sommerville (Sommerville 1996).

2.2.3 Software Maintenance Economics

Schach (Schach 2002) has combined figures from several sources from 1976 to 1981 and has found out that maintenance makes up 67% of the effort spent on software during its life cycle. Other sources from 1990s give different numbers for maintenance, varying from the low of 40-60% (Coleman et al. 1994) up to 95% (Pigoski 1996) of the effort of total software life-cycle costs spent in maintenance. The problem with the numbers from 90s is that they are highly speculative and not based on very accurate data. Sommerville (Sommerville 1996) also reminds us that maintenance costs fluctuate greatly between application domains.

Although maintenance seems to make up a very big part of the total cost of a software product, we must bear in mind that the maintenance costs are mostly influenced by the things that happen prior to the maintenance period (pre-delivery phase). According to several sources, high maintenance costs are the result of a poor coding and lack of design, and maintenance costs can be reduced by putting more effort into the pre-delivery phase (Haikala & Märijärvi 1998;Pigoski 1996;Sommerville 1996). It is true that the effort spent into and the comprehensiveness of the pre-delivery phase will greatly influence the length and the efforts of the maintenance period. However, the target of reducing the maintenance costs may not always be desirable. In many cases the customer wants to start using the software as early as possible, even when the software has faults and lacks features. This way the customer can be sure that development is building the right product and the development can also benefit from early customer feedback.

There are also several software processes like Evolutionary prototyping (Sommerville 1996), Incremental development (Sommerville 1996), Rational Unified Process (Kruchten 2000), and eXtreme Programming (Beck 2000) that encourage the customer to start using the system as early as possible. Those processes are not optimized to keep maintenance costs low, but to make sure that the customer gets the right product and that they can start using and getting business benefits from the system as early as possible. In the software product business it would be pointless to delay the software delivery, since this may result in high maintenance costs, because from the business perspective it can be crucially important to beat competition to the market with the new release. A good example on how software product business operates can be found in Cusumano's work (Cusumano & Yoffie 1999), which illustrates how Microsoft and Netscape used scheduled releases in their operation. Those companies could not have cared less about the high maintenance costs involved with early releases. On the other hand, it must be pointed out that in software product business the software product is in the maintenance phase a major part of its life cycle.

More support on why one should not care about the high life cycle effort percentage spent on maintenance phase comes from the fact that most of the maintenance effort is spent on enhancements. Pigoski (Pigoski 1996) quotes several studies made between 1980 and 1990, which all show that the effort spent on *corrective maintenance* varied from 16% to 22%. Pigoski also conducted his own studies while working at the U.S. Navy software maintenance organization and got similar results. Basically, this means that 80% of the maintenance effort is spent either on *Perfective or Adaptive maintenance*, which constitutes as enhancements. So it really makes no difference whether the software is developed during the maintenance period or not, because you are still doing the same things as you would do in pre-maintenance period, which is developing software and fixing bugs.

In the end I would like to suggest that organizations should not pay too much attention on reducing maintenance costs from the software's total life cycle cost, because maintenance really is the state that most software products are in. As discussed above, software product development also benefits from the maintenance phase in the form of increased user feedback. In addition, in most cases organizations start to get money from the software only after the software is deployed or sold. This does not mean that one should rush through initial development period too fast, since lack of design rational and poor coding done in a hurry might result in serious problems when the software is developed further. The key point here is to deploy the software product as fast as possible without making its further development too expensive.

2.2.4 About The Term Software Maintenance

Interestingly, the term software maintenance that was defined in Sections 2.2.1 and 2.2.2 poorly describes what happens after the software is released or deployed. A much better term for the activities that happen after the initial release would be in most cases *software evolution*, and this idea is also supported by Sommerville (Sommerville 1996).

The term "maintenance" is derived from the verb "maintain". The best definitions in this context for the word maintain according to Merriam-Webster's dictionary `<http://www.m-w.com/>` are as follows:

> *To keep in an existing state (as of repair, efficiency, or validity)*
>
> *Preserve from failure or decline (maintain machinery)*

So the word maintenance clearly does not imply that there would be an activity, which would enhance the state of the object to be maintained. In this sense, it is quite awkward that most of the activities performed in software maintenance are aimed to further develop or improve the software.

2.2.5 Definitions on Software Maintainability

As a rule of thumb we could think of maintainability as an attribute of how easy it is to perform software maintenance.

IEEE has defined software maintainability in (IEEE 1990) as follows:

> *The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or to adapt to a changed environment*

In that definition we can easily find the three maintenance (*corrective, adaptive, perfective)* activities defined by the IEEE, which were introduced in Section 2.2.1. Pigoski (Pigoski 1996) quotes several sources, which have almost similar definitions as the IEEE.

2.3 Software Evolution and Lifecycle

The term *software evolution* does not have an official definition, but some parts of software engineering community have used it as a replacement for software maintenance (Bennett & Rajlich 2000). As previously noted in Section 2.2.4, the term software evolution also describes more accurately the phase currently known as software maintenance.

2.3.1 Laws of Program Evolution

M.M. Lehman and L. A. Belady made foundational work on computer programs life cycle in the 1970s. They discovered a set of laws of program evolution that are still valid today. We will look into the first two laws that are considered to be the most relevant.

The 1st law of program evolution is called *The Law of Continuing Change* and it is defined by Lehman (Lehman 1980) as follows:

> *A program that is used and that, as an implementation of its specification, reflects some other reality undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the program with a recreated version.*

The first sentence, which asserts that programs must evolve or they became less useful, has endured well until our days. Currently, there is also a software process called *eXtreme Programming* (XP) developed by Beck (Beck 2000) that embraces change rather than tries to avoid it. One of the main ideas in XP is that change is inevitable, so there is no point in trying to avoid it. However, the second sentence, which states that program decay results in a program substitution, has received some conflicting opinions. Tamai & Torimitsu (Tamai & Torimitsu 1992) found out that the most important reason to replace a program was to satisfy user requirements. On the other hand, we could say that Tamai's results are not disagreeing with Lehman's, because the inability to satisfy user's requirements could be one indication of software decay. It has to be pointed out that Lehman & Belady were studying the IBM's operating system 360-series, while Tamai's research was targeted on Japanese business applications.

The 2nd law of program evolution is called *The Law of Increasing Complexity* and it is defined by Lehman (Lehman 1980) as follows:

> *As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.*

This law has also stood the test of time well and it is still applicable today. Some techniques can be applied to contradict this law and we will study them in more detail in Chapter 4. However, based on my personal experiences, I think that programming done without adequate design or vision also results in very complex and poorly maintainable software. This way we could speculate that program deterioration has different velocities based on the processes used and people involved.

2.3.2 Software Lifecycle Models

Traditionally, the software product lifecycle has been described with the waterfall model introduced by Royce (Royce 1970). An illustration of the model can be seen in Figure 1.

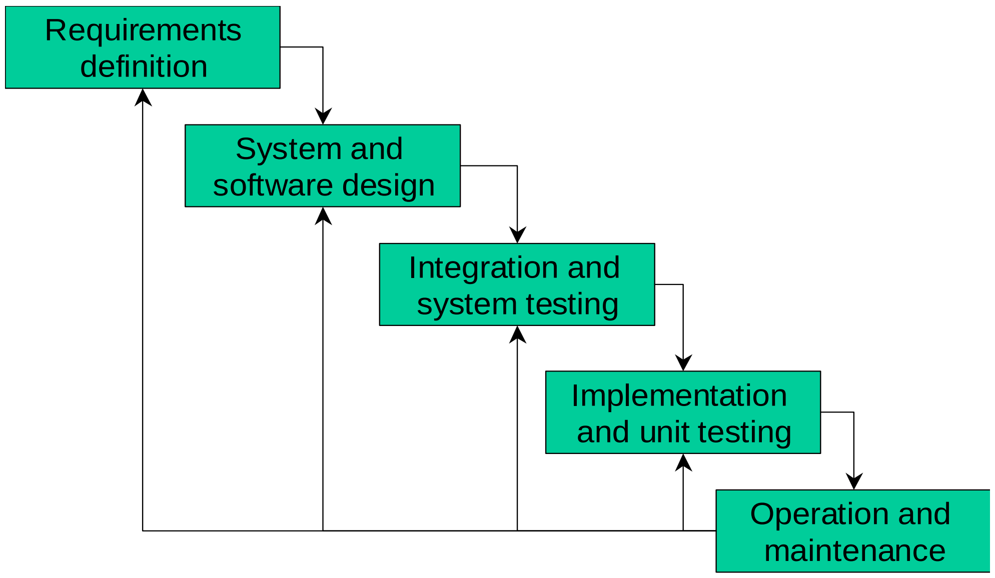
Requirements definition
System and software design
Integration and system testing
Implementation and unit testing
Operation and maintenance

Figure 1 The software lifecycle described by the waterfall model

One of the problems with the waterfall model is that it basically covers only the initial development phase that ends to operations and maintenance phase. In Section 2.2.3 we saw data that showed how maintenance phase takes most of the effort in software lifecycle. Keeping that in mind, it seems irrational to describe the maintenance phase as just one box. On the other hand, the waterfall model poorly describes a case where an enterprise builds a software product and issues incremental releases of the product in certain intervals. Going even further, I could claim that the waterfall model is not even a software lifecycle model, but more like a software development model. Here the waterfall model just acts as an example, which demonstrates the problems that many other lifecycle models also have when describing the software maintenance phase.

In my opinion, a more suitable model for viewing the software maintenance phase and the software evolution that happens is the staged model introduced by Rajlich and Bennett (Rajlich & Bennett 2000). Two figures describing the different versions of staged models can be seen in Figure 2 and Figure 3. Figure 2 represents the whole life cycle of a single product, while Figure 3 can be thought of as an elaboration of shrink-wrap software's evolution stage.

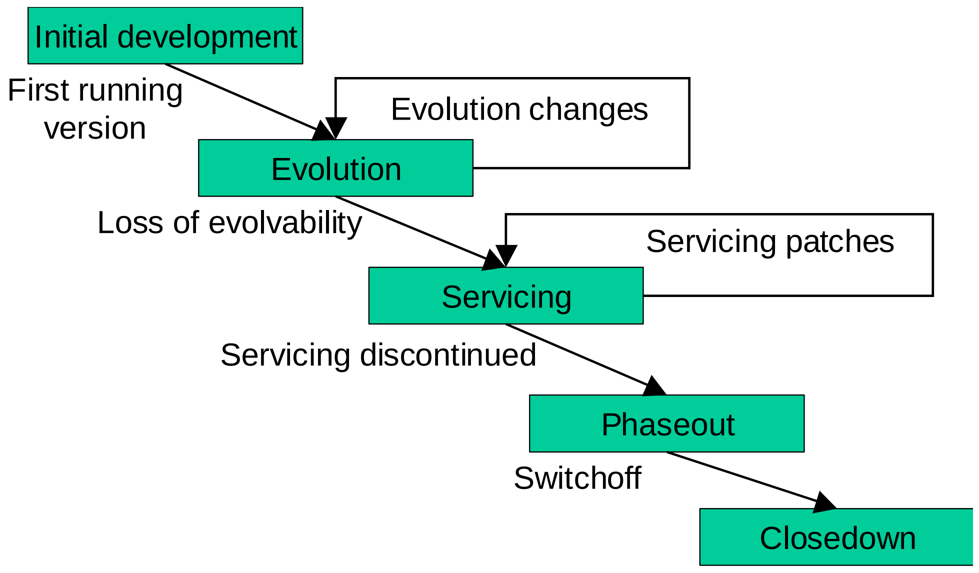

Figure 2 The simple staged model (Rajlich & Bennett 2000)

The staged model has 5 basic phases that can be seen in Figure 2. During the *Initial development* phase the software is built from scratch and the phase ends with the first operational version of the software. If the first version is released to the markets and delivered to the customer, this is where the maintenance phase, according to the maintenance definition introduced in Section 2.2.1, would begin. In the *Evolution* phase, the software is further developed in iterative manner, and the customer demands, competition pressures, and learning guide the development direction. According to Rajlich and Bennett (Rajlich & Bennett 2000), in most cases the software is released at some point during the evolution period, possibly after some alpha and beta releases. After the evolution period comes the *Servicing* phase. During this stage, the software product can no longer be effectively developed, because of the degraded architecture and/or lack of skilled developers from the original development team. In this phase, changes are minimized, because big changes can no longer be made, or they would be too difficult and expensive to make. This phase consists mostly of bug fixes and minor improvements. In the *Phaseout* period, the company no longer makes any changes to the software, but it still tries to generate revenue from it as long as possible. In the *Closedown* phase, the company no longer offers any version of the software to markets and guides interested users to other products. Another interesting thing in the staged model is that going backwards is most likely impossible and that transition between phases is not only affected by the program decay, but also by the availability and the skills of original developers.

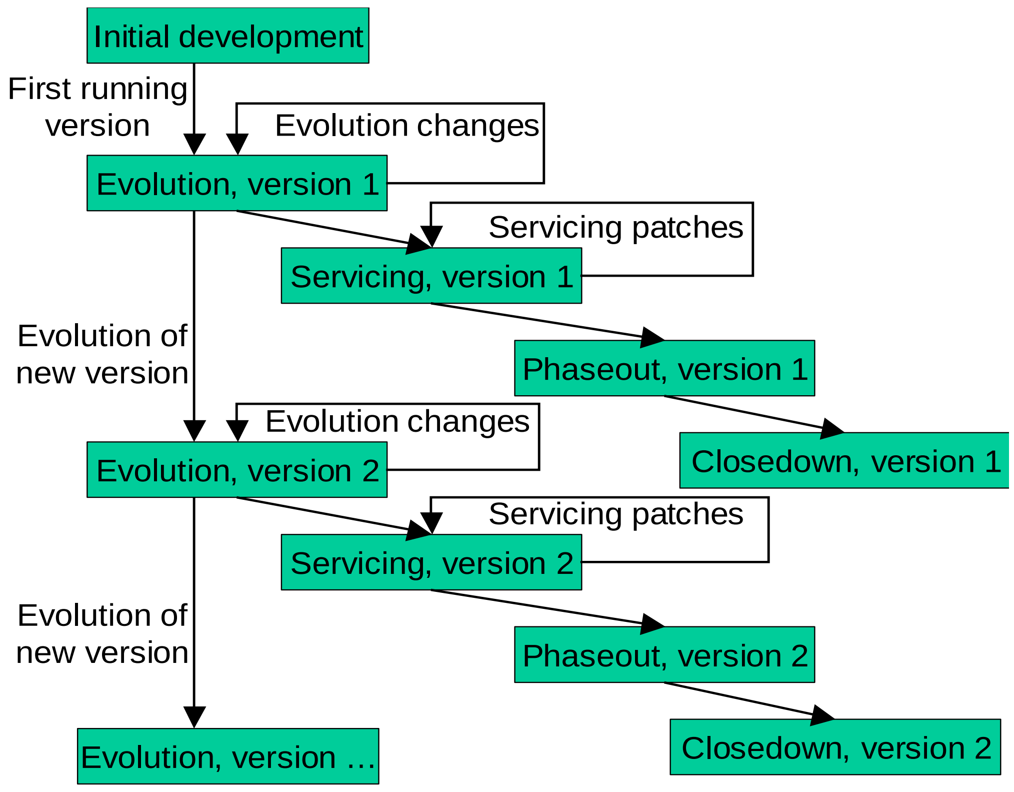

Figure 3 The versioned staged model (Rajlich & Bennett 2000)

The so called "shrink-wrap" software companies often follow the versioned staged model, which can be seen in Figure 3 (Bennett & Rajlich 2000). In this model, each evolution version could be understood as an incremental product version release to markets. In the versioned staged model it is more likely that moving a certain version to Servicing, Phaseout, or Closedown stage is based on a precise decision. In the regular staged model this change is more likely to happen for other reasons such as software decay.

2.4 Summary

In this chapter, the term software maintainability, one of the key elements in this work, was discussed. In order to understand what software maintainability is, the term software maintenance was defined and discussed. It appeared that software maintenance describes quite poorly for the things happening in software maintenance phase. Therefore I can conclude that software maintenance should be replaced with the term software evolution. However, later in this work I will use the term software maintainability instead of software evolvability, because software maintainability has much wider usage and it is also more generally accepted. In this chapter, the software life cycle was also studied and it appeared that the traditional software lifecycle models do not properly describe software evolution/maintenance. The laws of software evolution were also discussed and it became evident that continuous work is needed in order to keep software maintainable

3 Measuring Maintainability

3.1 Introduction

In the previous chapter the concept of software maintainability was addressed. This chapter introduces the most relevant research that has been done in order to measure the vague concept of software maintainability. Software maintainability, like any other software quality attribute, needs to be measured, if we wish to understand it or make comparisons based on it. In Tom DeMarco's words: *You cannot control what you cannot measure* (DeMarco 1982).

The focus will be on those maintainability measurement studies that have used an approach close to code level, because bad code smells are a code level approach as well. Unfortunately, to my knowledge there are no maintainability studies, where the bad code smells or development level antipatterns (Brown, Malveau, McCormick, & Mowbray 1998), which bear close resemblance to bad code smells, would have been studied. Therefore, most of the maintainability studies will focus on source code metrics.

First, in Section 3.2 I will give a short introduction to different source code metrics. Source code metrics are introduced for two reasons: 1) nearly all of the relevant research on software maintainability has utilized the source code metrics 2) I also plan to compare the bad code smells with source code metrics. Many of these metrics are also originally claimed to be a measures of software maintainability as well.

After introducing the source code metrics, I will look at the research on software maintainability in two sections. In Section 3.3 I will discuss the maintainability measurement in connection with procedural languages, and Section 3.4 will focus on object-oriented maintainability. Of course, various people who have created the different metrics for software engineering community have claimed that their metric is a good indicator of software maintainability. The problem with this approach has been that usually only a single metric has been used to evaluate the maintainability, and the evaluators have been the very same people who introduced the metric in the first place. More recently researchers have focused their effort on evaluating software maintainability with respect to a spectrum of metrics. This approach is more solid, since it gives different measures to maintainability, and it also allows dropping out metrics that do not correlate with maintainability. Sections 3.3 and 3.4 will show that code metrics can be used to predict software maintainability that is measured with different aspects. This is important, since I believe that bad code smells are a measure of software maintainability, and I also plan to measure them automatically.

3.2 Source Code Metrics

Measuring software size also presents great difficulties because of the different aspects (and their different interpretations) involved: effort, functionality, complexity, redundancy, and reuse (Fenton & Pfleeger 1996). Two different programs that have essentially the same features, but different authors can illustrate such a problem. If the authors have a big difference in programming experience, it is likely that the program created by the more experienced author is smaller in terms of effort, complexity, and redundancy, but greater in terms of reuse and functionality.

This section will look at some of the most recognized source code metrics. This section will focus only on metrics that can be calculated directly from the source code. Therefore, function points and other similar measures are not discussed here. Easy-to-calculate source code metrics are studied, because they are the most utilized and easiest to apply later on in the sections that deal with smell measurement.

In Section 3.2.1 I will first introduce the traditional and the oldest source code metrics, and then I will look at the most important source code metrics for object-oriented programming in Section 3.2.2. Object-oriented programming is studied separately, because the concept of bad code smells is only introduced in the object-oriented context. Object-oriented programming is the paradigm currently dominant in software development.

3.2.1 Traditional Source Code Metrics

Line of Code (LOC) is the most commonly used software *size metric*. This is likely due to the fact that the LOC is also the most available size metric, since almost all editor programs count the lines of the file being edited. Fenton & Pfleeger (Fenton & Pfleeger 1996) point out that some code lines are different from others and that we must define what is meant by a line of code. Different cases in lines of code are for example blank lines, comment lines, lines with more than one instruction, program headers, and so on. Fenton & Pfleeger (Fenton & Pfleeger 1996) refer to the work by Grady & Caswell at Hewlett-Packard (Grady & Caswell 1987) for the most widely accepted definition of a line of code. According to this definition a line of code is any statement in the program except comments and blank lines. This often abbreviated as NCLOC or just NLOC non-commented lines of code.

Halstead metrics, introduced by Maurice Halstead in 1977 (Halstead 1977), are one of the first metrics for trying to capture software size by other means than just counting lines of code (Fenton & Pfleeger 1996). Halstead's idea was to create a software measure that captures disciplines in physical and psychological measurements. The building blocks of Halstead metrics are:

$\mu_1 = \text{the number of unique operators}$

$\mu_2 = \text{the number of unique operands}$

$N_1 = \text{total occurences of operators}$

$N_2 = \text{total occurences of operands}$

This means that the following statement has two operands (*y* and *x*) and two operators (= and *sin).*

```
y = sin(x);
```

From those building blocks Halstead derived a wide range of different metrics. Those derived metrics will not be discussed here in more detail, because they do not offer any new elements in capturing the different aspects of the source code. Those derived metrics are also quite confusing and lack theoretical or empirical basis. Consequently, Halstead metrics have received a great amount of criticism (Card & Glass 1990;Fenton & Pfleeger 1996;Hamer & Frewin 1982;Weyuker 1988). Fenton & Pfleeger (Fenton & Pfleeger 1996) characterize Halstead metrics as follows:

> *Halstead's software science measures provide an example of confused and inadequate measurement.*

Although Halstead metrics have been questioned widely, they are still used in some cases, like in Section 3.3.3, which introduces some of the most recognized work in the area of maintainability measurement. Personally, I think that the derived Halstead metrics are something that we should not pay any attention to. Still the basic building blocks of Halstead metrics seem reasonable and can provide meaningful information on the source code.

McCabe's Cyclomatic Complexity measures the number of independent execution paths in a computer program and it was introduced by Thomas McCabe (McCabe 1976). The cyclomatic complexity is calculated as follows:

$V(G) = e - n + 2$

where

$V(G) = \text{cyclomatic complexity of graph } G$

$e = \text{number of edges}$

$n = \text{number of nodes}$

As an example consider the following piece of code,

```
int max (int a, int b){
      int max = a;
      if (a < b) {
            max = b;
      }
      return max;
}
```

whose flowgraph can be seen in Figure 4. This program has 4 edges and 4 nodes, thus the cyclomatic complexity is 2.

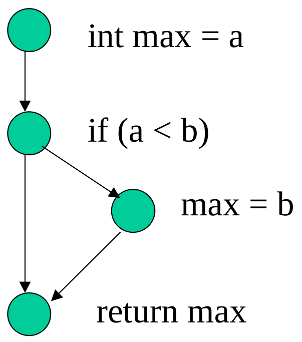

Figure 4 Flow graph of an example program

McCabe (McCabe 1976) originally suggested that small cyclomatic number per program module increases the testability and understandability of the module. In a study at Hewlett-Packard (Grady 1994) 830,000 lines of Fortran code were analyzed and it was found out that there was a strong relationship between the number of updates in the module and the cyclomatic number. Based on the data, the researchers determined that no module should have cyclomatic complexity higher than 15.

Some studies have tried to correlate the cyclomatic complexity with fault metrics. This can be considered sidetracking, since cyclomatic complexity was not originally intended to be a fault predictor. Nevertheless, Schach (Schach 2002) refers to a study by Walsh (Walsh 1979) that showed a correlation between cyclomatic complexity and fault numbers per module. A study by Fenton & Ohlsson (Fenton & Ohlsson 2000) showed that cyclomatic complexity as such does not correlate with fault metrics. However, when cyclomatic complexity was combined with another metric known as *Sig*FF (for details see (Ohlsson & Alberg 1996)), some correlation between the combined metric and fault metric was found. Card & Glass (Card & Glass 1990) also present some data that provides a weak correlation between the cyclomatic complexity and fault metric. Overall, it seems that the cyclomatic complexity cannot directly be used as a reliable fault number indicator.

3.2.2 Object-Oriented Metrics

The main idea of object-oriented programming in contrast to other programming techniques is to put the data and the logic that manipulates it inside a single unit known as a class. In procedural programming, good programmers have structured their programs to different modules that have their own data and logic. Even the idea of a constructor has been represented in procedural programming with each module having its own initialization function that must be called before using the module. So it seems that good programming style and design in procedural languages have always contained the main ideas behind object-oriented programming. This means that some metrics can be directly applied from procedural programming modules to object-oriented language classes.

The most cited work in the area of object-oriented metrics is written by Chidamber and Kemerer (Chidamber & Kemerer 1991;Chidamber & Kemerer 1994), who proposed the following metric suite for object-oriented design in the early nineties: *Weighted Methods Per Class, Depth of Inheritance Tree, Number of Children, Coupling Between Object Classes, Response for a Class,* and *Lack of Cohesion Methods.*

The ideas of cohesion and coupling were presented nearly two decades ago by IBM researchers Stevens, Myers, and Constantine (Stevens, Myers, & Constantine 1974). However, their suggestions are still very valid, because they can be applied to object-oriented programming as well. I believe that they are the two most important aspects of object-oriented programming and I will study them next.

Coupling in object-oriented programming regularly means the dependence between objects and/or classes. Chidamber and Kemerer (Chidamber & Kemerer 1994) presented a coupling metric called *coupling between object classes* (*CBO*) and they define it as follows:

CBO for a class is a count of the number of other classes to which it is coupled

They further define that coupling occurs when methods of one class use the methods or instance variables[5] of another class (Chidamber & Kemerer 1994). So the CBO tells the number of classes that the measured class interacts with. However, it does not take into account the tightness of the coupling, i.e., how much the measured class uses the methods or instance variables of the other class.

One way to look at the tightness of the coupling is shown in the work of Ma, Chang, Cleland-Huang (Ma, Chang, & Cleland-Huang 2001). Their work discusses coupling tightness as what actually happens in the object-to-object connection. In C++ there is a mechanism called *friendship* that can exist between classes. Briand, Devanbu and Melo (Briand, Devanbu, & Melo 1997) note that the friendship mechanism increases coupling, because it allows the friend class to access the body of the other class. Java's package mechanism also has similar an effect as C++ friendship, so it also affects coupling. There has also been some discussion on whether inheritance should be considered coupling. Even Chidamber and Kemerer (Chidamber & Kemerer 1991) earlier included the coupling created by inheritance, but later excluded it (Chidamber & Kemerer 1994). Here we saw just a few different coupling schemas. To get a better overview on different coupling types, refer to an excellent article by (Briand, Daly, & Wüst 1999).

To get some idea on the usefulness of coupling we shall look at Chidamber's and Kemerer's (Chidamber & Kemerer 1994) claims on their CBO metric:

- Excessive coupling prevents class reuse.
- To improve modularity and encapsulation coupling should be minimized.
- High coupling rate hinders the maintenance work.
- Higher coupling indicates need for more complex and rigorous testing.

Several studies (Basili, Briand, & Melo 1996;Briand et al. 1999;El Emam, Melo, & Machado 2001;Yu, Systä, & Müller 2002) indicate that coupling indeed correlates with fault metrics. So there appears to be empirical evidence to the last of Chidamber and Kemmerer's claims. However, it must be noted that the coupling studies presented above found that not all couplings are equally significant as fault predictors.

Cohesion like coupling was already briefly discussed. There we saw how cohesion dates back deep to the era of procedural languages. The best type of cohesion according to McConnell (McConnell 1993) is *functional cohesion*. It means that function performs one and only one task. A prime example of this would be a function called *sin()*. In object-oriented programming, functional cohesion can easily be applied at method level. Measuring class level cohesion is slightly more difficult. According to Grigg (Grigg 2002) an indication of a class being too big can be found in the number of *and*-words in the answer to the question *"what does this class do?"*. In my opinion, this is a good rule of thumb for class level cohesion. Unfortunately, this does not help us if we need an unambiguous measure for class level cohesion. Chidamber and Kemerer (Chidamber & Kemerer 1994) propose a class level

[5] It is interesting that Chidamber and Kemerer (Chidamber & Kemerer 1994) restrict the coupling through variables only to instance variables. One could argue that the use of other classes' static variables should also be considered. However, then the metric would have to be extended to mean more than coupling between object classes.

cohesion measure called *Lack of Cohesion in Methods* (LCOM). We can see the definition of the metric (Chidamber & Kemerer 1994) below:

Consider a Class C_1 with n methods $M_1, M_2, \cdots, M_n$. Let $\{I_j\} =$ set of instance variables used by method M_j. There are n such sets $\{I_1\}, \cdots, \{I_n\}$. Let $P = \{(I_i, I_j) \mid I_i \cap I_j = \varnothing\}$ and $Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varnothing\}$. If all n sets $\{I_1\}, \cdots \{I_n\}$ are $\varnothing$ then let $P = \varnothing$.

LCOM $= |P| - |Q|$, if $|P| > |Q|$

LCOM = 0 otherwise

So the LCOM is the number of method pairs whose similarity (measured with used instance variables) is zero minus the number of method pairs whose similarity is not zero. The equation shows only the lack of cohesion, not the strength of it, i.e., the classes which all have an LCOM value zero can still vary in cohesiveness.

Chidamber and Kemerer (Chidamber & Kemerer 1994) characterize their LCOM measure as follows:

- High cohesion of methods in a class promotes encapsulation.
- Low cohesion implies complexity and therefore the likelihood of errors. Also low cohesion classes should likely to be split up into two or more classes.
- Measuring differences between methods helps to identify problematic areas in the class design.

Whether a class cohesion is a good fault predictor is still under discussion. Some studies (Basili, Briand, & Melo 1996;Briand, Wüst, Ikonomovski, & Lounis 1999) claim that cohesion provides no indication about the fault-proneness, whereas others (Yu, Systä, & Müller 2002) have found evidence that it does.

Like with coupling, which was discussed earlier, there is a wide range of different measures to capture the different aspects of class cohesion. To get a better overview on different cohesion measures refer to the study of Briand, Daly and Wüst (Briand, Daly, & Wüst 1997). In a more recent work Misic (Misic 2001) also presents a new approach to cohesion. He suggests that cohesion (or coherence as he calls it) could be measured by looking at how external classes use the class to be measured. A recent work of Counsell, Mendes and Swift (Counsell, Mendes, & Swift 2002) also discusses why the definition of cohesion has been so elusive.

This section has introduced only a few most recognized object-oriented metrics. To get more information on other possible object-oriented source code metrics, refer to good books on this topic, such as Henderson-Sellers or Lorenz and Kidd (Henderson-Sellers 1996;Lorenz & Kidd 1994).

3.2.3 Conclusions

This chapter has introduced the most recognized source code metrics. These metrics have been introduced to provide the reader with a general idea of the most applied source code metrics and their claimed and researched benefits. The source code metrics are used in many maintainability measurement studies that we will discuss in the following section. I also plan to propose source code measures for the bad code smells, and that is the main target of this work.

3.3 Maintenance Models for Procedural Languages

This section will discuss the early work of maintainability measurement that has focused on source code metrics. This section will discuss the maintainability of procedural languages in Sections 3.3.1, 3.3.2, 3.3.3, and 3.3.4.

3.3.1 Case: University of Kaiserslautern

A research study conducted in Germany (Rombach 1987) reveals us that source code metrics can be used as maintenance effort predictors. Two programs were used, both of which had been implemented twice using two different languages. So the researchers had a total of four systems, which varied from 1,5 to 15,2 KLOC in size. They had also planned identical maintenance tasks that were implemented in each system. The researchers' results propose with high significance ($p < 0{,}001$) that source code metrics can predict maintainability, comprehensibility, locality, and modifiability. In the study, the maintainability was measured by the impact of software structure to maintenance effort.

3.3.2 Case: Virginia Tech

A study conducted by Kafura and Reddy (Kafura & Reddy 1987) in Virginia tech showed that there is a correlation between software code metrics and the perceived maintainability by the developers. In the study, the researchers used seven different software measures. The system under study was a database management system built by graduate students with Fortran programming language and it had an average size of 16 KLOC. Although the system was small, they followed its evolution through three different versions in three years. The developers felt that metrics were useful, as they helped them to pinpoint the poorly structured components of the system.

3.3.3 Case: Maintainability Index

The most famous study in the field of maintainability metrics is most likely the constructions of the maintainability index conducted in the mid-nineties that is reported at least in (Ash et al. 1994;Coleman, Ash, Lowther, & Oman 1994;Coleman, Lowther, & Oman 1995;Zhuo et al. 1993). In this case, the researchers wanted a quick and easy-to-use maintainability measure that software developers could use. So they created a polynomial measure called maintainability index that was used to track, record, and compare the code maintainability of at least two software systems having roughly a size of 240 KLOC. During the work, 50 different polynomial measures where created, tested, and calibrated against software developers subjective evaluation. The subjective evaluations were collected from 15 different software modules with reduced Afotec's maintainability evaluation pamphlet (AFOTEC 1989). The only drawback in the study is that it fails to report which different source code measures were used in different polynomials that were neglected. Nevertheless, the research revealed that Halstead's effort and volume, which are calculated from the basic elements of Halstead's measures (see Section 3.2.1 for more details), were the best predictors of maintainability. This was somewhat surprising because of the amount of criticism Halstead's measures have received. The final polynomial used in the study to measure maintainability was as follows:

$$\begin{aligned}&\text{Maintainability} = \\ &171 - 5{,}2 \times \ln(aveVol) \\ &- 0{,}23 \times aveV(g') - 16{,}2 \times \ln(aveLOC) \\ &+ (50 \times \sin(\sqrt{2{,}46 \times perCM}))\end{aligned}$$

Here *aveVol, aveV(g'),* and *aveLoc* are the average Halstead volume, extended cyclomatic complexity, and the lines of code per function or procedure. The *perCM* is the percent of comments in functions or procedure. According to the study, the idea of the polynomial metric is that maintainability can be divided into three dimensions, which are:

1. C*ontrol structure,*
2. *Information structure,*
3. *Typography, Naming, Commenting*

This maintainability study seems to be the most reported. It is a shame that there are not more reports on how the maintainability index has stood the test of time, i.e., if it is really helpful and in use over several years.

3.3.4 Case: Canadian Industry

A somewhat similar approach as introduced in Section 3.3.3 was also adopted in a study conducted for Canadian industry. In this study, the researchers claimed to focus on design level metrics rather than on code level metrics (Muthanna et al. 2000). In this study, the scientists started with 18 metrics, which they first narrowed down to six, because most of the metrics had very high correlation among themselves. From the remaining six metrics they chose three metrics, which were found as the best maintainability predictors. The approach in this study is particularly solid in how the metrics for predicting maintainability were chosen. From the three best maintenance predictor metrics, they created the following linear model:

$$SMI = 125 - 3{,}989 \cdot FAN_{avg} - 0{,}954 \cdot DF - 1{,}123 \cdot MC_{avg}$$

In the equation, *SMI* is the software maintainability index for a given module. FAN_{avg} is the average number of external calls coming from this module. *DF* is the number of incoming and outgoing dataflow for the module. MC_{avg} is the average cyclomatic complexity for the module.

They also validated the model in a single industrial software system with the size of 92 KLOC. In this validation, they compared model maintainability prediction with developers' opinions. In most cases, the prediction model was in line with developers' opinions, but there were also cases where it was not. Another restriction is that for the model to work, the software system needs to be broken up to modules whose sizes are from 1 to 2 KLOC.

3.3.5 Conclusions

This section showed that source code metrics correlate with maintainability measured with maintenance effort as well as with perceived maintainability by software developers. We also saw two different polynomial models that predict software maintainability. The software maintainability predictions model would, however, benefit from more empirical studies. Now it is impossible to conclude that they are really applicable in real world cases.

3.4 Object-Oriented Software Maintenance

When object-oriented programming started to become popular in the early 1990s, it was considered an answer to virtually every software engineering problem. This section tries to offer a review on object-oriented maintenance models. Although object-oriented measures have been widely used as fault predictors (see Section 3.2.2 for details), not many object-oriented maintainability studies have been conducted. Section 3.4.1 will introduce the significant maintainability studies that do not use source code level approach. Sections 3.4.2 and 3.4.3 will introduce two cases, where object-oriented source code measures are used to predict maintenance effort.

3.4.1 Maintainability of Object-Oriented Software

There is actually some empirical evidence that object-oriented systems are more maintainable than systems built using procedural languages (Henry, Humphrey, & Lewis 1990). This study was made with university students, but nevertheless it provides some empirical data on the issue. To my knowledge, there are no similar studies that would compare maintainability of object-oriented and procedural programs. Another study made by Briand et al. (Briand et al. 1997) showed that object-oriented paradigm offers no benefits when it comes to maintainability of the design documents. On the other hand, a survey study with nearly 300 respondents (Daly et al. 1995) showed that object-oriented developers clearly view software made with object-oriented paradigm more maintainable[6]. Unfortunately, it is unclear to what extent these views derive from personal experience and how much the "marketing hype" has influenced these opinions.

3.4.2 Case: Software Productivity Solutions Inc.

Li & Henry (Li & Henry 1993a;Li & Henry 1993b) studied the correlation of maintenance effort and object-oriented metrics. They studied two commercial systems that were developed with a proprietary object-oriented language called Classic-Ada™. The object-oriented metrics used were those introduced by Chidamber and Kemerer (Chidamber & Kemerer 1991), but they were slightly modified. Maintenance effort was measured in a number of changed code lines per class. Li & Henry rightly point out that change of code lines is more a measure of size than effort. Still they chose to use it rather than other (real) effort measures, claiming that it is easier to verify the accuracy of the data. They conclude with high significance level that the chosen object-oriented metrics can be used as mainte-

[6] Developers also think that object-oriented is beneficial in terms of ease of analysis and design, programmer productivity, and software reuse.

nance predictors. They also compared the object-oriented metrics and simple size metrics such as lines of code, and number of methods and attributes. Based on the comparison, they presented numbers which indicate that object-oriented metrics are better maintenance effort predictors than simple size metrics (such as lines of code) alone. This is important, because many object-oriented metrics have high correlation with the simple measure of program size.

3.4.3 Case: A Controlled Experiment with University Students

A more recent study that links object-oriented source code metrics and maintainability was conducted by Bandi, Vaishnavi and Turk (Bandi, Vaishnavi, & Turk 2003). Their study setting suggested that maintenance performance is dependent on design complexity, maintenance task, and programmer ability. In this work, the researchers studied only the effect of design complexity. The design complexity was measured with interaction level, interface size, and operation argument complexity (A more accurate description of these metrics can be found in the article). The study showed with high significance that all three measures were useful maintenance effort predictors. The limitations of the study were that 1) the subjects of the experiment were students, 2) the software under maintenance was quite small, consisting of only few classes, 3) and the study consisted of only two different maintenance tasks that lasted between 90-120 minutes.

3.4.4 Conclusions

Section 3.4.1 reviewed the studies comparing the maintainability of object-oriented and procedural programs. These studies did not offer conclusive support to the assumption that object-orientation would bring benefits with software maintainability. Sections 3.4.2 and 3.4.3 studied the correlation between object-oriented software measures and software maintainability. Those studies showed that object-oriented measures can predict maintain-ability and that they are better than simple size metrics alone.

3.5 Summary

This chapter has studied the various approaches to measuring the vague concept of software maintainability. First, some of the most recognized source code metrics were introduced and then the source code maintainability measurement of procedural and object oriented programming was studied. As I already mentioned, to my knowledge there are no maintainability studies where the bad code smells or development level antipatterns (Brown, Malveau, McCormick, & Mowbray 1998), which bear close resemblance to bad code smells, would have been studied. Therefore, my maintainability measurement review focused on studies that have used source metrics as a link to source code.

In this chapter we saw that source code metrics can be used as maintainability indicators. This is important, because it shows that by studying the aspects of source code, the vague concept of software maintainability can be measured. Therefore it should be possible to use bad code smells as maintainability indicators, which makes it important to study them. It would also be beneficial to use bad code smells to construct a polynomial similar to the studies introduced in Sections 3.3.3 and 3.3.4. However, this kind of study is outside of the scope of this work.

4 Keeping The Software Maintainable

4.1 Introduction

This chapter will look into some ways of keeping the source code maintainable, thus contradicting the laws of software evolution discussed in Section 2.3.1. The idea here is to look at ways to keep the software product evolvable. The basic ways of keeping software maintainable will be studied, but the main focus will lie on a technique called refactoring. Refactoring is one way to keep software maintainable. Together with refactoring, I will introduce the bad code smells that tell a programmer, when software needs refactoring. The bad code smells can be understood as a new measure of software maintainability, because removing those bad smells from the code is claimed to make software more maintainable. In this chapter, I will also present my critique against the bad code smells, and these will be basis of my own study. This chapter is organized as follows: Section 4.2 will discuss the problem when software needs to be rewritten from scratch. Section 4.3 will briefly introduce re-engineering, a well-known concept to improve maintainability. Section 4.4 contains an introduction to refactoring and bad code smells.

4.2 Replacing and Rewriting Software

There will eventually be a time when the maintenance of a software product is discontinued. This will happen regardless of how maintainable or evolvable the software is, because computer platforms (hardware, operating systems, databases, runtime environments, etc.) continuously advance and change.

A study conducted by Tamai and Torimitsu (Tamai & Torimitsu 1992) showed that in Japanese industry the average replacement times of business applications (categorized as personnel, accounting, sales support, manufacturing) was 10 years. The software discussed in this study were not software products, but mostly in-house made or tailored software for business needs. There are no similar studies available from companies that make software products. However, we could assume that the need to rewrite the software in product companies is greater, since the evolution of the product is quicker due to continuous development effort and changes in business direction.

In software product business, the decision to rewrite software from scratch almost always proposes high risks. This is due to the fact that while the company is rebuilding the product, there is a good chance that competition might knock the company out of the market. For example, Netscape tried to rewrite its browser product for the version 5.0, but failed, and so Microsoft's Internet Explorer took over on the markets (Spolsky 2000a). Cusumano and Yoffie (Cusumano & Yoffie 1998) reveal that Netscape's code base, in the times of Communicator 4.0, was in such a bad shape that Netscape employees thought that if they had shown their source code in the interview for the programmer prospects, they would not have had any new programmers to work for them. According to Cusumano and Yoffie (Cusumano & Yoffie 1998) Netscape's browser client version 5.0 was to be released on 12/1998. However, 5.0 never made it to the markets and the next major release of Netscape's web browser was versioned as 6.0 and it was released in 11/2000, about two years late of schedule. Even that release was far from the quality that at least I expected and only the version 7.0 (released 8/2002) was in the state that 5.0 should have been almost four years earlier. Spolsky (Spolsky 2000a;Spolsky 2000b) also refers to other cases or projects that went to disaster when doing complete code rewrite, such as Ashton-Tate, Lotus's 123, and Apple's MacOS, Borland's Arago and Quatro Pro. Spolsky (Spolsky 2000b) continues that even Microsoft got into trouble when they tried to rewrite document processor Word from scratch in the project called Pyramid, which was later shut down. However, Microsoft had also continued to work on the old Word code base, so they were still able to issue the next release and stay on the market. Still according to Spolsky (Spolsky 2000b) *the worst strategic mistake that a software company can make is the decision to rewrite the code from scratch*. Spolsky's articles certainly are far from scientific, but on the other hand I have no reason to cast doubt to the cases he presents. After all, we must bear in mind that software companies rarely brag about projects that lead to a failure. There is also a contradicting report to Spolsky's articles. Cusumano & Yoffie (Cusumano & Yoffie 1998) report that Microsoft redesigned their IE web browser in a project leading to version 3.0, but they note that at that time the code base was still quite small and thus manageable, which was not the case with Netscape's 5.0 project.

Thus, it appears that the decision to rewrite the software from scratch proposes a high risk, and that this risk increases in relation to the size of the software to be rewritten. In software business, the decision to rewrite the software has to come from economics. An economical model on software rewriting has been introduced by Chan, Chung and Ho (Chan, Chung, & Ho 1996). Some of the managerial implications that their study brings are that one should avoid rewriting large applications, and impose strict quality control in maintenance in order to prevent source code from degenerating. Thus it seems that maintaining high code quality will be more beneficial as the application size increases.

4.3 Re-Engineering

The maintenance standard of the IEEE (IEEE 1998) defines *re-engineering* as follows:

> *A system-changing activity that results in creating a new system that either retains or does not retain the individuality of the initial system.*

Traditionally, re-engineering consists of two components called *reverse engineering* and *forward engineering*. In reverse engineering, the system's structure and internal functionality is re-discovered and documented, but the system's behavior is not modified in any way. In forward engineering, the reverse-engineered system is then further developed and enhanced. Re-engineering is a recognized area of software maintenance, which has been studied for several years. There are many case studies that describe the different re-engineering projects and the approaches used in them (Adolph 1996;Bianchi et al. 2003;Bray & Hess 1995). Sneed (Sneed 1995) discusses the planning of re-engineering from the financial point of view. To get the latest practical information on how to actually conduct a re-engineering project, see the book that discusses object-oriented re-engineering by Demeyer, Ducasse and Nierstrasz (Demeyer, Ducasse, & Nierstrasz 2003).

4.4 Refactoring

4.4.1 Refactoring - Introduction

Refactoring was previously known as software restructuring. There is no great distinction between these terms. However, restructuring is often used with procedural programming, while refactoring is its supplement in the era of object-oriented programming. Refactoring can be thought as a super-set of restructuring, because it basically contains all the techniques in restructuring, but also adds the object/class specific techniques.

History of refactoring goes back to 1960s and 1970s, when it was based on structured programming guidelines. These techniques focused on replacing goto-statements and refining case-statements. The first restructuring tools appeared in the early 1980s. To get more information on the history of refactoring, see Arnold and Opdyke (Arnold 1989;Opdyke 1992).

Refactoring changes the structure of a program without changing its external behavior. The prime target of refactoring is to *improve the design of existing code* (Fowler 2000). Opdyke (Opdyke 1992) made the first academic contribution to the issue with the term refactoring in his PhD thesis, which concentrated on automatizing the refactorings. Opdyke's motivation for his work came from his experiences in the world of telecommunications, where the maintenance periods are long and code decay is a serious problem. Currently refactoring is vastly popular[7], riding with the wave of the agile methodologies and especially eXtreme Programming (XP) (Beck 2000), which has adopted refactoring as one of its practices. XP is a software methodology that disregards heavy up-front design and embraces change rather than tries to avoid it. Therefore, XP needs refactoring to cope with continuous change and also to compensate the lack of design.

In a way refactoring offers nothing new, because good programmers have always restructured their code when required. Even the author of this thesis has done refactoring before he even became aware of such a technique. A good thing about the recent enthusiasm around refactoring is that it increases developers' awareness of the technique and that it helps discovering better ways of doing refactoring. If we compare refactoring to re-engineering, we see that they can be used together and that refactoring can be one technique in the re-engineering process.

4.4.2 Refactoring – The Business Case

Benefits of refactoring might not be clear particularly to those who have never done programming work with large software systems. Unfortunately, those people sometimes sit in the management and therefore persuading them to allow developers to do refactoring might not be easy. Fowler (Fowler 2000) even suggests that when a manager is schedule-oriented, developers do not tell the manager that they are doing refactoring.

Refactoring also breaks the old engineering rule, which says *if it ain't broken don't fix it.* To certain extent the rule holds, but we must also remember the 2nd law of software evolution

[7] The Yahoo!Groups refactoring mailing list, which was founded in 14.9.2001, has today (1.5.2003) 1864 members.

(Lehman 1980), which was called the law of increasing complexity and was discussed in Section 2.3.1. The high complexity of a software system might hinder the further development of the system. So refactoring is just a way to make sure that further development is possible. It is absolutely true that refactoring increases the risk of introducing new faults to the system. There is a study (Graves et al. 2000) that indicates that most faults are found in the parts of the system that have been most recently changed. However, faults are also introduced during the development of new features. So the counterargument against the "*if it ain't broken don't fix it*" rule is that "*are we so afraid of bugs that we cannot further develop the system?*". The motivation here is that refactoring is one of the enablers of the development of new features, and if the answer to the counterargument is **yes**, then no refactoring should be done.

Fowler (Fowler 2000) also tries to convince people to refactor with the following arguments:

- *Refactoring makes software easy to understand.* This is certainly true, because one of the goals of refactoring is that software would be easy to understand and that source code would be self-documenting. Of course, there can be cases when developers disagree on what kind of code is easier to understand. However, in most cases the ideas about what is known as good programming style and good design should be uniform.
- *Refactoring helps you find bugs.* This can be accepted by common sense if we agree that refactoring also increases the understandability. However, empirical evidence to the argument would be nice.
- *Refactoring helps you program faster.* This argument is partly supported by the laws of software evolution according to which the increasing complexity of software system can hinder the software development.
- *Refactoring improves the design of software.* This argument goes hand in hand with the Fowler's first argument, because in software good design is almost always easy to understand.

Rather than justify the importance of refactoring with just reasoning and arguments that lack empirical data, we can take a look at some cases from the industry. A book called Microsoft Secrets (Cusumano & Selby 1995) divulges that Microsoft also does source code refactoring under the name "*20 % Tax for Rewriting Code*". This means that projects reserve 20% of their development effort on reworking the weak parts of the product. This time is often spent right after the product release before new development on the product is started. Netscape's current browser client is developed in the open source project Mozilla, which also does code cleanup (Eich 2002). They target their code clean up to the beta phase as a follow-up to alpha, which includes the more risky development. A case study on Lucent (Mancl 2001) tells us how they used refactoring in what was a more traditional re-engineering project. In this case, after doing reverse engineering phase, they redesigned the system with design patterns and then used small refactorings to get the system's code to match the new design.

4.4.3 Measuring Refactoring Need

This section has discussed refactoring, which is one of the techniques to keep software maintainable. However, refactoring itself will not bring the full benefits, if we do not understand when refactoring needs to be applied. To make it easier for a software developer to decide whether certain software needs refactoring or not, Fowler & Beck (Fowler & Beck 2000) give a list of bad code smells. Fowler & Beck's idea was that bad code smells are a more concrete indication for the refactoring need than some vague idea of programming aesthetics. Fowler & Beck also acclaim that no set of precise metrics can be given to identify the need of refactoring. Therefore, bad code smells are kind of a compromise between the vague programming aesthetics and precise source code metrics. With bad code smells the reader must bear in mind that some smells represent the opposite ends of the same attribute. For example, the size of a class could be a single attribute, and in one end of the attribute the existing smell is called Large Class and in the other it is referred to as Lazy Class. In addition, for each bad code smell Fowler (Fowler 2000) introduces a set of refactorings (*move methods, inline temp, etc)* with step wise instructions on how each smell can be removed. Therefore, the reader should realize that the refactoring concept also includes detailed instructions on how to actually improve the source code. The purpose of this section is to introduce those 22 bad code smells, which are listed below:

- **Long Method** is a method that is too long, so it is difficult to understand, change, or extend. Fowler and Beck (Fowler & Beck 2000) strongly believe in short methods.
- **Large Class** means that a class is trying to do too much. These classes have too many instance variables or methods.
- **Primitive Obsession** smell represents a case where primitives are used instead of small classes. For example, to represent money, programmers use primitives rather than creating a separate class that could encapsulate the necessary functionality like currency conversion.
- **Long Parameter List** is a parameter list that is too long and thus difficult to understand.
- **Data Clumps** smell means that software has data items that often appear together. Removing one of the group's data items means that the those items that are left make no sense, e.g., integers specifying RGB colors.
- **Switch Statements** smell has a slightly misleading name, since a switch operator does not necessarily imply a smell. The smell means a case where type codes or runtime class type detection are used instead of polymorphism. Also type codes passed on methods are an instance of this smell.
- **Temporary Field** smell means that class has a variable which is only used in some situations.
- **Refused Bequest** smell means that a child class does not fully support all the methods or data it inherits. A bad case of this smell exists when the class is refusing to implement an interface.

- **Alternative Classes with Different Interfaces** smell means a case where a class can operate with two alternative classes, but the interface to these alternative classes is different. For example, a class can operate with a ball or a rectangle class, and if it operates with the ball, it calls the method of the ball class playBall() and with the rectangle it calls playRectangle().
- **Parallel Inheritance Hierarchies** smell means a situation, where two parallel class hierarchies exist and both of these hierarchies must be extended.
- **Lazy Class** is a class that is not doing enough and should therefore be removed.
- **Data Class** is a class that contains data, but hardly any logic for it. This is bad since classes should contain both data and logic.
- **Duplicate code**. According to Fowler and Beck (Fowler & Beck 2000), redundant code is the worst smell. We should remove duplicate code whenever we see it, because it means we have to do everything more than once.
- **Speculative Generality** smell is a case, where unnecessary code has been created in anticipating the future changes of the software. Predicting the future can be difficult and often this just adds unneeded complexity to the software.
- **Message Chains** smell means a case, where a class asks an object from another object, which then asks another and so on. The problem here is that the first class will be coupled to the whole class structure. To reduce this coupling, a middle man can be used.
- **Middle Man** smell means that a class is delegating most of its tasks to subsequent classes. Although this is a common pattern in oo programming, it can hinder the program, if there is too much delegation. The problem here is that every time you need to create new methods or to modify the old ones, you also have to add or modify the delegating method.
- **Feature Envy** smell means that a method is more interested in other class(es) than the one where it is currently located. This method is in the wrong place since it is more tightly coupled to the other class than to the one where it is currently located.
- **Inappropriate Intimacy** means a smell where two classes are too tightly coupled with each other. As Fowler and Beck (Fowler & Beck 2000) put it, classes spend too much time delving in each other's private parts.
- **Divergent Change** smell means that one class needs to be continuously changed for different reasons, e.g., we have to modify the same class whenever we change a database, or add a new calculation formula.
- **Shotgun Surgery** smell is the opposite to the Divergent Change. It means that for every small change we must modify a bunch of classes, e.g., whenever we change a database we must change several classes.
- **Incomplete Library Class** smell means that the software in question is utilizing library that is not complete. This means that developers have to extend the functionality of the library.

- **Comments** are not necessarily a bad smell, but they can be misused to compensate poorly structured code.

Here we saw the bad code smells that tell the programmer when refactoring is needed. These bad code smells can be considered a measure of software maintainability, because removing them will make the software more maintainable.

4.4.4 Critique on Bad Code Smells

In the previous section, the bad code smells by Fowler and Beck (Fowler & Beck 2000) were introduced. In this section, I will show some problems related to the bad code smells. Further in this work I will then offer my own contribution to these problems.

In my opinion, the first problem comes from the way the bad code smells are presented in Fowler & Beck's work. The smells are presented as a single flat list, which makes it quite difficult to get an overview of them, because the number of the smells is so high. For human mind it is quite difficult to remember 22 separate smells. The number of the smells and the way they are presented also hinders the ability to understand the smells themselves and the relationships between them. To summarize this, when presenting a concept as complex and varying as bad code smells, a single flat list with 22 entries should not be used, but instead a more hierarchical view should be provided.

My second critique is directed at the comment where Fowler & Beck say that *In our experience no set of metrics rivals informed human intuition.* Here the authors wish to say that human judgment should always be the ultimate authority when assessing whether the smell spotted in the source code needs to be refactored out or not. I have two points I would like to make on this issue. First of all, that comment does not make automatic smell measurement obsolete. Automatic measurement can actually help a human to make a more informed and better decision on the smells and the possible need for refactoring. My second point on this issue is that relying strictly on human intuition might also be dangerous, because different people can have different opinions on when a smell needs refactoring. For instance, one developer can prefer really tiny methods, while the other developer might think that method should have at least 100 lines of code.

The problem with the bad code smells is that they lack empirical academic research. This final critique against the smells is the one that actually motivates my research. Currently the bad smells are just concepts created by famous (and most likely talented) individuals of software engineering community, but nobody has tried to actually test and investigate the smells more thoroughly.

4.5 Summary

In this chapter, various ways of preserving the maintainability of software have been presented. I also introduced cases from industry where the lack of software maintainability led to a disaster. These cases points out the importance of maintainable software. Refactoring was introduced as one of the ways of keeping the software maintainable. With refactoring we saw the concept of bad code smells that tell the software developer when to refactor. These bad code smells can be considered as one measure of software maintainability. In this chapter I also presented my critique against the bad code smells. Based on this critique I will study the bad code smells in more detail later on in this work.

5 Bad Code Smells – Taxonomy & Measures

5.1 Introduction

In the previous chapter, the bad code smells by Fowler and Beck (Fowler & Beck 2000) were introduced. In this chapter, I will study those bad code smells in more detail by providing a taxonomy for the smells and looking at the possible ways of automatically measuring the bad code smells. Taxonomy makes the smells more understandable, since currently they have been introduced only in a single flat list. Measuring the smells with tools is important, because it enhances their usability as a measure of maintainability. Smell measurement is also needed later in this work, where I compare the perceived smell evaluations from software developers with the smell metrics from the actual source code. Section 5.2 provides the smell taxonomy and Section 5.3 tries to find applicable measures for each smell.

5.2 Smell Taxonomy

5.2.1 Introduction

The motivation for creating this taxonomy lies in the critique that I presented against the bad code smells in Section 4.4.4. There I stated that the flat list of 22 bad smells makes the smells difficult to understand, fails to recognize the relationship between the smells, and does not take in to account the larger context for each smell. To address these problems I have created a taxonomy for the smells. The taxonomy in this section is created based on some of the common concepts that the smells inside one group share.

5.2.2 The Bloaters

The smells in the Bloaters category are as follows: *Long Method, Large Class, Primitive Obsession, Long Parameter List,* and *Data Clumps*. According to the smell descriptions from previous chapter, the entire group of Bloater smells represents something that has grown so large that it cannot be effectively handled. For instance, in general it is more difficult to understand or modify a single long method than several smaller methods. The same kind of argument holds also for Long Parameter List and Large class.

I argue that the smell Primitive Obsession is in the right place here, even though it does not represent a bloat. Primitive Obsession is more of a symptom which causes bloats, because if you do not create small classes for phone numbers and so on, you have to add the functionality to some other class and this increases the class and method size in the software. Therefore Primitive Obsession smell increases the number of Bloaters.

For Data Clumps I could also argue that it should be in the Object-Orientation Abusers, because in theory a class should be created from each Data Clump. However, I have decided to keep it in this category for two reasons. First of all, this smell should have a close connection with the Long Parameter List smell based on Fowler and Beck (Fowler & Beck 2000) and on my own experience. The second reason is that it might not make sense to create a class for very small Data Clumps, since this will lead to a Lazy Class smell. Therefore, I believe that in order for the Data Clumps smell to be a real problem, it needs to be of a considerable size. For these reasons I have included it in the Bloaters category.

There might be a common pattern in how these smells are created. In my previous work, I have seen a method that was 437 NLOC long and had cyclomatic complexity of 137. I do not think that any programmer would think ahead or design such a method to the program. Rather, I believe that bloater smells are born in small steps. A programmer continuously makes small changes or additions to the code, and then one day a bloater smell can be spotted, e.g., continuous increase in class or method functionality or addition of new data attributes.

5.2.3 The Object-Orientation Abusers

The smells in the Object-Orientation Abuser category are: *Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces, Parallel Inheritance Hierarchies*. The common denominator for the smells in the Object-Orientation Abuser category is that they represent cases where the solution does not fully exploit the possibilities of object-oriented design.

For example, a Switch Statement might be considered acceptable or even good design in procedural programming, but is something that should be avoided in object-oriented programming. The situation where switch statements or type codes are needed should be handled by creating subclasses. Parallel Inheritance Hierarchies and Refused Bequest smells lack proper inheritance design, which is one of the key elements in object-oriented programming. The Alternative Classes with Different Interfaces smell lacks a common interface for closely related classes, so it can also be considered a certain type of inheritance misuse. The Temporary Field smell means a case where variable is in the class scope, when it should be in method scope. This violates the information hiding principle.

According to Fowler and Beck (Fowler & Beck 2000) the Parallel Inheritance Hierarchies smell is just a special case of shotgun surgery. Still I feel it is closer to the Object-Orientation abusers category than the Change Preventers, because it clearly is a misuse of object-orientation.

5.2.4 The Change Preventers

The smells in the Change Preventers category are: *Divergent Change and Shotgun Surgery*. The common theme with the Change Preventer smells is that they prevent or hinder the changing or further developing of the software. So if the software has these smells, it has to be refactored to make it more modifiable. All bad smells make software less modifiable, but these smells violate the rule suggested by Fowler and Beck (Fowler & Beck 2000), which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class. The Divergent Change smell means that we have a single class that is modified in many different types of changes. With the Shotgun Surgery smell the situation is the opposite, we need to modify many classes when making a single change to a system.

5.2.5 The Dispensables

The smells in the Dispensables category are *Lazy class, Data class, Duplicate Code, Speculative Generality*. The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code. This group contains actually two types of smells, but since they violate the same principle, we will look at them together. Fowler and Beck (Fowler & Beck 2000) say that each class requires effort to maintain and understand, and if a class is not doing enough it needs to be removed or its responsibility needs to be increased. This is the case with Lazy class and Data class smells. Fowler & Beck also point out that code which is not used or which is redundant needs to be removed. This is the case with Duplicate Code and Speculative Generality smells.

Fowler and Beck (Fowler & Beck 2000) did not present a smell for dead code, which is quite surprising, since to my experience it is quite common. With Dead Code I mean code that has been used in the past, but is not currently used. Dead Code hinders the code comprehension and makes the structure less obvious. In my opinion, Dead Code should be included in this category.

5.2.6 The Encapsulators

The smells in the Encapsulators category are *Message Chains and Middle Man*. This category has only two smells, and they both deal with data communication mechanism or encapsulation. The smells in this category are somewhat opposite, meaning that decreasing one smell will cause the other to increase.

Data hiding or encapsulation is one of the key principles of object-oriented programming. If class A needs data from another object called B, but class A does not have a reference to B, A needs to ask the data it needs from a third object called C. This is called encapsulation, since class A does not have any knowledge of the existence of B, but A can still get data it needs from B through C. This is the basic principle that has been traditionally used to separate objects/classes from each other. According to Fowler and Beck (Fowler & Beck 2000), if class C is just doing the kind of delegation work described the example, it suffers from the Middle Man smell. This means that the class is doing too much simple delegation and does not contain real logic. In our example case above, the fix would be to remove C and make a direct connection from A to B.

Message Chains is a smell where class A needs data from class D. To achieve this data, class A needs to ask object C from object B (A and B have a direct reference). When class A gets object C it then asks C to get object D. When A finally has D it asks it for the data it needs. The problem with this smell is that A becomes coupled to class B, C, and D. One way to solve this problem is to make B and/or C a Middle Man class.

Removing the Message Chains smell does not always cause the Middle Man smell and vice versa, since the best solution is often to restructure the class hierarchy by moving methods or adding subclasses. However, I think that these two smells belong together, because they both deal with the way objects, data, or operation are accessed. Naturally, one could argue that the Message Chains smell belongs to the Couplers group and that the Middle Man smell belongs to the Object-Orientation users, but I personally believe that in order to get a better understanding of these smells they should be introduced together.

5.2.7 The Couplers

The smells in the Couplers category are: *Feature Envy and Inappropriate Intimacy.* This group has two coupling-related metrics. The Feature Envy smell means a case where one method is too interested in other classes, and the Inappropriate Intimacy smell means that two classes are coupled tightly to each other. Both of these smells represent high coupling, which is against the object-oriented design principles that emphasize minimal coupling between objects. Coupling was discussed also earlier in Section 3.2.2, where object-oriented metrics were discussed. Of course here I could make an argument that these smells should belong to the Object-Orientation abusers group, but since they both focus strictly on coupling, I think it makes the smell taxonomy more understandable if they are introduced in their own group.

5.2.8 Others

This class contains the two remaining smells that do not fit into any of the categories above: Incomplete Library Class, Comments. These smells have nothing in common, expect that they cannot be included into any of the previously introduced categories.

5.2.9 Conclusions

This section introduced a taxonomy for bad code smells. The purpose of this taxonomy is to prevent the problems arising from the flat list of 22 bad code smells. I feel that this taxonomy makes the smells more understandable, recognizes the relationships between smells and puts each smell into a larger context. This taxonomy is only initial, so it probably has weaknesses and needs to be improved in the future. Nevertheless, I think that this taxonomy fixes many problems with the current presentations of bad code smells, which were discussed in Section 4.4.4.

5.3 Measuring The Bad Code Smells

5.3.1 Introduction

As stated earlier in Section 4.4.4, measuring bad code smells automatically could help people in making decisions on refactoring. I also need measures for bad code smells later in my work where I try to compare the developers' perceived smell evaluations and the measures gathered from the actual source code. This section consists of two parts. First I will introduce the most significant prior work on measuring bad code smells and then I will try to come up with measures for each bad code smell.

5.3.2 Prior Work

There has been quite a lot prior research on combining refactoring and source code measurement. However, the approaches have been somewhat different compared to the one I will use later in this section. This section tries to summarize the prior work that is closest to my work.

Balazinska et al. (Balazinska et al. 2000) have worked on detecting clones in OO systems to support refactoring. The target of this work has therefore been the Duplicate Code smell. Maruyama and Shima (Maruyama & Shima 1999) report on a tool that uses historical data to identify the spots where programmers have changed the software. They also acclaim that their tool then creates template and hook methods that separate the invariant dependence and the variant dependence. Demeyer, Ducasse and Nierstrasz (Demeyer, Ducasse, & Nierstrasz 2000) used a set of OO-metrics to detect refactorings from software version history. They focused on finding two types of refactorings, first a case where either a subclass or superclass is created or merged, and a second case where a method is moved to other class or split into several methods of the same class. So they focused on detecting the changes either in the class hierarchy or in the method place and form. Simon, Steinbruckner and Lewerentz (Simon, Steinbruckner, & Lewerentz 2001) used a visualization tool to detect and/or assist a developer to find smells leading to four different refactorings. The refactorings, whose smells they searched were *move method, move field, extract class,* and *inline class*. Kataoka (Kataoka et al. 2001) also worked on a tool for finding suitable spots for refactorings (*remove parameter, eliminate useless return value, separate query from modifier, encapsulate downcast,* and *replace temp with query*). Kataoka also used a developer to verify the usefulness of the tool's findings. Kataoka (Kataoka et al. 2002) focused on quantitative evaluation of refactoring effects. In this work, the target was to measure the differences in three method coupling metrics before and after refactoring and then compare it to the developer's opinion on those refactorings. The refactorings that were studied here were the ones that reduce method coupling (*Extract Method, Extract Class, Move Method*).

5.3.3 Smell Measurement

As we saw in the previous section, most of the prior work on smell measurement has focused on finding places, where some of the refactorings from Fowler (Fowler 2000) could be applied. None of the above studies has used the smells presented by Fowler and Beck (Fowler & Beck 2000) as a basis. One of the reasons that the original smells presented by Fowler and Beck (Fowler & Beck 2000) have not been studied could be that most of them are quite vague. Fowler & Beck also point out that human intuition is the key for finding the smells. Thus, the bad code smells have not really been studied or measured.

First plan to smell measurement comes from the claim that there is no opposition to human judgment, when it comes to finding smells. This can be studied quite painlessly with a survey, where several developers share their opinion on a particular piece of code on each smell. Another option is to choose metrics that reflect the particular smell. In the past, some maintenance studies have compared the measured maintainability and the developers' perceived opinions on maintainability. I will use this approach later in Chapter 6.

In this section, I will relate the bad code smells against source code measures introduced in Section 3.2. The bad code smells and the related measures will be compared. I have also tried to come up with a number that indicates the measurability of each smell. This is subjectively evaluated on a scale of 0 to 5. M=0 means that the author believes that measuring is impossible, or that it yields to results which are mostly false. M=5 means that the author believes that measuring the smell is easy and will almost always bring correct results. I will also try to find the best possible measures for each smell. The criterion for selecting metrics for each smell is the expert evaluation performed by the author of this work, based on his understanding on the bad code smells. Since I do not have complete knowledge of all the possible source code metrics, I will not cite the original authors of each measure. This is because I have used some combined metrics, but it is very likely that someone has already invented these metrics earlier. This way I will not discriminate based on my limited knowledge on different source code measures. Most measures that I propose for smells are the ones previously introduced, or a modification or extension of them. I also propose quite a few polynomial measures that take into account several metrics. However, I cannot say anything about the weight of the polynomials in the scope of this work.

Long Method. Measuring long methods should be quite easy. However, relying on too simple a size measure such as Lines of Code (NLOC) will definitely bring a wrong result, because e.g., initiation methods can often be quite long. There is no sense in refactoring long initiation methods, because they often do not have any cyclomatic complexity and therefore they are very easy to understand and modify. Cyclomatic complexity and Halstead measures, which measure the number of operators and operands, can provide necessary information on method complexity. Therefore I conclude that measuring this smell is quite easy, and this results in a measurability value of 5. In my opinion, the best metric for this smell is a polynomial metric that combines NLOC, Cyclomatic Complexity, and Halstead metrics.

Long Parameter List. Measuring this smell is very easy. It can be done simply by counting the number of parameters of each method. Measuring the type of parameters (primitives, classes) with this smell can additionally help the refactoring process, since redundant primitive lists might make good candidates for new classes. Thus, measurability of this smell is also 5.

Large Class. Many measures for measuring class size have been introduced in the past. Traditional way of measuring class size has been to measure the number of attributes and methods. I personally feel that the best measure of class size is class cohesion. Class cohesion metrics such as Lack of Cohesion Methods can sometimes be difficult to calculate. In such cases, the number of methods and attributes should be used as a measure of class size. Simple size measures such as (NLOC) can tell the size of the class, but they offer no help on whether the class is doing too much or not. Measurability of the Large Class smell is 4. This is because GUI classes and other special cases can make the definition of Large Class more elusive.

Feature Envy. Measuring this smell can be done by measuring the strength of couplings that a method has to methods or data of foreign classes. Measuring this smell should be quite easy with proper tools. There are no other measures for detecting this smell. Since I do not have experience in this type of measurement, I conclude that the measurability of this smell is only 4 instead of 5.

Inappropriate Intimacy. This smell needs to be measured by measuring the strength of coupling between two classes. This smell is particularly bad if classes are accessing each other's fields. Measuring the strength of coupling should lead to quite good results, thus the measurability of this smell is 4.

Duplicate Code. Measuring duplicate code is easy. It can be done by measuring the percentage of duplicate code lines in the system. The problem with measuring this smell is that the best tools for this kind of work need compiler type of functionality and are thus quite expensive. Nevertheless, with proper tools measuring should be quite applicable, and thus the measurability is 4.

Data Class smell can be measured by comparing the number of fields in a class with the cyclomatic complexity of the class. I believe that this proposed measure should work quite well, since Data Classes typically contain many fields compared to their complexity. Although I have not ever heard of combining these two metrics, I evaluate that the measurability of this smell is 4.

Lazy Class. This smell should be quite easy to measure by looking at the number of fields and methods in a class in conjunction with cyclomatic complexity. Even the simple NLOC metric for a class might also bring good results. Even though many good measures are available, I believe that there might be some false positives with this smell. The measurability of this smell is therefore 4.

Message chains. There are two possible ways of detecting this smell. One way is to measure the number of separate class couplings that a method has. With the Feature Envy smell I propose measuring the strength of coupling that a method has to other classes. If a method has couplings to several classes, we can suspect a message chain. However, since the detection of this smell relies somewhat on speculation, I estimate that the measurability of this smell is only 3.

Switch Statements. Using the length (NLOC/Cyclomatic Complexity) of conditional statements to detect the smell might not work, since the intention of the switch statement is the key for finding this smell. For this smell to exist, there need to be type-codes that are detected using switch statements. A special case for this smell could be a situation where runtime type detection is utilized instead of type-codes. Since both measures (long conditional statements and runtime type detection) can only provide hints that this smell may exist, I conclude that the measurability of this smell is 3.

Speculative generality & Dead Code. I previously concluded that Dead Code should be considered a bad code smell, although it is not included in the original list. These two smells have exactly the same measures, because in my opinion the Speculative Generality smell is a special case of Dead Code. There are two ways of measuring these smells. If measurements show that the number of references to a method or a class is zero, we can suspect this smell. With static tools it is not possible to detect dynamic class loading, and thus static detection might lead to wrong results. Dynamic detection, which shows the parts of software that have been executed, can also help in detecting this smell. Dynamic detection also has its own problems, because it is possible (or in most cases very likely) that not all the code is executed during the test. However, such code fragments could be error-reporting routines that are executed in cases of rare errors and cannot therefore be removed. I believe that by combining static detection with dynamic detection this smell can be effectively measured. Since combining the two types of measurements is quite tricky, the measurability of this smell is only 3.

Temporary Field. This smell can be measured by counting the different methods that access each field and by comparing that number to the total number of methods in the class. I believe that such a measure will bring quite good results. However, I believe that there can be some false positive detections of the smell. Therefore the measurability is only 3.

Middle Man smell can be suspected, if a class has many methods that are coupled to exactly one class and the methods also have a low cyclomatic complexity. Such measures for a method can indicate that the method is only doing simple delegation, which means that it has a Middle Man smell. There can be several other explanations why a method is coupled to exactly one class and has a low cyclomatic complexity. Based on that, I believe that the measurability of this smell is only 2.

Comments. Measuring the number of comment lines could be pointless since comments are not always used in a bad way. However, if there are many comments in the middle of the method, we can suspect the abuse of comments. Still, measuring this is very speculative and could be in fact useless, and therefore the measurability is 1.

Data Clumps. Detecting the repeating groups of primitives that belong to the same group and have the same intention is nearly impossible. The best way to find this smell is to look at the spots where the Long Parameter List smell appears. This is because Data Clumps often form Long Parameter Lists, when they are passed as parameters. However, since I cannot offer any direct measures for this smell, I must conclude that the measurability of this smell is 0.

In addition to the smells already presented above, I must admit that I have no suggestions for measuring the following smells, and thus they all have measurability of 0: **Alternative Classes with Different Interfaces, Refused Bequest, Incomplete Library Class, Primitive Obsession, Parallel Inheritance Hierarchies, Divergent Change, Shotgun Surgery.**

5.3.4 Conclusions

The measurability of the code smells was discussed in this section. First I looked at the most significant prior research and found out that nobody has studied the measuring of the bad code smells introduced by Fowler and Beck (Fowler & Beck 2000). After that I tried to propose the best measures for each smell based on my knowledge and understanding of source code measures and the bad code smells. This work was important for two reasons. First, measuring bad code smells can help the developers when they decide whether to refactor a certain part of the software. Second, I need to measure the bad code smells from source code automatically in order to compare them with the developers' perceived smell evaluations.

Based on the smell measurement study I conducted, I can conclude that there is a great fluctuation on how measurable the different smells are. On one hand, there are eight smells where measuring should be very beneficial (measurability 4 or above). On the other hand, there are also eight smells that cannot be measured at all. I can also say that a little over half of the smells could be effectively measured. This is based on the conclusion that 13 out of 23 code smells (smells defined by Fowler & Beck plus the Dead Code smell) have measurability of three or above.

Finally, I must admit that the measures proposed for each smell and the smell's measurability should be considered initial, since they are based on the literature and no empirical study was made in order to verify the measures for each bad code smell.

5.4 Summary

This chapter has shown my contribution to bad code smells. I have provided a taxonomy of 7 categories for the bad code smells, and I believe that this taxonomy makes the smells more understandable, recognizes the relationships between smells, and puts each smell into a larger context. This taxonomy is based on some of the concepts that the smells in the same category share. This is an improvement, because the original presentation of the bad code smells only consisted of a single flat list that caused problems when trying to understand smells and failed to describe their relationships to each other. These problems were discussed in Section 4.4.4. I also proposed and discussed possible measures based on my knowledge of bad code smells and source code metrics. Finding source code metrics for bad code smells is imperative, because metrics can help the developers decide, when refactoring is needed. Later in this thesis I also need to measure bad code smells from source code and compare them to developers' perceived smell evaluations. From the smell measurement study we learned that little over half of the smells can be effectively measured and that the measurability of smells fluctuates considerably. The weakness of the results is that they are based only on literature study and therefore they lack empirical validation.

6 Smell Survey

This chapter describes the smell survey conducted in the case company BeachPark. In Section 6.1 I will introduce the survey and some basic results from it. In Section 6.2 I will look at the correlations between smell evaluations. Section 6.3 will show how the background variables affected the developers' smell evaluations. In 6.4 I will use the data from the smell survey and try to evaluate its reliability. Section 6.5 compares the smell evaluations and the source code metrics, and thus provides more information on the reliability of smell evaluations.

6.1 Introduction

6.1.1 Introduction of the Case Company: BeachPark

The survey was conducted with the case company called BeachPark. BeachPark is a small Finnish software product development company. At the time of the survey, the company employed about 20 software developers. The company had developed two software products in the last 4-5 years and during that time some parts of the products had become quite complex. The products that the company develops are not domain-specific, which means that organizations and people from different domains can use them. The development language used in the company is Delphi, which is an object extension of PASCAL programming language. The software modules and their sizes are displayed in Table 1. The module sizes marked with an asterisk are estimates made by the case company employees. However, when measuring the actual size of the modules, I noticed that the estimates tended to be a bit too high. This means that estimated sizes should be dealt with caution. The core modules of the product start with Gamma and Delta. The other modules are shared between two products.

Table 1 The modules and their size in the case company

Module name	Size in LOC
Gamma-C	47058
Gamma-P	28589
Gamma-S	16868
Delta-C	55342
Delta-P	-
Delta-S	54780
Epsilon-P	80000*
Zeta-C	30000*
Zeta-S	20000*
Kappa-S	20000*
Omega-X	-

6.1.2 Introduction to the Survey

In this section I will explain the setting of the survey more thoroughly. The survey was a web-based survey consisting of two parts. In the first part, each respondent first provided their background information, including age, location (organization had software developers at two locations), role (developer or a lead developer), education, work experience in the company, and overall software development work experience. After that the respondents selected the software modules they had primarily worked with. This information was used in the second part of the survey. In the second part of the survey, the respondents were asked about the code smells in the modules they had selected in the first part. The code smells were described with a definition and an example which had a combined length of no more than 35 words. Evaluation of each smell was asked for each module. The scale was from 1 to 7, where 1 meant that the current smell did not exist in the module at all and 7 meant that there was a lot of the current smell present in the module. The respondents could also select "I don't know" or "I don't understand the smell explanation". The "I don't know" option was checked by default to prevent wrong smell evaluations from getting into the data. The respondents also estimated how well they knew each module they had selected. That scale was also from 1 to 7 where 1 meant "I know the module very poorly" and 7 meant "I know the module very well". The survey questions can be found in Appendix A.

In the survey the sample consisted of 12 developers. The total number of people in the population consisted of 18 software developers working in the case company. The response rate was pretty good with two-thirds of the possible respondents answering the survey. The sampling technique was opportunistic, i.e., whoever answered the survey became a part of the sample. The sampling technique certainly was not optimal, and in further studies it should be improved.

One problem was that the survey was a so-called "cold turkey" survey. After conducting the survey I learned that the proper way to conduct a survey is to talk with the respondent in order to make sure he/she understands the questions correctly and is actually able to answer the question. However, such "cold turkey" surveys are commonly used in many different settings, and therefore it cannot be stated that the survey was completely worthless, but the results should be studied with caution.

Since the survey in this study was "a cold turkey survey", I cannot tell whether the developers answered the question based on their recollection of the smells, or whether they actually looked at the source code. However, as people generally are lazy, it is likely that most of the developers answered the survey based on their recollection.

I received a total of 37 module smell evaluations from the sample 12 developers. Thus, the average number of modules covered was little over 3, but the actual number varied from 2 to 6.

Before continuing any further, I suggest looking at Table 2 to understand the terms used in this study

Table 2 Terms used in the study

Term	Explanation
Smell evaluation	Evaluation of one smell in one module from one evaluator (=developer).
Smell mean for a module	This is a number received by calculating the mean from a particular smell based on the smell evaluations of a single module
Smell mean for a developer	This is a number received by calculating the mean from a particular smell based on the smell evaluations of a single developer
Combined smell mean for a module	This is a number received by calculating the mean of all smell evaluations of a single module
Combined smell mean for a developer	This is a number that is received by calculating the mean of all smell evaluations of a single module

6.1.3 Introduction to Results

Table 3 shows the work experience and age of the sample population. The respondents were quite well educated, since all were studying at least for a bachelor's degree and most of the respondents were studying for a master's degree. One third of the respondents had already finished their studies. There were four lead developers and eight regular developers. The division by location was a bit unbalanced, as only three answers came from one location while the other had nine. The effects of the role and work experience are discussed in more detail in Sections 6.3.1 and 6.3.3.

Table 3 Demographic data about the respondents. All numbers are in years

Information	Mean	Std. Dev	Median	Low-High
Age	28,67	5,280	26,5	23 - 40
Work experience at the case company	3,42	1,903	2,79	1,42 - 7,00
Overall programming work experience	4,74	2,514	3,46	2,50 - 10,00

To understand the value of the results, we must study how the respondents understood the different smells. In some cases, the respondents selected the "I don't know" option for all modules with a particular smell, and in other cases they selected it for just one module. This could indicate that if all modules were checked with "I don't know" option the developer had not paid attention to the smell in question. Explaining why "I don't know" was chosen for a single module only is more difficult. The "I don't understand" option was chosen only once by one developer.

Table 4 shows data for the smells that were most frequently left without evaluation, i.e., the smells that had the "I don't know" or "I don't understand" option selected more than four times (out of 37). The fact that Alternative Classes with Different Interfaces and Refused

Bequest smells are in the table is not very surprising, since recalling such structures can be difficult. The explanation why the Data Clumps smell is the number one smell that could not be understood or found, is most likely the inadequately formulated question in the survey (see Appendix A). In my opinion, the Data Clumps smell should be quite easy to understand.

Table 4 The data for smells that were not evaluated

Smell Name	**Total number of responses with**	
	Don't understand	Don't know
Data Clumps	3	10
Alternative Classes with Different Interfaces	0	7
Refused Bequest	0	6

Table 5 displays the answer percentage for all smells. In this case, the answer percentage means the number of responses (ranging from 1 to 7) to a smell. For most smells the answer percentage is close or above 90%, and hence I believe that most of the smells were understood, which increases the reliability of the results. However, this data does not give any indication of how correctly the smells were understood.

Table 5 Answer percentage for the smells

Smell Name	**Percentage**
Long Parameter List, Duplicate code	100,0
Long Method, Large Class, Message Chains, Middle Man, Lazy Class, Primitive Obsession, Temporary Field, Shotgun Surgery	97,3
Dead Code, Speculative Generality, Feature Envy, Switch Statements	94,6
Comments, Incomplete Library Class	91,9
Parallel Inheritance Hierarchies, Divergent Change, Data Class, Inappropriate Intimacy	89,2
Refused Bequest	83,8
Alternative Classes with Different Interfaces	81,1
Data Clumps	64,3

6.1.4 Investigation on the Worst Smells in Different Modules

Table 6 shows the combined smell means for modules. In the table we can see that the two modules with most respondents (Gamma-P & Gamma-C) also have the highest combined smell mean. The differences in the combined smell means are quite small, and F-test from ANOVA also shows that differences between groups are not significant with a p-value of 0,097.

Table 6 Combined smell means for modules

Module name	n	Combined smell mean
Gamma-P	6	3,51
Gamma-C	5	3,49
Omega-X	1	3,45
Delta-S	4	3,23
Delta-P	4	3,23
Delta-C	3	3,04
Epsilon-P	3	2,96
Kappa-S	3	2,93
Zeta-S	4	2,76
Zeta-C	3	2,69
Gamma-S	1	1,60

In this survey, the worst smells seem to be Long Method, Large Class, Duplicate Code, Inappropriate Intimacy, Message Chains, and Primitive Obsession. All smell means can be seen in Appendix B.

In Appendix D we can see how much of each smell exists in different modules. The three worst smells per module are also displayed in Table 7. From the table we can see that different modules actually contain different types of smells. Based on the combined smell mean, the worst modules (Gamma-P and Gamma-C) both seem to contain a lot of bloater type smells. The Gamma-P and Gamma-C have the highest smell means with Long Method and Large Class smells. The Gamma-P module also has the highest smell means in Long Parameter List and Data Clumps smells. The Delta-S module contains different smells and it has a lot of Inappropriate Intimacy and Shotgun Surgery smells, which indicate improper class architecture. The Delta-S also has a lot Switch Statements smell, which indicates the usage of type codes instead of inheritance. The Delta-C seems to contain the most Duplicate Code smell when compared to other modules. The Zeta-C, which has quite a low combined smell mean in Table 6, contains the most Divergent Change smell, which is difficult to explain. The Kappa-S contains the most Dead Code, Middle Man, and Speculative Generality smells. Discussion with the case company also revealed that this module was built to handle a lot more features than is currently needed. This explains nicely the high values on Dead Code and Speculative Generality smells. The Epsilon-P module has by far the highest mean on the Incomplete Library Class smell, which indicates that the Epsilon-P has suffered from buggy or incomplete library classes.

Table 7 Three worst smells per modules

Module Name	**Smell**	**Worst Smells**		
		1st	**2nd**	**3rd**
Gamma-P	Name	Long Method	Large Class	Message Chains
	Mean	5,17	5,17	4,83
Gamma-C	Name	Long Method	Large Class	Message Chains
	Mean	6,00	5,20	4,80
Delta-S	Name	Inappropriate Intimacy	Switch Statements	Shotgun Surgery
	Mean	4,75	4,67	4,50
Delta-C	Name	Duplicate Code	Long Method	Temp Field
	Mean	4,67	4,67	4,00
Zeta-C	Name	Divergent Change	Inappropriate Intimacy	Duplicate Code
	Mean	5,00	4,00	3,67
Kappa-S	Name	Message Chains	Dead Code	Speculative Generality
	Mean	4,67	4,33	4,33
Epsilon-P	Name	Incomplete library Class	Message Chains	Speculative Generality
	Mean	5,33	4,00	3,67

I also used ANOVA to analyze how significant the differences in smell means between groups (modules) were. The smells with significant differences between modules according to F-test from ANOVA are displayed in Table 8. The smells in Table 8 are the smells that made the best distinction between modules. Based on this I can also speculate that those are the archetype smells that find the biggest difference between modules and that other smells are not so significant because they cannot give information on the differences between modules. However, because of the small amount of data, I will not make strong conclusions based on these results.

Table 8 The ANOVA test results between modules

Smell Name	p-value
Large Class	0,001**
Long Method	0,004**
Incomplete Library Class	0,011*
Comments	0,027*
Speculative Generality	0,038*

In this section, we have seen that the differences in the combined smell means are quite small and that this difference is not significant. On the other hand, the differences in the worst smells per module are quite large. We also saw how some smells were more significant than others in finding the differences between modules. This indicates that some smells could be more useful than others in measuring the different aspects of software's maintainability.

6.2 Investigating Smell-to-Smell Correlation

It seems natural that some smells correlate with each other, while others have a negative correlation. By this I mean that it would seem natural that if one class has a method that suffers from the Long Method smell, the same class would suffer from Large Class smell. The idea of negative correlations comes from the conclusion that if the class size is one attribute, then at one end of this attribute the class suffers from the Large Class smell, and at the other end from the Lazy Class smell.

The correlations, or actually the correlation coefficients, tell how strong the relationship between two variables is, e.g., if variable A and variable B have a strong correlation, then low value on A will mean that there is a great chance that the value of B will also be low. Correlations are measured on a scale of –1 to 1. Zero indicates no correlation, and one indicates complete correlation. The sign of the correlation indicates whether the increase in one variable will lead to increase or decrease in the other variable. To find out the correlations I used Spearman's Rho, which is applicable here, because the data was measured on Likert scale. Spearman's Rho does not correlate the variables based on their absolute value, instead it looks at the rank of the variable against other variables. This makes it applicable with the Likert scale used in the survey.

In this study, the only significant negative correlations were found with the Primitive Obsession smell, and this is most likely due to the fact that the smell was measured on a reversed scale. I present the most significant correlations in Figure 5, and the whole list of correlations can be found in Appendix C. In Section 5.2 I introduced the smell taxonomy that should help understand the smells and put them into a larger context. I have applied this taxonomy to the smell correlations as well.

In Figure 5 we can see that the strongest correlation is between the Long Method and Large Class smells. This is probably the most anticipated correlation. The Large Class smell also correlates with the Message Chain smell. The explanation to this could be that large classes need data from many different sources, because they are involved in many operations. The Message Chain smell also correlates with its opposite smell, Middle Man. This could be caused by the fact that both smells indicate encapsulation or lack of it between objects and can therefore be easily confused. This could also be caused by the way the questions were constructed. If we look at the questions in Appendix A, we can see that the question related to the Message Chain smell is actually formulated in a way that is actually closer to the Middle Man smell. So this can be interpreted as an error in the formulation of the questions.

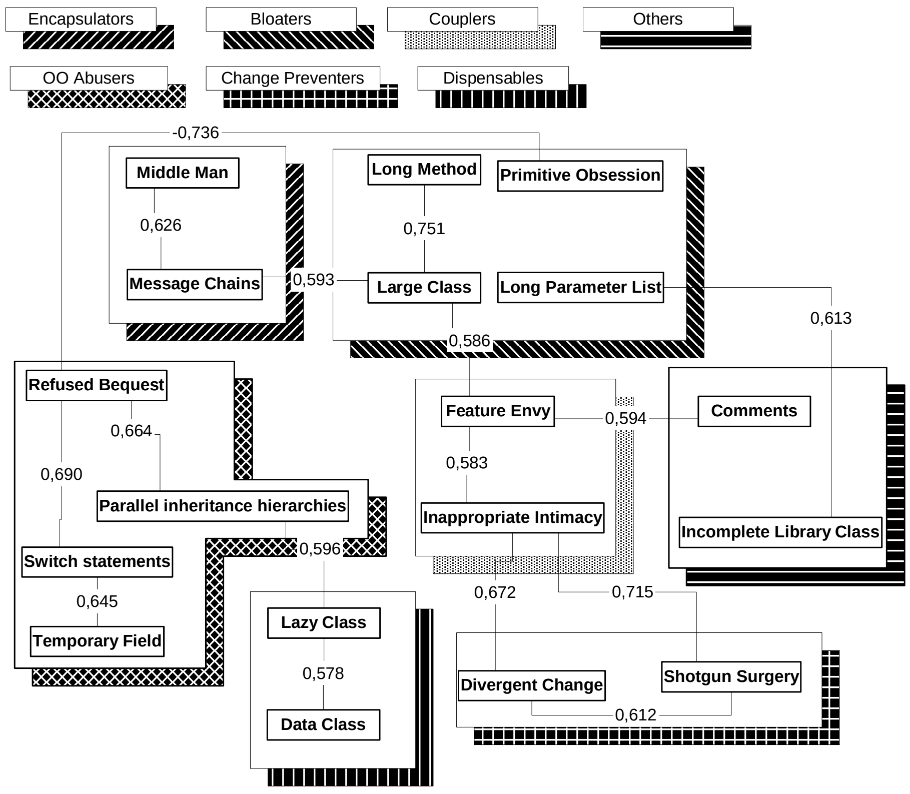

Figure 5 Spearman's Rho Correlation cofficient above 0,575 between smells. All correlations are significant with the p-value of 0,01 or greater

The Large Class smell also has a correlation with the Feature Envy smell, which means that a method is highly coupled to other objects. This indicates that large classes also create problems with high coupling. The Feature Envy smell is also correlated with another coupling-related smell, Inappropriate Intimacy, which is what I expected. It seems that the Inappropriate Intimacy hooks the Change Preventer smells together, since it has a strong correlation with both of them. Change Preventers are also correlated, but the correlation is not as strong as the correlation between them and the Inappropriate Intimacy smell. These correlations are not surprising, since it seems reasonable to think that while we have high coupling and large classes it becomes difficult to have one-to-one relationships between common changes and classes like Fowler and Beck (Fowler & Beck 2000) suggest.

Parallel class hierarchies also seem to cause the Refused Bequest smell, where a child class refuses to properly implement all inherited methods. When a class refuses to implement inherited behavior, we can think that this will cause the need for various type checks, which is represented by the Switch Statements smell. The Parallel Inheritance Hierarchies smell also seems to connect the Dispensables to the Object-orientation Abusers with the Lazy Class smell. It is easy to understand how duplicated class hierarchies can cause the Dispensable classes to appear. The correlation between the Temp-field smell and the Switch Statement smell is more difficult to grasp. I cannot offer an explanation to the strong correlation between the Refused Bequest smell and the Primitive Obsession smell, either.

No support to the claims of Fowler and Beck (Fowler & Beck 2000) regarding the correlation of the Large Classes and Duplicate Code smell was found. However, we must bear in mind that the Duplicate Code smell is difficult to spot. The Duplicate Code smell also had low deviations, meaning that developers felt it was distributed quite evenly across different modules.

In Figure 5 we can identify two more distinct groups. One group is formed around the Couplers and the Large Class smell. The other group seems to be formed around the Object-orientation Abusers. One could assume that large classes and inability to minimize coupling could cause the first group. The second group seems to focus on poor inheritance usage and dispensable classes. This indicates that understanding of when and how to use inheritance and objects, and remembering the two basic principles of minimizing coupling and maximizing cohesion (Stevens, Myers, & Constantine 1974) helps preventing smells.

Comparing the taxonomy from Section 5.2 and the correlations shows that Figure 5 has 8 correlations within groups and also 8 correlations between groups. To appreciate this information we must see, what the number of possible correlations within a group is and compare it to the number of possible correlations between groups. The total number of possible correlation lines comes from the following formula.

$$\frac{n*(n-1)}{2}$$

In this case *n* gets a value of 23, and thus the total number of correlations is 253. The maximum number of correlations within all groups comes from adding together the number of each individual group's within-group correlations, and this results in 34 ((The Bloaters n=5) 10, + (The Object-Orientation Abusers n=5) 10+ (The Disbensables n=5) 10+ (The Change Preventers n=2) 2+(The Couplers n=2) 2+(The Encapsulators n=2) 2+ (Others n=2) 2). The total number of between-group correlations is the number of all correlations minus the number of within-group correlations, and this results in 219 (253-34).

Based on the calculations presented above, I can conclude that the correlations add support to my theoretical taxonomy. This is because the strongest correlations, shown in Figure 5, represent 23,53% (8/34) of the total amount of within-group correlations. The amount of between-group correlations among the strongest correlations is only 0,04% (8/219). Therefore, my theoretical taxonomy is supported by the data from smell correlations, since a relatively larger number of within-group correlations are strong compared to between-group correlations. This means that the smell taxonomy can provide help for using the smells, since the groups from the taxonomy can be used instead of all smells.

6.3 Background Variable Analysis

This section will try to analyze the gathered background information with the developers' opinions of the smells. I will not analyze the effect of age, because analyzing variables like race, sex or age is hardly reasonable in this kind of study. Studying the location of developers is also left out, because there was a great difference in the modules developed at different sites. Therefore, it is likely that the differences between module quality would pollute the results of analyzing the smell evaluations from different sites. I also decided that the education will not be studied either, since the developer's role in the company and the knowledge of the module are likely to be much more informative.

6.3.1 The Effect of Role

In this section, I will try to analyze how the role affects developers' opinions, i.e., if there is a difference between developers and lead developers. In this study the company had two roles for software developers, lead developer and developer.

The developers had an average combined smell mean of 3,21, while lead developers had 3,03. This does not reveal the whole truth, since different people answered to different modules. Unfortunately, we only have two modules where there are answers from more than two developers and lead developers. Thus, we can only compare these two modules. The data presented in Table 9 indicates that lead developers actually evaluate the smells higher when comparing the same module. However, because for the module Gamma-P the significance of *t*-test is only 0,169 and for the Gamma-C it is 0,394, it cannot be concluded that there would be a difference in the smell evaluations of developers and lead developers.

Table 9 Role-based smell combined mean in two software modules

Role	**Combined smell mean in modules**	
	Gamma-P	Gamma-C
Developers	3,15	3,30
Lead Developers	3,87	3,78

Table 10 shows the differences in smell means between developers and lead developers. A positive difference means that developers evaluate that smell higher, while negative difference is in the smells that lead developers evaluate higher. From the table we can see that lead developers evaluate some smells much higher than developers, while developers evaluate other smells higher. The regular developers seem to think that there is more duplication and unused (dead) code and temporary fields than lead developers. On the other hand, lead developers seem to think that there are more structural problems, such as parallel inheritance hierarchies and shotgun surgery. The numbers of respondents related to the data in Table 10 are displayed in Table 11.

Table 10 also shows the p-values for the *t*-tests. This reinforces the assumption that high evaluation on the Parallel Inheritance Hierarchies smell is more likely to come from lead developers. Shotgun Surgery's p-value is relatively low from all modules, but if we look at the numbers from the Gamma-P and Gamma-C modules, we can see that it appears to be significant after all. From the smells that regular developers see more only the Duplicate Code is significant in all modules, but not in the Gamma-P and Gamma-C. However, the other two smells (Dead Code, Temp Field) that regular developers have evaluated more are not significant

Table 10 Greatest smell mean differences (developers minus lead developers)

Smell	**Difference in smell means in modules *t*-test results**					
	All Modules		Gamma-P		Gamma-C	
	Diff.	p-value	Diff.	p-value	Diff.	p-value
Duplicate Code	0,82	0,031*	0,67	0,579	1,00	0,495
Dead Code	0,85	0,056	1,00	0,535	1,50	0,148
Temp Field	0,83	0,058	0,33	0,686	0,83	0,473
Parallel Inheritance Hierarchies	-0,78	0,007**	-2,00	0,013*	-2,17	0,032*
Shotgun Surgery	-0,54	0,139	-2,00	0,013*	-1,67	0,032*

Table 11 Number of answers based on the role in different modules

Role	**All Modules**	**Gamma-P**	**Gamma-C**
Answers Developers	20	3	3
Answers Lead Developers	17	3	2

This data indicates that lead developers are more likely to see more smells related to structural issues. Because the smells reported by regular developers are not significant, we cannot say anything about them. However, based only on the mean differences, we can assume that developers seem to see more issues on the code level. This would match the assumption that developers see more issues on the code level. Lead developers, on the other hand, look at the system from a higher level and focus on structures.

6.3.2 The Effect of Knowledge

This section examines the effect of knowledge of the particular module on the smell evaluations given for that module. The knowledge variable related to the software modules in question was measured from each respondent. The scale was from 1 to 7, where 1 indicated the lowest amount of knowledge and 7 the highest. To make this variable more analyzable I formed two groups, one with a knowledge value between 3 and 5 (low knowledge) and the other with knowledge values 6 and 7 (high knowledge). There were no answers where the developer's knowledge of a module was under 3.

The average combined smell mean in the low knowledge group was 3,08 and 3,16 in the high knowledge group. There were only two modules with more than 2 developers from each knowledge group. Unfortunately, the modules were the ones already studied in the previous section, and there was only one developer and lead developer, who got into separate groups. Thus, the results would resemble very closely the data displayed in Table 10.

The smells that have significant differences between in the two knowledge groups are displayed in Table 12. I have included smells with p-value higher than 0,15 and mean differences greater than 0,5. The reason for including insignificant p-values is to show which other smells are close to being significant. All mean differences in Table 12 are negative, which means that the higher smell values are coming from the high knowledge group.

Table 12 The smell mean difference (low minus high knowledge group) and p-values for the *t*-test between two knowledge groups

Smell name	Mean difference	p-value
Inappropriate Intimacy	-0,89	0,095
Lazy Class	-1,02	0,007
Middle Man	-0,61	0,097
Divergent Change	-0,67	0,149

The smells that receive higher means from the lower knowledge group are not even close to being significant. However, it seems that the smells that the lower knowledge group evaluates higher are typically simple smells, such as the Long Method. The smells that the high knowledge group evaluates higher seem to be more complex. The only smell that is significant is the Lazy Class, which is not complex itself. However, in my opinion, knowing the existence of this smell requires more knowledge of the software system. This way it seems natural that it is recognized more widely in the higher knowledge group.

6.3.3 The Effect of Work Experience

Work experience averages within the sample were quite high as we can see in Table 3. The shortest work experience among the respondents was 17 months, so we can say that there were no "green" software developers among the respondents. All developers also had a minimum of two and half years of overall software development work experience. So there were no developers straight from the school, either. Since all developers were quite experienced, it is not possible to compare the smell means between experienced and inexperienced developers.

However, there were two developers who had been in the company for a very long time (nearly eight years). Both of these developers had been in the company for the entire lifetime of the current software modules. Comparing their smell means to the rest of the developers provided interesting results. These two developers had evaluated smells for a total of 11 modules. Therefore, their answers represent a quite wide range of the software modules that were under evaluation. The most interesting results are summarized in Table 13. It shows that the developers with the longest work history in the company generally thought the code was less smelly. The difference in the combined smell mean is very big and the p-value shows that this is also significant. It easy to understand that the most experienced developers can think that the code is less smelly, because it is likely that they have written very large portions of it.

The smells that had the greatest and most significant differences are listed in Table 13. The Large Class smell had by far the greatest difference in mean. This may indicate that the two developers, who possibly had developed most of the class hierarchy, refused to see that their classes had grown too big and needed to be split up.

Table 13 The smell mean difference and p-values from *t*-test (two oldest developers minus the rest)

Smell	Mean difference	p-value
Combined smell mean	0,6103	0,011
Large Class	1,42	0,004
Long Parameter List	1,09	0,021
Feature Envy	0,98	0,015

6.4 Trustworthiness of The Smell Opinions

In the previous sections the results from the smell survey have been discussed. In this section I try to evaluate how reliable the results are.

6.4.1 Investigation of the Uniform Opinion on the Smells

This section tries to investigate how uniform the developers' opinions are on each smell in the different modules.

Developers' opinions on the Long Methods are displayed in Table 14. The cross tabulation shows us that developers' opinions are not always uniform (modules Delta-C, Zeta-S), but we can also see cases where opinions are nicely matched (modules Gamma-C, Epsilon-P, Zeta-C).

Table 14 The cross tabulation of Long Method smell and modules

Module name	**Number of answers on a scale of 1 to 7**						
	1	2	3	4	5	6	7
Epsilon-P		1	2				
Gamma-P				2	1	3	
Delta-S			1	2	1		
Delta-C		1				2	
Delta-P		1		1	1		
Zeta-C		2	1				
Zeta-S		1	1	1		1	
Kappa-S		1	1	1			
Gamma-C						5	

The opinions on the Inappropriate Intimacy smell, in Table 15, do not seem to make any clusters, and thus the developers' smell evaluations on this smell are not consistent at all.

Table 15 The cross tabulation of Inappropriate Intimacy smell and modules

Module Name	**Number of answers on a scale of 1 to 7**						
	1	2	3	4	5	6	7
Epsilon-P			2	1			
Gamma-P		2	1	1	1		1
Delta-S			1		2	1	
Delta-C			1		1		
Delta-P		1		1	1		
Zeta-C				1			
Zeta-S		3			1		
Kappa-S		1	1	1			
Gamma-C		2		1	1		1

To save us from studying all cross tabulations in detail, it is more interesting to concentrate on the standard deviations. Table 16 shows the standard deviations of the smell means for the different modules[8]. For all smells there is a module that has very low standard deviation, which means that developers' opinions about the smell in that module are uniform. On the other hand, some module smell evaluations contain very high standard deviations, e.g., the Switch Statements smell, where the smell mean for one module has a standard deviation of 3,06. In this case, we cannot say anything about how much the module really contains this smell, because the developers' opinions are so conflicting. Another point seems to be that the standard deviations do not really change, if we remove people who reported their knowledge was lower than 6

[8] I have removed the modules with just one developer as there can be no std. dev for those modules

.

Table 16 The Std. Dev. of smell means for different modules

Smell Name	**Std deviations of smell means**		
	Max	Min	Mean
Long Methods	2,31	0,00	1,06
Large Class	1,53	0,00	0,89
Long Parameter List	2,06	0,00	1,03
Data Clumps	1,53	0,00	0,73
Duplicate Code	1,53	0,50	1,14
Dead Code	2,12	0,58	1,29
Speculative Generality	1,53	0,50	0,87
Feature Envy	1,38	0,00	0,76
Inappropriate Intimacy	2,12	0,58	1,42
Message Chains	1,73	0,58	1,23
Middle Man	1,41	0,58	1,03
Lazy Class	2,31	0,00	0,98
Data Class	2,00	0,58	1,23
Incomplete Library Class	2,63	0,58	1,09
Primitive Obsession	1,64	0,58	0,98
Switch Statement	3,06	0,00	1,72
Temp Field	2,71	0,00	0,89
Refused Bequest	1,53	0,58	0,75
Alternative Classes with Different Interface	1,41	0,58	1,02
Parallel Inheritance Hierarchies	1,30	0,00	0,68
Divergent Change	1,75	0,50	1,09
Shotgun Surgery	1,30	0,00	0,93
Comments	1,38	0,00	0,72

6.4.2 Investigating of the Individual Developer Profiles

In this section I will look at how each individual developer answered on each smell. If a developer's opinion on a particular smell is uniform across the different modules, this could indicate one of two things. First, it could mean that the developer truly feels that the modules in question have the same amount of smelliness. Second, it could mean that the developer really does not have enough vision, understanding, or motivation about the module or the smell in question to make this evaluation.

The answers based on the Long Method, in Table 17, do not indicate that the developer would have more effect on the answers than the module. However, answers based on the Inappropriate Intimacy, in

Table 18, give us some indication that developer's person could give some explanation to the scattering of the smell evaluations.

Table 17 The cross tabulation of the Long Method smell and developers

Developer	**Number of answers on a scale of 1 to 7**						
	1	2	3	4	5	6	7
1				2		1	
2					1	1	
3		2	2		1		
4				1		1	
5			2	1		1	
6		1	1	2	1	1	
7		2	1				
8				1		1	
9		1				1	
10						2	
11		2			1		
12						2	

Table 18 The cross tabulation of the Inappropriate Intimacy smell and developers

Developer	**Number of answers on a scale of 1 to 7**						
	1	2	3	4	5	6	7
1					3		
2			1		1		
3		1	1	1		1	
4		3					
5		2		2			
6		2	2		2		
7			1	1		1	
8			1	1			
9				1	1		
10							2
12		2					

Table 19 shows the standard deviations of smell means for different developers. Table 19 is very similar to Table 16, but the difference is that the smell mean standard deviations are from the developers, not from the modules. In Table 19, we can see that for every smell there is a developer who could not make a distinction between the modules he/she was evaluating. This is shown in the Min standard deviation column, which is zero for every smell. The zero standard deviation means that the developer has evaluated that all modules contain an equal amount of that smell. Still we must bear mind that this could mean that the modules under evaluation really are similar. This means that we cannot conclude that the smell evaluations are wrong, when an individual developer claims that several modules contain an equal amount of particular smell, if we cannot somehow compare it to the true amount of each smell.

Table 19 The Std. Dev. of smell means for different developers

Smell Name	Std deviations of smell means		
	Max	Min	Mean
Long Methods	2,83	0,00	1,16
Large Class	2,12	0,00	0,85
Long Parameter List	2,83	0,00	0,68
Data Clumps	1,15	0,00	0,40
Duplicate Code	1,53	0,00	0,76
Dead Code	2,12	0,00	0,95
Speculative Generality	1,10	0,00	0,57
Feature Envy	1,50	0,00	0,52
Inappropriate Intimacy	1,71	0,00	0,78
Message Chains	2,83	0,00	0,80
Middle Man	1,50	0,00	0,61
Lazy Class	1,73	0,00	0,51
Data Class	1,50	0,00	0,33
Incomplete Library Class	1,79	0,00	0,43
Primitive Obsession	1,21	0,00	0,25
Switch Statement	1,37	0,00	0,35
Temp Field	1,00	0,00	0,35
Refused Bequest	1,41	0,00	0,21
Alternative Classes with Different Interface	1,41	0,00	0,52
Parallel Inheritance Hierarchies	0,98	0,00	0,13
Divergent Change	1,73	0,00	0,61
Shotgun Surgery	2,12	0,00	0,73
Comments	1,64	0,00	0,30

6.4.3 Conclusions

If we compare the data from Table 16 and Table 19, we can see that module-smell means have generally higher standard deviation than smell-developer means. This means that if the smell evaluations are grouped based on modules, they are more widely distributed than if they are grouped based on individual developers. This indicates that developers' smell opinions on the same module are not uniform and thus not very reliable. One reason could be that it may not be possible to remember how much of each smell exists in individual modules. It could also be that the human mind cannot accurately remember such details on module level. Maybe the module level is still too high to obtain uniform evaluations. There could be different parts in different modules that contain different amount of smells, which will also cause bias. It could be that understanding the smells first and then searching for them could produce quite different results.

Part of the unexpected bias that has been discussed in this section could be due to the setup of the survey. In the survey, each individual smell was evaluated against all modules the developer had worked with. This could cause bias on individual opinions based on the quality of the other modules the developer had worked with. For example, developer A and

B could have worked with one common module X, but A could also have worked with module Y and B with module Z. So in this case developers A and B could evaluate module X quite differently based on the modules Y and Z.

I compared the rankings between five developers, who had evaluated two modules, and I found out that a certain individual gave the highest smell evaluations in both modules and there was also another individual who constantly gave the lowest smell evaluations. So there clearly was one developer that had the most positive opinion on the modules and other that had the most negative opinion, which could also explain the conflicting opinions in different modules.

The bias in this section also illustrates the problems with the Likert scale that was used. The problem is that people tend to evaluate differently on the same numeric scale. Even though developers' opinions are not uniform, this does not mean that the results in this section would be completely worthless. We just have to remember that they might be biased and therefore the smell valuation on individual modules and smells might not be precise. However, I believe that if we look beyond individual developer differences and look at the modules with several respondents, the results on smell evaluations could be more reliable. In the next section we will look at how some of the smells and their evaluations correlate with source code metrics.

6.5 Source Code Metrics & Smell Survey

In this section, source code measures will be compared with smell evaluations. The comparison is limited to only three smells and three modules. Only few smells can be studied effectively, since it was not possible for me to get good metrics on other smells due to the fact that I do not have the source code analyzing tools to gather those metrics. The smells that I try to correlate with source code metrics are Large Class, Long Parameter List and Duplicate Code. The reason for studying only source code metrics from three modules is that at the time this thesis was written those were the only modules, whose source code I had access to. In addition, analyzing more modules and smells would have greatly extended the time needed to finish this thesis. Luckily, two of those three modules were the ones that had most smell evaluations on the smell survey. The modules, whose source code measures I got are Gamma-C, Gamma-P, and Gamma-S. The sizes and approximate ages of the modules are presented in Table 20.

Table 20 Module Information

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
NLOC	83200	28654	16787
Age (years)	6-7	2-3	<1

I will now briefly introduce the tools used in getting the source code measures. A tool called *same*[9] was used to measure the number of duplicate code lines. Technically, *same* does

[9] http://sourceforge.net/projects/same

not measure the duplicate code lines, since it only investigates whether the lines are identical. This means that *same* will not recognize all possible duplicated lines. So actually the number of duplicated code lines will be higher than the one measured here. To gather measures for the Large Class and Long Parameter List smells I used a metric tool called *SDMetrics*[10]. *SDMetrics* calculates metrics from XMI files. XMI is standardized form of representing UML diagrams. To get the XMI files from Delphi source code I had to use two different tools, because the XMI output of these tools was not according to the standard. I got the necessary metrics by using *SDMetrics* to calculate metrics on both of the XMI output files. The tools used to generate the XMI-files were *ESS-Model*[11] and *Enterprise Architect*[12].

6.5.1 Large Class

The first question arising with the Large Class smell is, of course, what is a large class. Fowler & Beck (Fowler & Beck 2000) say that a large class can often be spotted by looking at the number of instance variables. The Large Class smell is also recognized as an anti-pattern known as the Blob, Winnebago, and the God Class. The book that describes this anti-pattern in detail (Brown, Malveau, McCormick, & Mowbray 1998) refers to "AntiPattern Session Notes" held by Michael Akroyd, who, according to Brown, Malveau, McCormick and Malveau, said that a class with more than 60 variables and operations often indicates the presence of the Blob. The number of operations in a class is also a metric whose extended version Chidamber and Kemerer (Chidamber & Kemerer 1994) introduced. The number of operations measure was also used in object-oriented design quality assessment by Bansiya and David (Bansiya & David 2002).

I also previously mentioned in Section 5.3.3 that class cohesion would be an optimal measure for the Large Class smell. However, unfortunately I do not have tools to collect such a measure. Therefore, I must limit the study and only measure the number of variables/attributes and operations/methods. The number of variables as a Large Class smell measure is suggested by Fowler and Beck (Fowler & Beck 2000), and it also sounds more reasonable than using number of operations. The complexity of an operation can fluctuate considerably, unlike variables, which in my opinion increase the class complexity more constantly. Since the number operations are commonly accepted as a measure of class complexity I will use them as well. For me a class sounds too large, if it has 20 or more variables. The only exceptions to this might be GUI classes that often have many instance variables. However, according to Fowler and Beck (Fowler & Beck 2000) this problem with GUI classes can be handled with a separate domain object.

Based on the discussion above and the tools available, I have created two categories to measure the Large Class smell. One is based on attributes and the other is based on operations. With variables and operations I have three limits for a Large Class. With variables the limits are 10, 20, and 40 or more variables, and with operations the limits are

[10] http://www.sdmetrics.com/

[11] http://www.essmodel.com/

[12] http://www.sparxsystems.com.au/

30, 50, and 100 or more. Much tighter thresholds are presented by Lorenz and Kidd (Lorenz & Kidd 1994), who suggest a threshold of 3 for instance variables in a model class, and 9 for a UI class. They also suggest that a model class should not have more than 20 operations and a UI class should have max 40 operations.

The metrics related to the class size in different modules can be seen in Table 21. The data shows that the Gamma-C module clearly has the largest classes, if we measure large classes by the number of variables. When comparing the number of large classes to the number of operations, we can see that Gamma-C and Gamma-S modules have the same number of large classes. Overall it looks like the Gamma-P module has the smallest number of large classes in both categories among these modules. The reason why the Gamma-C module has many classes with a big number of variables is due to the fact that many classes in that module are GUI classes. As discussed earlier, we can accept slightly larger GUI classes than regular classes. I might be willing to accept that GUI class can be three times larger in terms of variables than a model class as suggested by Lorenz and Kidd (Lorenz & Kidd 1994). Still we can see that the Gamma-C module has the largest classes, because 23% of its classes have 40 or more variables, while in the other two modules 7,3% and 9,7% of classes have 10 or more variables.

Table 21 Large Class source code measures

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
Number of classes	126	82	31
Percentage of classes with 30 or more operations	19,0	9,8	16,1
Percentage of classes with 50 or more operations	8,7	3,7	9,7
Percentage of classes with 100 or more operations	3,4	0,0	3,2
Percentage of classes with 10 or more variables	69,0	7,3	9,7
Percentage of classes with 20 or more variables	44,4	1,2	3,2
Percentage of classes with 40 or more variables	23,0	1,2	0,0

Smell means and medians in Table 22 show that modules Gamma-C and Gamma-P have been evaluated to contain an equal quantity of the Large Class smell. If we compare this to the data in Table 21, we can see that the smell mean or median does not correlate with the measured number of large classes. When I compared the five developers who had evaluated both Gamma-C and Gamma-P modules, I saw that only one developer had made distinctions between these modules. This developer had correctly evaluated that the Gamma-C module has more Large Class smell, although the difference in the Likert scale (1-7) was the smallest possible. I also studied how the developer who had given the sole evaluation on the Gamma-S module, had evaluated the other two modules. It appeared that this developer had evaluated both Gamma-C and Gamma-P modules with 4 on Likert scale, while the Gamma-S module had received only 2. This evaluation can be correct, if we compare Gamma-C and Gamma-S modules, but with Gamma-P and Gamma-S this modules evaluation is false if we compare it to my measures.

Table 22 Large Class smell means and medians

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
Number of evaluations	5	6	1
Mean of Large Class smell	5,20	5,17	2,00
Std Dev.	1,095	1,169	-
Median	5	5	2

My initial expectation was that the Large Class would nicely correlate with the measures, because I believe that Large Classes are quite easy to spot. It was unfortunate that I did not have class cohesion measures available that could have been more effectively used in detecting large classes. However, I feel that this data shows, how developers' evaluations on the Large Class smell seem to be incorrect, when compared to large class measures.

6.5.2 Long Parameter List

As mentioned earlier, the Long Parameter List smell means a case, where a method has too many parameters. Now it needs to be decided how many is actually too many. In the era of procedural programming, all data was generally passed as parameters. At that time the alternative to passing parameters was global data, which was much worse than long parameter lists. A widely recognized guide that was written for procedural programming recommends that parameters should be limited to seven (McConnell 1993). In object-oriented programming passing everything as parameters is no longer necessary. I personally have received education in and worked mostly on object-oriented programming, and my view is that generally the maximum amount of parameters should be three. In some extreme cases I could accept using as many as five parameters. So we have three opinions on what is a long parameter list. I shall call those opinions tolerance levels, which I will refer to as low, medium, and high. The maximum number of parameters in these categories is 3 for low, 5 for medium, and 7 for high.

Table 23 shows how the oldest and biggest module (Gamma-C) actually has the fewest long parameter lists compared to two younger modules. Gamma-P and Gamma-S modules have the same number of long parameter lists in low and high tolerance groups. In the medium tolerance group, the Gamma-P has more than twice as many long parameter lists. It is quite interesting that the oldest module seems to be clearly in the best shape, if we measure its internal quality by just looking at this single measure.

When Fowler & Beck (Fowler & Beck 2000) introduced the Long Parameter List smell, they had made the assumption that long parameter lists are made of primitives rather than objects. This source code material supports that assumption. From methods with over 3 parameters only 13,9% of parameters are classes, while 86,1% are primitives. The maximum number of primitive parameters is 16, whereas the maximum number of class parameters is 3. So it seems clear that the Long Parameter List smell is made up from primitives.

Table 23 Long Parameter List source code measures

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
Methods	2838	1077	464
Mean number of parameters	1,85	2,05	2,04
Percentage of Methods with 4 or more parameters (low tolerance)	9,9	14,9	15,1
Percentage of Methods with 6 or more parameters (medium tolerance)	1,3	7,1	3,4
Percentage of Methods with 8 or more parameters (high tolerance)	0,1	1,2	1,1

Table 24 shows the smell means and medians of the three modules under study. If we compare the two modules with more than one evaluation, we can see that developers have correctly evaluated that the Gamma-P has more Long Parameter List smell. We still have to bear in mind that the standard deviations for the smell means are quite high, so with different sampling the results might look different. The difference between the Long Parameter smell median of the Gamma-C and Gamma-P modules is greater than the smell mean values. The median values are the ones we wish to look at since the deviation is so large. I also studied the five developers who had evaluated both Gamma-C and modules and found out that only one of them had made a difference with the Long Parameter List in these two modules. This developer had evaluated correctly that the Gamma-P module has more of the Long Parameter Lists smell (with Likert scale numbers 5 and 3) while others had evaluated that this smell is equally present in both of the modules.

For the newest module, Gamma-S, I received only one smell mean or median. This single evaluation seems to be way off, if we compare it to the metric data and to the smell means and medians of the other two modules. It is even more interesting that the developer who had evaluated the Gamma-S module had also evaluated the Gamma-C module and given it a smell evaluation three for the Long Parameter List smell.

Table 24 Long Parameter List smell means and medians

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
Number of evaluations	5	6	1
Mean of Long Parameter List smell	3,40	4,00	1,00
Std Dev.	1,140	1,265	-
Median	3,00	4,50	1,00

So in the case of Long Parameter List smells developers correctly assumed that the Gamma-P module has more long parameter lists than the Gamma-C. On the other hand, individual developers' opinions were untrustworthy, since many developers were unable to make distinctions between two modules which according to measures contained different amount of smells. In addition, the comparison with the individual developers showed that developers' evaluations could simply be false.

6.5.3 Duplicate Code

As previously mentioned according to Fowler and Beck (Fowler & Beck 2000), the Duplicate Code smell *"is number one in the stink parade"*. Removing duplication makes programs easier to understand, maintain, and to develop further. When we measure the Duplicate Code smell, we must decide what the size of the duplicated fragments we wish to identify is. It is not very wise to remove duplicated code fragments that consist of only few lines of code, since the effort spent in removing them will outweigh the benefits. I am not aware of any recommendations on how many duplicate code lines are too much. The *same* tool reports on default duplicates with 10 lines of code or more. For me this sounds acceptable, but given that bigger duplicates are more interesting, I also defined groups with 15, 20, and 50 lines of code. The line of code in this context is again actually NLOC, which means that only the code lines are counted and empty lines and comment lines are ignored.

Table 25 shows the percentages of duplicate code lines measured in NLOC. In the table we can see that the Gamma-C module has clearly the most duplicate code. The Gamma-S has almost 30% of duplicate code if we define the duplicate code chunks to be 10 NLOC or more. If we only measure larger duplicate code chunks, the duplicate code percentage of Gamma-S module drops very much. From the source code I found out that the Gamma-S has many methods that terminate in similar way, i.e., they check out of a critical section, do some exception handling, and then report to the log system that the method has exited. So this kind of duplication will only cause problems if the exiting sequence has to be changed.

Table 25 Duplicate Code source code measures

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
Percentage of duplicate code lines with 10 NLOC or more	13,6	11,0	28,2
Percentage of duplicate code lines with 15 NLOC or more	8,8	4,3	3,5
Percentage of duplicate code lines with 20 NLOC or more	5,6	1,5	0,9
Percentage of duplicate code lines with 50 NLOC or more	1,1	0,0	0,4

In Table 26 we can see that the Gamma-P module is evaluated to contain most Duplicate Code smell. Although the difference to the Gamma-C is not very big, we can clearly see that developers have mistaken, because the Gamma-C has much more duplicate code according to my measurement. When I looked at the five developers that had evaluated both Gamma-C and Gamma-P modules I saw that two developers had evaluated that the Gamma-P contains more duplicate code, one developer had determined that the Gamma-C has more of this smell, and two developers had decided that the modules contain the same amount of Duplicate Code smell. To explain why developers felt that the Gamma-P contains more Duplicate Code than it actually does, I also tried to look at duplicates between modules. I found out that there seems to be more in-module duplication than inter-module duplication. By this I mean that percentages of duplicated code lines did not exceed the values of the Gamma-C, if I measured the duplicated code line percentages from all the modules simultaneously.

One reason for the differences between the smell mean and the source code measurements was revealed to me during the discussion with the case company. I found out that the Gamma-P module actually has quite a few lines, where copy-paste coding has been applied, but after each paste operation the code has been slightly modified. The *same* tool is unable to detect such form of duplication. This information indicates that the smell evaluations are not as false as they would appear according to the measurements[13].

Again, I also studied the answers of the respondent who had evaluated the newest Gamma-S and the other two modules as well. This developer had evaluated that the Gamma-P has more duplicate code smell than the Gamma-S, whereas in reality they have about same amount of duplicate code. This result with this single developer is very similar as with the previously compared smells.

Table 26 Duplicate Code smell evaluations

Property	**Module**		
	Gamma-C	Gamma-P	Gamma-S
Number of evaluations	5	6	1
Mean of Duplicate Code smell	3,60	4,00	1,00
Std Dev.	1,140	1,265	-
Median	3,00	3,50	1,00

So based on this data it seems that the developers have mistaken in their evaluations, when it comes to duplicate code. However, the copy-paste-modify programming caused some bias to this result. Nevertheless, based on these results it is obvious that duplicate code smell could benefit very much on automatic detection.

6.5.4 Conclusions

In sections 6.5.1, 6.5.2, and 6.5.3 I have compared the developers' smell evaluations and source code measures. This data showed that the developers' opinions on bad code smells do not seem to correlate with the used source code measures. With the Long Parameter List smell the developers' smell evaluations had the best correlation with the measures, while in the Duplicate Code smell the developers' evaluations did not correlate with the used measures at all. When I compared the evaluations of the only developer, who had worked with the newest module (Gamma-S) and the other two modules as well, I saw how the developer evaluated that the newest module was considerably less affected by the bad smells than the other modules. The measures revealed, however, that the newest module did contain as much of bad smell as the other two modules.

[13] It is a completely different story to find out whether code that has been developed with a copy-paste-modify method is actually duplicate code, and how much modification is needed before the code can no longer be considered duplicate code. However, it is certain that removing completely identical code chunks is easier than removing code that has been slightly modified.

When looking at the data in Table 25 and in Table 21 it is evident that Fowler & Beck's claim that Duplicate Code and Large Class smells go hand in hand might actually be true. We can see how the Gamma-C module has the highest number of large classes and the most duplication. On the other hand, the Gamma-P module has the fewest duplication and fewest large classes.

The results in this section showed that developers' opinions do not correlate well with used source code measures. Therefore I must say that the smell evaluations are not really reliable. This result is similar to the one presented in Section 6.4. This result also indicates that there is a need for automatic smell detection and decreases the reliability of the smell evaluations.

6.6 Summary

This section has presented the setting and the results of my survey on bad code smells that was targeted to developers of the case company BeachPark. The survey was introduced in Section 6.1 and the different smells contained by different modules were also discussed. By studying the smell evaluation differences between modules, I found out that some smells have significant differences between modules while other smells have not. The smells with significant differences could be thought as archetype smells, as they have clearly different strength in different modules. In Section 6.2 the correlations between smells were introduced. There we saw how my theoretical taxonomy from Section 5.2 gained more support, as many of the taxonomy's correlations within group were strong and significant. In Section 6.4 I studied how uniform the smell evaluations of a single module from several developers were. There we saw that the smell evaluations are not very since standard deviations of the smell evaluations were greater per module than they were per developer. In Section 6.3 I studied how the different background variables affected the smell evaluations. It appeared that lead developers see more structural smells than regular developers, and respondents with a better knowledge of the system tended to see more structural issues as well as smells that are difficult to spot. In Section 6.3 we saw how the two developers with most work experience from the case company saw the code considerable less smelly. This led us to discuss that the reason for this could be that the two developers have originally written the code and therefore are not willing to see as many smells as the rest. Finally, in Section 6.5 the smell evaluations and the results of the source code measurement were compared for the particular smells. This showed us that the smells evaluations and the source code measures do not seem to correlate.

7 Discussion

This is the final chapter of the study. This chapter recaptures the plans and actual work done to answer the research questions, summarizes the answers to the research questions, discusses the reliability and generalizability of findings and finally provides ideas for further research.

7.1 Answering the Research Questions

The research questions were introduced in Section 1.3. In this section, there will be a summary of the answers to the research questions and the plans and actions that provided the results will be discussed.

7.1.1 Answering Research Question 1

The first research question was: **How effectively can different code smells by Fowler & Beck be measured by tools?** To answer this research question I planned to study the different source code metrics and the bad code smells. This action was carried out as planned. While answering the research question I tried to find metrics for each bad code smell. This research question was studied in detail and answered in section 5.3. Based on the results it appears that roughly speaking little more than half of the smells can effectively be measured with tools.

7.1.2 Answering Research Question 2

The second research question was: **How can the smells be made more understandable?** My plan was to introduce a taxonomy for the bad code smells to make them more understandable. I created the taxonomy as planned. The taxonomy maps the 22 code smells to 7 higher-level categories in Section 5.2. I believe that this taxonomy makes the smells easier to understand, because it is easier to get an overview of a taxonomy with 7 categories than of a straight flat list of 22 code smells. To demonstrate the usability of this taxonomy I also used it to group the correlating smells from the developers' smell evaluations in Section 6.2. This taxonomy matched the evaluations very well, since 23,53% of the within-group correlations were strong, whereas only 0,04 percent of the between-group correlations were strong.

7.1.3 Answering Research Questions 3 and 3a

The third research question was: **Do software developers have a uniform opinion on the "smelliness" of the source code?** To answer this research question I planned and conducted a web-based survey where the developers of the case company were asked to evaluate the bad code smells in the modules they had worked with. We saw in Section 6.4 that the developers' evaluations on the smells of individual modules can fluctuate considerably. So it seems that the personality of an individual developer can affect the smell evaluation even more than the module in question. The extension to third research question was: **How does the developer's experience and capability affect the smell evaluations?** To answer this research question I used the data from the web survey. In Section 6.3 I studied how the experience, role, and knowledge of the software module affected the smell evaluations. I think that the role and knowledge are attributes that correspond to the developers' capability. Developers with higher capability tend to think that there are more structural and higher-level smells considering, e.g., the class hierarchy. We also saw how the two most experienced developers seemed to think that the code is much less smelly. This is probably because the two developers have written so much of the code themselves and thus considered it less smelly than the other nine developers.

7.1.4 Answering Research Questions 4

The final research question was: **Do the developers' evaluations on different code smells correlate with source code metrics?** To be able to answer this research question, I compared the smell evaluations from the web survey with measured source code metrics data. This action was carried out as planned, although not all source code of the module was measured. All source code could not be measured, because I did not have access to it. Regardless of that, I got enough source code to be able to answer the research question. The answer to this research question was studied in Section 6.5. As I did not have source code for all modules and only had measures for few smells, the answers to this research question are only initial. Nevertheless, based on the data I had, it seems that the developers' smell evaluations do not correlate with source code metrics for the smells. One of the consequences of this result is that automatic smell detection or maintainability measurement is useful for providing ideas on which parts of software need refactoring and for providing information on the general code quality of different software modules. On the other hand, this result could also indicate that the bad code smell based on human evaluations might not be very applicable after all.

7.2 Reliability of The Results

In the previous section I summarized the answers to the research questions. In this section I will discuss the reliability of the results. This section will go through the research questions and evaluate the reliability of the answers.

7.2.1 Research Question 1

When looking for an answer to the first research question I tried to evaluate how different smells could be measured and how well they could be measured. I have good confidence in that the measures presented for the smells are quite valid. Nevertheless, the measurability of the smells was only based on my personal intuition on what I had learned about the code smells and source code metrics. Because this work did not validate the measurability in any way, I think the presented measurability evaluation of the smells only provides suggestions on how easy it would be to measure a particular smell.

7.2.2 Research Question 2

The answer to the second research question provided a taxonomy for the bad code smells and it was applied to the smell survey correlations. I think that the taxonomy for the smells is useful, because it makes the smells and their relationships easier to understand. The taxonomy proved useful in analyzing the smell correlation relationships. The taxonomy is by no means complete, and it should be changed if and when empirical work shows it needs adaptation. This taxonomy is to my knowledge the first bad code smell taxonomy, so it is merely nothing more than a starting point. The empirical validation of the reliability of the taxonomy is, however, a bit questionable due to the nature of the survey. The reliability of the survey is discussed in the following section.

7.2.3 Research Question 3 and 3a

The third research question studied how uniform the developers' smell evaluations were. The results showed that the smell evaluations were not very uniform in all cases, which was surprising. The basic problem with the reliability was that the survey was a so-called "cold turkey" survey. After conducting the survey I learned that the proper way to do a survey is to talk with the respondent in order to make sure he understand the questions correctly and is actually able to answer the questions. Because the survey in this study was a so-called cold turkey survey, I cannot know whether the developers answered the question based on their recollection of the smells or whether they actually looked at the source code. However, as people are generally lazy, it is likely that most of the developers answered the survey based on their recollection.

Another point that reduces the reliability of the results related to the third research question is that the software modules under evaluation were quite large. So it is possible that the respondents have worked with different parts of the software module, and this would explain the fluctuation in the smell evaluations. I feel that understanding the smells was not a big factor for the respondents, because the answer percentage was quite high in almost all the smells. A further problem that makes this result a bit unreliable is the possibility that people might evaluate something differently on the Likert scale, e.g., because of their background (positive versus negative orientation). This problem is actually more a problem of the Likert scale itself, but nevertheless it weakens the reliability of the results.

When answering the research question 3a I studied the effect of the background variables on the smell evaluations. The results in this case were more or less what was expected. However, I must admit that this single survey is too small a study for conclusive results. These results illustrate a single case, whose results can be compared with future studies. These results also suffer from the reliability problems of the cold turkey survey.

7.2.4 Research Question 4

The last research question studied the relationship of the smell evaluations and source code metrics. The results showed that the metrics did not correlate with the smell evaluations. I have high confidence in the source code measures used, since the smells that I tried to measure were the ones that I found quite measurable in Section 6.2. With each source code measure I used several threshold values, as there can be no single number for all contexts that tells when a particular value is too high or low. Therefore I think the measurement of the smells is reliable. However, the issues that make the result unreliable are the problems with the cold turkey survey that were already discussed in Section 7.2.3. A further issue that makes this result a bit unreliable is the fact that developers evaluated modules that were quite big. Therefore I can only conclude that with this slightly unreliable survey and with modules of this size, the smell evaluations do not correlate with source code metrics.

7.3 Generalizability of the Results

The results from the research questions 1 and 2 should be easily generalizable, because they were not in any way specific to the case company. Therefore I believe that the result should be effective in other settings as well. The generalizability and reliability of these results can be improved with empirical studies of the issue.

The generalization of the results from research questions 3, 3a and 4 is more difficult. I believe that with similar settings, these kinds of results could be received from other companies as well. However, since the results in the research questions 3, 3a, and 4 suffer from the reliability problems of the cold turkey survey, it is more important to improve their reliability before further generalization is performed.

7.4 Further Work

This section lists some possible further topics for study that seem interesting based on the results of this work.

A further topic for study would be to look at how developers would evaluate the smells in the same piece of code. The code sample would have to be quite small so that the developers could go through it in a reasonable time. With this kind of study it would be possible to present more conclusive results on how uniform the developers' smell evaluations are. This kind of study could also be combined with the automatic smell measurement and this would bring more information on the relationships of code smells and source code measures.

There are also other motivating topics of research. For example, one could study how the ownership affects the smell evaluations. In this study, I did not specifically ask how much of the source code for each module the developer had written. So I was only able to speculate on this based on the data of the developer work experience in the company. So it would be interesting to see how the smell evaluations would differ between the source code author and an outsider.

It would also be interesting to see how some of the development methods would affect the source code smelliness or maintainability, e.g., if automatic unit testing, pair-programming, or code reviews affect the code quality. Personally I believe that if a programmer must write a unit test for his/her code he/she will write code that is more maintainable, as it is likely that this kind of code is also easier to test. Validating such assumptions seems to provide a promising research topic.

As an industrial study it would be appealing to look at ways to prevent or clean up low quality code, e.g., do simple listings in the coffee room pointing out the duplicate code segments reduce the amount of duplicate code, or if a more institutionalized process is needed. This would help the industry in choosing the right ways to make developers produce more maintainable code.

All these research topics could also help improving the smell taxonomy.

One final research topic would be to try to find the worst bad code smells or antipatterns by interviewing developers. This would help software engineering community to concentrate on the worst issues of unmaintainable software. Maybe this type of study could be extended to build polynomials, such as in Sections 3.3.3 and 3.3.4, to measure maintainability with bad code smells.

References

Adolph, W. S. 1996, "Cash cow in the tar pit: Reengineering a legacy system", *IEEE Software*, vol. 13, no. 3, pp. 41-47.

AFOTEC. Software Maintainability --- Evaluation Guide. [3]. 1989. Kirkland Air Force Base, New Mexico, USA, HQ Air Force Operational Test and Evaluation Center. AFOTEC Pamphlet 800-2 (updated).

Arnold, R. S. 1989, "Software Restructuring", *Proceedings of the Ieee*, vol. 77, no. 4, pp. 607-617.

Ash, D., Alderete, J., Yao, L., Oman, P. W., & Lowther, B. 1994, "Using Software Maintainability Model to Track Code Health", in *Proceedings of International Conference on Software Maintenance*, IEEE, pp. 154-160.

Balazinska, M., Merlo, E., Dagenais, M., Lague, B., & Kontogiannis, K. 2000, "Advanced clone-analysis to support object-oriented system refactoring", in *Proceedings of Seventh Working Conference on Reverse Engineering*, IEEE, pp. 98-107.

Bandi, R. K., Vaishnavi, V. K., & Turk, D. E. 2003, "Predicting maintenance performance using object-oriented design complexity metrics", *IEEE Transactions on Software Engineering*, vol. 29, no. 1, pp. 77-87.

Bansiya, J. & David, C. G. 2002, "A Hierarchical Model for Object-Oriented Design Quality", *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 4-17.

Basili, V. R., Briand, L. C., & Melo, W. L. 1996, "A Validation of Object Oriented Design Metric as Quality Indicators", *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751-761.

Beck, K. 2000, *Extreme Programming Explained* Addison-Wesley, Canada.

Bennett, K. H. & Rajlich, V. T. 2000, "Software maintenance and evolution: a roadmap", in *Proceedings of the conference on The future of Software engineering*, ACM Press, New York, NY, USA, pp. 73-87.

Bianchi, A., Caivano, D., Marengo, V., & Visaggio, G. 2003, "Iterative reengineering of legacy systems", *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 225-241.

Bray, O. & Hess, M. M. 1995, "Reengineering a Confuguration Management System", *IEEE Software*, vol. 12, no. 1, pp. 55-63.

Briand, L. C., Bunse, C., Daly, J. W., & Differding, C. 1997, "An Experimental Comparison of the Maintainability of Object-Oriented and Structured Design Documents", *Empirical Software Engineering*, vol. 2, no. 3, pp. 291-312.

Briand, L. C., Daly, J. W., & Wüst, J. K. 1997, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems", in *Proceedings of the Fourth International Software Metrics Symposium*, pp. 43-53.

Briand, L. C., Daly, J. W., & Wüst, J. K. 1999, "A Unified Framework for Coupling Measurement in Object-Oriented Systems", *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91-121.

Briand, L. C., Devanbu, P., & Melo, W. 1997, "An Investigation into Coupling Measures for C++", in *Proceedings of the 1997 International Conference on Software Engineering*, IEEE, pp. 412-421.

Briand, L. C., Wüst, J., Ikonomovski, S. V., & Lounis, H. 1999, "Investigating Quality Factors in Object-oriented Designs: an Industrial Case Study", in *Proceedings of the 1999 International Conference on Software Engineering*, IEEE, pp. 345-354.

Brown, W. J., Malveau, R. C., McCormick, H. W., & Mowbray, T. J. 1998, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* Wiley, New York.

Card, D. N. & Glass, R. L. 1990, *Measuring Software Design Quality* Prentice-Hall, Eaglewood Cliffs, New Jersey, USA.

Chan, T. Z., Chung, S. L., & Ho, T. H. 1996, "An economic model to estimate software rewriting and replacement times", *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 580-598.

Chapin, N. 2000, "Do we know what preventive maintenance is?", in *Proceedings of International Conference on Software Maintenance*, IEEE, San Jose, CA, USA, pp. 15-17.

Chidamber, S. R. & Kemerer, C. F. Phoenix, Arizona, USA, "Towards a Metrics Suite for Object-oriented Design", in *Conference proceedings on Object-oriented programming systems, languages, and applications*, ACM Press, New York, NY, USA, pp. 197-211.

Chidamber, S. R. & Kemerer, C. F. 1994, "A Metric Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493.

Coleman, D., Ash, D., Lowther, B., & Oman, P. W. 1994, "Using Metrics to Evaluate Software System Maintainability", *Computer*, vol. 27, no. 8, pp. 44-49.

Coleman, D., Lowther, B., & Oman, P. W. 1995, "The Application of Software Maintainability Models in Industrial Software Systems", *Journal of Systems and Software*, vol. 29, no. 1, pp. 3-16.

Counsell, S., Mendes, E., & Swift, S. 2002, "Comprehension of object-oriented software cohesion: the empirical quagmire", in *Proceedings of the 10th International Workshop on Program Comprehension*, IEEE, pp. 33-42.

Cusumano, M. A. & Selby, R. W. 1995, *Microsoft Secrets* The Free Press, USA.

Cusumano, M. A. & Yoffie, D. B. 1998, *Competing on Internet Time* The Free Press, New York, USA.

Cusumano, M. A. & Yoffie, D. B. 1999, "Software development on Internet time", *Computer*, vol. 32, no. 10, pp. 60-69.

Daly, J. W., Miller, J., Brooks, J., Roper, M., & Wood, M. 1995, *Issues on the object-oriented paradigm: A questionnaire survey.*

DeMarco, T. 1982, *Controlling Software Projects* Prentice Hall.

Demeyer, S., Ducasse, S., & Nierstrasz, O. "Finding refactorings via change metrics", in *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, ACM Press.

Demeyer, S., Ducasse, S., & Nierstrasz, O. 2003, *Object-Oriented Re-engineering Patterns* Morgan Kaufman Publishers.

Eich, B. Mozilla Development Roadmap. http://www.mozilla.org/roadmap.html . 27-11-2002. 29-11-2002.

El Emam, K., Melo, W., & Machado, J. C. 2001, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics", *The Journal of Systems and Software*, vol. 56, no. 1, pp. 63-75.

Fenton, N. E. & Ohlsson, N. 2000, "Quantitave Analysis of Faults and Failures in a Complex Software System", *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797-814.

Fenton, N. E. & Pfleeger, S. L. 1996, *Software Metrics*, Second edn, Thomson Publishing Inc., USA.

Fowler, M. 2000, *Refactoring: Improving the Design of Existing Code* Addison-Wesley, Canada.

Fowler, M. & Beck, K. 2000, "Bad Smells in Code," in *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, pp. 75-88.

Glass, R. L. & Noiseux, R. A. 1981, *Software Maintenance Guidebook* Prentice-Hall, Englewood Cliffs.

Grady, R. B. 1994, "Successfully Applying Software Metrics", *Computer*, vol. 27, no. 9, pp. 18-25.

Grady, R. B. & Caswell, D. L. 1987, *Software Metrics: Establishing a Company-wide Program* Prentice Hall, Englewood Cliffs, NJ.

Graves, T. L., Karr, A. F., Marron, J. S., & Siy, H. 2000, "Predicting Fault Incidence Using Software Change History", *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661.

Grigg, J. Re: [refactoring] Size of Java Classes? Refactoring mailing list. 6-8-2002.

Haikala, I. & Märijärvi, J. 1998, *Ohjelmistotuotanto* Gummerus Kirjapaino Oy, Jyväskylä.

Halstead, M. H. 1977, *Elements of software science* Elsevier, New York.

Hamer, P. G. & Frewin, G. D. 1982, "M.H. Halstead's Software Science - a critical examination", in *Proceedings of the 6th international conference on Software engineering*, IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 197-206.

Henderson-Sellers, B. 1996, *Object-Oriented Metrics*, 2nd edn, Prentice Hall, Upper Saddle River, New Jersey, USA.

Henry, S. M., Humphrey, M., & Lewis, J. 1990, "Evaluation of the Maitainability of Object-Oriented Software", in *Proceedings of IEEE Region 10 Conference on Computer and Communication Systems.*

IEEE 1990, *IEEE Standard Glossary of Software Engineering Terminology* The Institute of Electrical and Electronics Engineers, Inc., New York.

IEEE 1998, *IEEE Standard for Software Maintenance* The Institute of Electrical and Electronics Engineers, Inc., New York.

Kafura, D. G. & Reddy, G. R. 1987, "The Use of Software Complexity Metrics in Software Maintenance", *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 335-343.

Kataoka, Y., Ernst, M. D., Griswold, W. G., & Notkin, D. 2001, "Automated support for program refactoring using invariants", in *Proceedings of International Conference on Software Maintenance*, IEEE, pp. 736-743.

Kataoka, Y., Imai, T., Andou, H., & Fukaya, T. 2002, "A Quantative Evaluation of Maintainability Enhancment by Refactoring", in *Proceedings of the International Conference on Software Maintenance*, IEEE, pp. 576-585.

Kruchten, P. 2000, *The Rational Unified Process An Introduction, Second Edition* Addison Wesley Longman, Inc., Canada.

Lehman, M. M. 1980, "On Understanding Laws, Evolution, and Conservation in the Large-Program Life Cycle", *The Journal of Systems and Software*, vol. 1, pp. 213-221.

Li, W. & Henry, S. M. 1993a, "Maintenance metrics for the object oriented paradigm", in *Proceedings of the First International Software Metrics Symposium*, IEEE, pp. 52-60.

Li, W. & Henry, S. M. 1993b, "Object-Oriented Metrics that Predict Maintainability", *Journal of Systems and Software*, vol. 23, no. 2, pp. 111-122.

Lorenz, M. & Kidd, J. 1994, *Object-Oriented Software Metrics* Prentice Hall, Upper Saddle River, New Jersey, USA.

Ma, C.-S., Chang, C. K., & Cleland-Huang, J. 2001, "Measuring the intensity of object coupling in C++ programs", in *Proceedings of Annual International Computer Software and Applications Conference*, IEEE, pp. 538-543.

Mancl, D. 2001, "Refactoring for Software Migration", *IEEE Communications Magazine*, vol. 39, no. 10, pp. 88-93.

Maruyama, K. & Shima, K. 1999, "Automatic method refactoring using weighted dependence graphs", in *Proceedings of the 1999 International Conference on Software Engineering*, IEEE, pp. 236-245.

McCabe, T. J. 1976, "A Complexity Measure", *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308-320.

McConnell, S. 1993, *Code Complete* Microsoft Press, Redmond, Washington, USA.

Misic, V. B. 2001, "Cohesion is structural, coherence is functional: different views, different measures", in *Proceedings of the Seventh International Software Metrics Symposium*, pp. 135-144.

Muthanna, S., Stacey, B., Kontogiannis, K., & Ponnambalam, K. 2000, " A maintainability model for industrial software systems using design level metrics", in *Proceedings of Seventh Working Conference on Reverse Engineering*, pp. 248-256.

Ohlsson, N. & Alberg, H. 1996, "Predicting Error-Prone Software Modules in Telephone Switches", *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886-894.

Opdyke, W. 1992, *Refactoring Object-Oriented Frameworks,* Doctoral Dissertation, Graduate College of the University of Illinois at Urbana-Champaign.

Pigoski, T. M. 1996, *Practical Software Maintenance* John Wiley & Sons Inc..

Rajlich, V. T. & Bennett, K. H. 2000, "A staged model for the software life cycle", *Computer* pp. 66-71.

Rombach, D. H. 1987, "Controlled Experiment on the Impact of Software Structure on Maintainability", *IEEE Transactions on Software Engineering*, vol. 13, no. 3, pp. 344-354.

Royce, W. W. 1970, "Managing the Development of Large Software Systems", in *Proceedings of Wescon*, pp. 1-9.

Schach, S. R. 2002, *Object-Oriented and Classical Software Engineering* McGraw-Hill, New York.

Simon, F., Steinbruckner, F., & Lewerentz, C. 2001, "Metrics based refactoring", in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, IEEE, pp. 30-38.

Sneed, H. M. 1995, "Planning the Reengineering of Legacy Systems", *IEEE Software*, vol. 12, no. 1, pp. 24-34.

Sommerville, I. 1996, *Software Engineering* Addison-Wesley.

Spolsky, J. Painless Software Schedules.
http://www.joelonsoftware.com/articles/fog0000000245.html . 29-3-2000a. 28-11-2002a.

Spolsky, J. Things You Should Never Do, Part I.
http://www.joelonsoftware.com/articles/fog0000000069.html . 6-4-2000b. 28-11-2002b.

Stevens, W., Myers, G., & Constantine, L. 1974, "Structured Design", *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139.

Swanson, E. B. 1976, "The Dimensions of Maintenance", in *Proceedings of the 2nd International Conference on Software Engineering*, IEEE Computer Society Pres, Los Alamitos, CA, USA, pp. 492-497.

Tamai, T. & Torimitsu, Y. 1992, "Software Lifetime and its Evolution Process over Generations", in *Proceedings of Conference on Software Maintenance*, IEEE, pp. 63-69.

Walsh, T. J. "A Software Reliability Study Using a Complexity Metrics", in *Proceedings of the AFIPS National Computer Conference*, New York, pp. 761-768.

Weyuker, E. J. 1988, "Evaluating software complexity measures", *IEEE Transactions on Software Engineering*, vol. 14, no. 9, pp. 1357-1365.

Yu, P., Systä, T., & Müller, H. 2002, "Predicting fault-proneness using OO metrics. An industrial case study", in *Proceedings of Sixth European Conference on Software Maintenance and Reengineering*, IEEE, pp. 99-107.

Zhuo, F., Lowther, B., Oman, P. W., & Hagemeister, J. 1993, "Constructing and testing software maintainability assessment models", in *Proceedings of Software Metrics Symposium*, pp. 61-70.

Appendix A

This appendix contains the questions used in the web survey to collect the software developers' evaluations on bad code smells. Below is the part of the module that was used to gather the background information.

Taustatiedot

Ikä:

Sijainti:

Asema:

Kuinka kauan olet työskennellyt ohjelmoijana BeachPark:lla (vuotta, kuukautta):

Kuinka kauan olet kaiken kaikkiaan työskennellyt ohjelmoijana (vuotta, kuukautta):

Koulutustausta (valitse yksi): Jatkotutkinto (esim. Lisensiaatti, Tohtori); Korkeakoulututkinto (esim. Maisteri, Diplomi-insinööri); Opisto / Ammattikorkeakoulututkinto (esim. Insinööri, Tradenomi); Opiskelija tiedekorkeakoulu (esim. Yliopisto, TKK); Opiskelija ammattikorkeakoulu; Muu, Mikä?

Moduulit

Seuraavassa lomakkeessa kysellään koodista löytyvistä rakenteista (koodihajuista) per moduuli. Lomake generoidaan valitsemiesi moduuleiden perusteella, joten vastaat kysymyksiin ainoastaan niiltä osin.

Ole ystävällinen ja valitse listasta ne moduulit joiden kanssa olet pää-asiassa työskennellyt. Suositeltavaa on, että rajoitat valitsemasi moduulit 3:een. Lead developerit (ja halutessaan toki muutkin) voivat vastata useampaan kuin 3:een moduuliin

The name of the modules are withheld

The second part of the questionnaire was generated based on the names of the selected software modules. The respondents answered to each of the following questions for all the modules they had chosen in the previous phase.

1. Kuinka hyvin arvelet tuntevasti listaamasi moduulit

Tunnen ohjelmiston osan
Asteikko 1-7 (1 erittäin huonosti ... 7 erittäin hyvin)

2. Minkä verran ohjelmistojen eri osista löytyy seuraavia rakenteita?

Ole ystävällinen ja vastaa vain sen perusteella, mitä olet itse nähnyt ja kokenut
Asteikko 1-7 (1 ei lainkaan ... 7 paljon; EOS; En ymmärrä)

1. Liian pitkiä funktioita

2. Liian isoja luokkia (liikaa instanssimuuttujia tai toiminnallisuutta)

3. Funktioita jotka ottavat liikaa parametreja

4. Usein yhdessä esiintyviä data-alkioita (Esim. 3 int tyyppistä muuttujaa joista ei olisi mitään iloa yksinään)

5. Duplikaatti koodia (Eli koodia, jota on esim. kehitetty copy-paste menetelmällä)

6. Tarpeetonta (kuollutta) koodia (Eli koodia, jota on joskus käytetty ja "säästetty" tulevaisuuden varalle)

7. Spekuloivaa koodia (Eli koodia, jota rakennettu ottaen huomioon erillaisia skenaariota tulevaisuudessa)

8. Funktioita jotka ovat liian kiinnostuneita toisista luokista ja niiden datasta. (Eli käyttävät esim. useita get-metodeja)

9. Kahden tai useamman luokan kokonaisuuksia, jotka ovat tiukasti naimisissa keskenään s.e. niiden erottaminen toisistaan muodostuisi vaikeaksi.

10. Pitkiä viestiketjuja (Eli tilanteita jossa tiettyä dataa pyydetään seuraavalta oliolta, joka edelleen delegoi kutsua eteenpäin)

11. Luokkia, jotka pääasiassa delegoivat kutsuja eteenpäin eivätkä juuri sisällä omaa toiminnallisuutta.

12. Luokkia, jotka sisältävät erittäin vähän toiminnallisuutta tai dataa s.e. ne voitaisiin pienellä vaivalla poistaa.

13. Luokkia, jotka sisältävät pääasiassa dataa eivätkä juurikaan toiminnallisuutta.

14. Kohtia, joissa ohjelmoija on joutunut paikkamaan esim. delphin luokkakirjastossa olevia ongelmia omalla koodilla.

15. Tilanteita, joissa valmiiden primitiivien asemasta käytetään pieniä luokkia. (Esim. puhelinnumerolle on tehty oma luokka integerin käyttämisen sijaan)

16. Tilanteita, joissa pyritään ajon aikana selvittämään olion todellinen tyyppi. (Esim. luetaan luokan "ID" kentän arvo tai käytetään Delphin is operaatiota)

17. Luokkia, jotka sisältävät temp-muuttujia. (Eli siis muuttujia joita luokka tarvitsee vain esim. tietyssä tilanteessa tai tilassa)

18. Luokkia, jotka eivät tue kaikkia perimiään metodeja. (Esim. tilanne joissa voit kutsua kantaluokalta A saatua metodia, mutta sen suorituksen toimivuudesta/oikeellisuudesta perityssä luokassa B ei ole takeita.)

19. Rakenteita, joissa voidaan käyttää yhtä luokkaa useammasta vaihtoehdosta, mutta vaihtoehtoisia luokkia käytetään eri rajapinnan läpi. (Esim. Voit piirtää ruudulle neliön tai pallon, mutta piirtääksesi neliön kutsut metodia draw_rectanle() ja piirtääksesi pallon kutsut metodia draw_ball())

20. Rinnakkaisia luokkahierarkioita. Eli tilanteita, joissa periessäsi yhden uuden luokan sinun pitää periä toinen rinnakkainen luokka toisesta rinnakkaisesta hierarkiasta.

21. Yksittäisiä luokkia, jota pitää muokata useiden erilaisten muutosten yhteydessä. (Esim. Muuttaessasi sekä tietokannan tyyppiä, että autentikointi mekanisimia joudut muokkaamaan samaa luokka)

22. Tilanteita, joissa useampiin luokkiin pitäisi tehdä pieniä muokkauksia yhden muutoksen yhteydessä. (Esim. vaihtaessasi tietokanta tyyppiä pitää muutoksia tehdä useaan luokkaan)

23. Tilanteita, joissa joku kohta koodista on vain kommentoitu sen sijaan, että olisi ohjelmoitu se selkeämmin. (Esim. Kommentti: "Fix this later")

Appendix B

This appendix contans all responses, smell means, medians, and standard deviations

Smell	N	Mean	Median	Std. Dev.	Range
Long Method	36	4,11	4	1,563	4
Large Class	36	3,72	4	1,466	6
Long Parameter List	37	2,95	2	1,332	5
Data Clumps	24	3,17	3	0,963	4
Duplicate code	37	3,62	4	1,187	5
Dead Code	35	3,03	3	1,339	5
Speculative Generality	35	2,89	3	1,157	4
Feature Envy	35	3,00	3	1,085	4
Inappropriate Intimacy	33	3,73	4	1,547	5
Message Chain	36	4,08	4	1,422	6
Middle Man	36	2,78	3	1,149	5
Lazy Class	36	2,19	2	1,167	4
Data Class	33	2,52	2	1,202	4
Incomplete Library class	34	3,00	2,5	1,723	5
Primitive Obsession *	36	6,03	6	1,134	4
Switch Statements	35	3,29	3	1,827	6
Temporary Field	36	2,56	2	1,319	6
Refused Bequest	31	1,77	2	0,717	3
Alternative Classes with different interfaces	30	2,33	2	1,040	4
Parallel Inheritance Hierarchies	33	1,85	2	0,834	3
Divergent Change	33	3,03	3	1,331	4
Shotgun Surgery	36	3,39	3	1,103	3
Comments	34	2,91	3	1,190	5

* Measured on a reversed scale. The reason why the Primitive Obsession clearly has the highest smell mean is that it was measured on a reversed scale, i.e., the smaller the number was, the more smell there was.

Appendix C

This table contains the correlations between the smell evaluations. I have numbered the cells to be able to fit the table on the page:

Long Method = 1, Large Class = 2, Long Parameter List = 3, Data Clumps = 4, Duplicate Code = 5, Dead Code = 6 Speculative Generality = 7, Feature Envy = 8, Inappropriate Intimacy = 9, Message Chains = 10, Middle Man = 11, Lazy Class = 12, Data Class = 13, Incomplete Library Class = 14, Primitive Obsession = 15, Switch Statement = 16, Temp Field = 17, Refused Bequest = 18, Alternative Classes with Different Interfaces = 19, Parallel Inheritance Hierarchies = 20, Divergent Change = 21, Shotgun Surgery = 22, Comments = 23

CC means Correlation Coefficients and Sig. means 2-tailed significance.

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
1	CC	1,000	0,751	0,387	0,246	0,317	-0,039	0,066	0,455	0,277	0,488	0,245	-0,119	-0,018	0,159	-0,303	0,431	0,497	0,419	0,293	0,349	0,102	0,082	0,266
	Sig.	.	0,000	0,020	0,247	0,060	0,822	0,710	0,007	0,125	0,003	0,155	0,496	0,919	0,378	0,077	0,011	0,002	0,021	0,123	0,051	0,579	0,639	0,135
	N	36	35	36	24	36	35	34	34	32	35	35	35	33	33	35	34	35	30	29	32	32	35	33
2	CC	0,751	1,000	0,549	0,316	0,313	-0,042	0,090	0,586	0,352	0,593	0,349	0,012	0,065	0,335	-0,271	0,441	0,240	0,285	0,274	0,323	0,365	0,219	0,493
	Sig.	0,000	.	0,001	0,133	0,063	0,812	0,615	0,000	0,044	0,000	0,037	0,946	0,721	0,057	0,116	0,009	0,164	0,126	0,150	0,067	0,037	0,198	0,004
	N	35	36	36	24	36	34	34	35	33	36	36	36	33	33	35	34	35	30	29	33	33	36	33
3	CC	0,387	0,549	1,000	0,549	0,160	0,292	0,471	0,326	0,088	0,432	0,302	0,318	0,241	0,613	-0,555	0,428	0,168	0,353	0,509	0,379	0,386	0,315	0,530
	Sig.	0,020	0,001	.	0,005	0,343	0,089	0,004	0,056	0,627	0,008	0,073	0,059	0,177	0,000	0,000	0,010	0,328	0,052	0,004	0,030	0,026	0,061	0,001
	N	36	36	37	24	37	35	35	35	33	36	36	36	33	34	36	35	36	31	30	33	33	36	34
4	CC	0,246	0,316	0,549	1,000	0,247	0,256	0,372	0,378	0,169	0,176	0,111	0,258	0,357	0,143	0,005	-0,153	0,124	-0,054	0,563	0,080	0,303	0,261	0,344
	Sig.	0,247	0,133	0,005	.	0,245	0,239	0,088	0,069	0,465	0,410	0,606	0,224	0,086	0,524	0,982	0,475	0,562	0,823	0,012	0,723	0,182	0,217	0,117
	N	24	24	24	24	24	23	22	24	21	24	24	24	24	22	24	24	24	20	19	22	21	24	22
5	CC	0,317	0,313	0,160	0,247	1,000	0,328	-0,067	0,371	0,120	0,307	0,079	0,141	0,418	0,098	0,082	0,176	0,286	-0,062	-0,038	0,220	0,011	0,038	0,165
	Sig.	0,060	0,063	0,343	0,245	.	0,054	0,701	0,028	0,505	0,068	0,647	0,412	0,016	0,581	0,633	0,313	0,091	0,740	0,841	0,218	0,950	0,827	0,353
	N	36	36	37	24	37	35	35	35	33	36	36	36	33	34	36	35	36	31	30	33	33	36	34
6	CC	-0,039	-0,042	0,292	0,256	0,328	1,000	0,226	0,046	0,042	0,188	-0,009	0,245	0,157	0,302	0,005	0,031	0,146	-0,121	0,298	-0,095	0,002	0,142	0,320
	Sig.	0,822	0,812	0,089	0,239	0,054	.	0,205	0,801	0,818	0,286	0,959	0,163	0,392	0,093	0,980	0,866	0,411	0,523	0,117	0,611	0,991	0,423	0,070
	N	35	34	35	23	35	35	33	33	32	34	34	34	32	32	34	33	34	30	29	31	32	34	33
7	CC	0,066	0,090	0,471	0,372	-0,067	0,226	1,000	-0,080	-0,263	0,148	0,301	0,296	0,117	0,500	-0,433	-0,025	0,137	0,184	0,459	0,018	0,032	-0,125	-0,039
	Sig.	0,710	0,615	0,004	0,088	0,701	0,205	.	0,660	0,153	0,403	0,084	0,089	0,531	0,004	0,011	0,891	0,440	0,339	0,014	0,924	0,864	0,483	0,831
	N	34	34	35	22	35	33	35	33	31	34	34	34	31	32	34	33	34	29	28	31	31	34	32
8	CC	0,455	0,586	0,326	0,378	0,371	0,046	-0,080	1,000	0,583	0,481	-0,005	0,307	0,463	0,239	-0,176	0,550	0,431	0,379	0,438	0,529	0,368	0,263	0,594
	Sig.	0,007	0,000	0,056	0,069	0,028	0,801	0,660	.	0,000	0,003	0,976	0,073	0,008	0,187	0,311	0,001	0,010	0,039	0,017	0,002	0,038	0,127	0,000
	N	34	35	35	24	35	33	33	35	32	35	35	35	32	32	35	34	35	30	29	33	32	35	33
9	CC	0,277	0,352	0,088	0,169	0,120	0,042	-0,263	0,583	1,000	0,351	0,311	0,364	0,398	-0,071	0,086	0,365	0,079	0,257	0,152	0,446	0,672	0,715	0,514
	Sig.	0,125	0,044	0,627	0,465	0,505	0,818	0,153	0,000	.	0,045	0,078	0,037	0,029	0,709	0,641	0,043	0,667	0,187	0,448	0,014	0,000	0,000	0,003
	N	32	33	33	21	33	32	31	32	33	33	33	33	30	30	32	31	32	28	27	30	31	33	31
10	CC	0,488	0,593	0,432	0,176	0,307	0,188	0,148	0,481	0,351	1,000	0,626	0,259	0,253	0,438	-0,133	0,344	0,293	0,271	-0,130	0,406	0,075	0,283	0,489
	Sig.	0,003	0,000	0,008	0,410	0,068	0,286	0,403	0,003	0,045	.	0,000	0,127	0,156	0,011	0,447	0,046	0,088	0,147	0,503	0,019	0,680	0,095	0,004
	N	35	36	36	24	36	34	34	35	33	36	36	36	33	33	35	34	35	30	29	33	33	36	33

11	CC	0,245	0,349	0,302	0,111	0,079	-0,009	0,301	-0,005	0,311	0,626	1,000	0,384	0,334	0,341	-0,180	0,039	-0,075	0,248	-0,180	0,358	0,337	0,284	0,094
	Sig.	0,155	0,037	0,073	0,606	0,647	0,959	0,084	0,976	0,078	0,000	.	0,021	0,058	0,052	0,300	0,829	0,667	0,186	0,351	0,041	0,055	0,094	0,602
	N	35	36	36	24	36	34	34	35	33	36	36	36	33	33	35	34	35	30	29	33	33	36	33
12	CC	-0,119	0,012	0,318	0,258	0,141	0,245	0,296	0,307	0,364	0,259	0,384	1,000	0,578	0,369	-0,117	0,203	0,104	0,162	0,230	0,596	0,310	0,413	0,246
	Sig.	0,496	0,946	0,059	0,224	0,412	0,163	0,089	0,073	0,037	0,127	0,021	.	0,000	0,035	0,502	0,249	0,554	0,392	0,229	0,000	0,079	0,012	0,167
	N	35	36	36	24	36	34	34	35	33	36	36	36	33	33	35	34	35	30	29	33	33	36	33
13	CC	-0,018	0,065	0,241	0,357	0,418	0,157	0,117	0,463	0,398	0,253	0,334	0,578	1,000	-0,050	-0,236	0,041	0,024	0,200	0,264	0,534	0,250	0,329	0,031
	Sig.	0,919	0,721	0,177	0,086	0,016	0,392	0,531	0,008	0,029	0,156	0,058	0,000	.	0,794	0,194	0,826	0,896	0,317	0,193	0,002	0,182	0,062	0,873
	N	33	33	33	24	33	32	31	32	30	33	33	33	33	30	32	31	32	27	26	30	30	33	30
14	CC	0,159	0,335	0,613	0,143	0,098	0,302	0,500	0,239	-0,071	0,438	0,341	0,369	-0,050	1,000	-0,447	0,285	0,135	0,228	0,282	0,347	0,253	-0,005	0,552
	Sig.	0,378	0,057	0,000	0,524	0,581	0,093	0,004	0,187	0,709	0,011	0,052	0,035	0,794	.	0,009	0,107	0,453	0,242	0,145	0,060	0,177	0,979	0,001
	N	33	33	34	22	34	32	32	32	30	33	33	33	30	34	33	33	33	28	28	30	30	33	31
15	CC	-0,303	-0,271	-0,555	0,005	0,082	0,005	-0,433	-0,176	0,086	-0,133	-0,180	-0,117	-0,236	-0,447	1,000	-0,461	-0,354	-0,736	-0,555	-0,491	-0,203	0,008	-0,230
	Sig.	0,077	0,116	0,000	0,982	0,633	0,980	0,011	0,311	0,641	0,447	0,300	0,502	0,194	0,009	.	0,005	0,034	0,000	0,001	0,004	0,264	0,963	0,191
	N	35	35	36	24	36	34	34	35	32	35	35	35	32	33	36	35	36	31	30	33	32	35	34
16	CC	0,431	0,441	0,428	-0,153	0,176	0,031	-0,025	0,550	0,365	0,344	0,039	0,203	0,041	0,285	-0,461	1,000	0,645	0,690	0,350	0,513	0,435	0,158	0,554
	Sig.	0,011	0,009	0,010	0,475	0,313	0,866	0,891	0,001	0,043	0,046	0,829	0,249	0,826	0,107	0,005	.	0,000	0,000	0,058	0,003	0,014	0,372	0,001
	N	34	34	35	24	35	33	33	34	31	34	34	34	31	33	35	35	35	30	30	32	31	34	33
17	CC	0,497	0,240	0,168	0,124	0,286	0,146	0,137	0,431	0,079	0,293	-0,075	0,104	0,024	0,135	-0,354	0,645	1,000	0,565	0,471	0,356	-0,105	-0,092	0,280
	Sig.	0,002	0,164	0,328	0,562	0,091	0,411	0,440	0,010	0,667	0,088	0,667	0,554	0,896	0,453	0,034	0,000	.	0,001	0,009	0,042	0,568	0,598	0,108
	N	35	35	36	24	36	34	34	35	32	35	35	35	32	33	36	35	36	31	30	33	32	35	34
18	CC	0,419	0,285	0,353	-0,054	-0,062	-0,121	0,184	0,379	0,257	0,271	0,248	0,162	0,200	0,228	-0,736	0,690	0,565	1,000	0,513	0,664	0,448	0,281	0,180
	Sig.	0,021	0,126	0,052	0,823	0,740	0,523	0,339	0,039	0,187	0,147	0,186	0,392	0,317	0,242	0,000	0,000	0,001	.	0,005	0,000	0,017	0,133	0,334
	N	30	30	31	20	31	30	29	30	28	30	30	30	27	28	31	30	31	31	28	28	28	30	31
19	CC	0,293	0,274	0,509	0,563	-0,038	0,298	0,459	0,438	0,152	-0,130	-0,180	0,230	0,264	0,282	-0,555	0,350	0,471	0,513	1,000	0,209	0,337	0,149	0,192
	Sig.	0,123	0,150	0,004	0,012	0,841	0,117	0,014	0,017	0,448	0,503	0,351	0,229	0,193	0,145	0,001	0,058	0,009	0,005	.	0,276	0,074	0,440	0,309
	N	29	29	30	19	30	29	28	29	27	29	29	29	26	28	30	30	30	28	30	29	29	29	30
20	CC	0,349	0,323	0,379	0,080	0,220	-0,095	0,018	0,529	0,446	0,406	0,358	0,596	0,534	0,347	-0,491	0,513	0,356	0,664	0,209	1,000	0,361	0,420	0,316
	Sig.	0,051	0,067	0,030	0,723	0,218	0,611	0,924	0,002	0,014	0,019	0,041	0,000	0,002	0,060	0,004	0,003	0,042	0,000	0,276	.	0,043	0,015	0,083
	N	32	33	33	22	33	31	31	33	30	33	33	33	30	30	33	32	33	28	29	33	32	33	31
21	CC	0,102	0,365	0,386	0,303	0,011	0,002	0,032	0,368	0,672	0,075	0,337	0,310	0,250	0,253	-0,203	0,435	-0,105	0,448	0,337	0,361	1,000	0,612	0,478
	Sig.	0,579	0,037	0,026	0,182	0,950	0,991	0,864	0,038	0,000	0,680	0,055	0,079	0,182	0,177	0,264	0,014	0,568	0,017	0,074	0,043	.	0,000	0,007
	N	32	33	33	21	33	32	31	32	31	33	33	33	30	30	32	31	32	28	29	32	33	33	31
22	CC	0,082	0,219	0,315	0,261	0,038	0,142	-0,125	0,263	0,715	0,283	0,284	0,413	0,329	-0,005	0,008	0,158	-0,092	0,281	0,149	0,420	0,612	1,000	0,319
	Sig.	0,639	0,198	0,061	0,217	0,827	0,423	0,483	0,127	0,000	0,095	0,094	0,012	0,062	0,979	0,963	0,372	0,598	0,133	0,440	0,015	0,000	.	0,070
	N	35	36	36	24	36	34	34	35	33	36	36	36	33	33	35	34	35	30	29	33	33	36	33
23	CC	0,266	0,493	0,530	0,344	0,165	0,320	-0,039	0,594	0,514	0,489	0,094	0,246	0,031	0,552	-0,230	0,554	0,280	0,180	0,192	0,316	0,478	0,319	1,000
	Sig.	0,135	0,004	0,001	0,117	0,353	0,070	0,831	0,000	0,003	0,004	0,602	0,167	0,873	0,001	0,191	0,001	0,108	0,334	0,309	0,083	0,007	0,070	.
	N	33	33	34	22	34	33	32	33	31	33	33	33	30	31	34	33	34	31	30	31	31	33	34

Appendix D

This table contains the smell means from different modules. I have numbered the smells to be able to fit the table on the page:

Long Method = 1, Large Class = 2, Long Parameter List = 3, Data Clumps = 4, Duplicate Code = 5, Dead Code = 6 Speculative Generality = 7, Feature Envy = 8, Inappropriate Intimacy = 9, Message Chains = 10, Middle Man = 11, Lazy Class = 12, Data Class = 13, Incomplete Library Class = 14, Primitive Obsession = 15, Switch Statement = 16, Temp Field = 17, Refused Bequest = 18, Alternative Classes with Different Interfaces = 19, Parallel Inheritance Hierarchies = 20, Divergent Change = 21, Shotgun Surgery = 22, Comments = 23

	Omega-x	Gamma-S	Epsilon-P	Gamma-P	Delta-S	Delta-C	Delta-P	Zeta-C	Zeta-S	Kappa-S	Gamma-C
N	1,00	1,00	3,00	6,00	4,00	3,00	4,00	3,00	4,00	3,00	5,00
1	5,00	2,00	2,67	5,17	4,00	4,67	3,67	2,33	3,75	3,00	6,00
2	4,00	2,00	2,33	5,17	4,25	3,67	3,00	2,00	3,25	2,33	5,20
3	3,00	1,00	2,33	4,00	2,50	2,00	3,25	1,67	3,75	2,67	3,40
4	-	1,00	3,00	4,00	2,67	3,33	3,50	3,00	3,33	2,50	3,33
5	4,00	1,00	3,33	4,00	3,75	4,67	3,75	3,67	3,50	2,67	3,60
6	3,00	1,00	2,33	2,83	2,50	2,67	4,00	3,50	2,75	4,33	3,40
7	2,00	2,00	3,67	3,20	2,25	2,67	3,25	1,33	2,25	4,33	3,50
8	4,00	1,00	2,50	3,50	3,50	3,33	3,25	3,00	2,25	1,67	3,40
9	6,00	2,00	3,33	3,83	4,75	4,00	3,67	4,00	2,75	3,00	4,00
10	5,00	1,00	4,00	4,83	4,25	3,00	4,00	2,50	3,75	4,67	4,80
11	2,00	1,00	3,00	3,50	2,50	2,00	2,25	2,00	2,75	4,33	2,80
12	2,00	1,00	3,33	1,83	1,75	2,00	2,75	1,50	1,75	3,67	2,00
13	2,00	1,00	2,33	2,40	2,25	3,00	3,00	3,00	3,00	3,00	1,75
14	2,00	1,00	5,33	3,83	1,67	1,33	3,75	1,50	1,67	2,67	4,00
15	7,00	7,00	6,00	5,33	6,50	6,67	6,25	6,67	5,75	6,33	5,20
16	4,00	3,00	2,00	3,67	4,67	3,67	3,25	2,67	2,50	1,67	4,20
17	3,00	1,00	2,00	2,50	3,00	4,00	2,50	2,00	2,00	2,00	3,00
18	1,00	1,00	2,00	2,00	2,33	1,67	1,50	1,50	1,67	1,50	2,00
19	2,00	1,00	2,00	2,50	2,00	2,50	3,00	1,50	1,67	1,67	2,60
20	2,00	1,00	2,00	2,33	1,67	1,50	1,50	1,50	1,75	1,67	2,20
21	2,00	2,00	2,67	3,67	4,00	2,50	3,00	5,00	2,25	2,33	3,20
22	5,00	2,00	3,33	3,33	4,50	2,67	3,25	3,00	3,50	3,33	3,20
23	6,00	1,00	2,50	3,50	3,00	2,33	3,00	3,00	2,00	2,00	3,60