Master thesis

April 19, 2011

SmellTagger

Augmenting Design and Code Reviews with Multi-Touch Technology

Sebastian Müller
of Vaduz, Liechtenstein (05-707-948)

supervised by
Prof. Dr. Harald C. Gall
Michael Würsch and Emanuel Giger

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Master thesis

SmellTagger

Augmenting Design and Code Reviews with Multi-Touch Technology

Sebastian Müller

University of Zurich
Department of Informatics

s.e.a.l.
software evolution & architecture lab

Master thesis

Author: Sebastian Müller, mail@sebastian-mueller.li

Project period: 11.10.2010 - 20.4.2011

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

At this point I would like to thank all the people who were involved in this thesis. First of all, I thank professor Harald Gall for giving me the opportunity to write this thesis at his software evolution and architecture lab. Many thanks also go to Michael Würsch and Emanuel Giger for providing the idea of this application and for their great support.

Abstract

The new multi-touch technology that is used on devices such as the Microsoft Surface has the potential to fundamentally change the way how people interact with digital content. It provides the users with an intuitive and very natural user interface. This new interaction principle is widely used in various domains such as customer servicing. But very little research has been performed on how it can be used beneficially in the context of software engineering.

In order to contribute to overcome this issue we developed a Microsoft Surface application called *SmellTagger*. This application uses well-defined heuristics to automatically detect code smells in a software project. These code smells can provide the starting point for a collaborative design and code review. Therefore our prototype application demonstrates how multi-touch interfaces can be used advantageously in the field of software engineering. Additionally, our *SmellTagger* application can also be used to verify that multi-touch interfaces can foster the collaboration between software engineers.

The subsequent evaluation of the prototype application has shown that multi-touch interfaces are generally well accepted and intuitively easy to handle. It also became evident that our *SmellTagger* application fulfills the necessary requirements to be used beneficially in a code and design review process.

Zusammenfassung

Multi-touch Bildschirme werden in den letzten Jahren immer häufiger verwendet. Beispiele dafür sind das iPhone oder der Microsoft Surface. Diese neue Technologie hat das Potential, die Art und Weise wie Menschen mit digitalen Inhalten interagieren, grundlegend zu verändern. In vielen verschiedenen Bereichen, wie zum Beispiel kundenorientierten Dienstleistungen, wird diese neue Technologie bereits häufig eingesetzt. Allerdings gibt es bisher noch wenige Kenntnisse darüber, wie auch die Softwareentwicklung von diesem technologischen Fortschritt profitieren könnte.

Um dazu beizutragen diese neue Technologie auch in der Softwareentwicklung zu etablieren, haben wir *SmellTagger*, eine Applikation für den Microsoft Surface, entwickelt. Diese Applikation benutzt Heuristiken um ein Software Projekt automatisch zu analysieren und Code Smells zu finden. Diese gefundenen Code Smells können anschliessend als Ausgangslage für ein Design und Code Review benutzt werden. Dadurch kann mit Hilfe unserer Applikation erklärt werden, wie Multitouch Interfaces im Bereich der Softwareentwicklung vorteilhaft genutzt werden können. Zudem wird durch unsere *SmellTagger* Applikation ersichtlich, dass Multitouch Interfaces die Zusammenarbeit zwischen Softwareentwicklern fördern können.

Die anschliessende Evaluation unserer Applikation hat gezeigt, dass Multitouch Interfaces im Allgemeinen von den Benutzern akzeptiert werden und auch leicht zu bedienen sind. Ausserdem wurde ersichtlich, dass unsere *SmellTagger* Applikation die notwendigen Anforderungen erfüllt, um gewinnbringend in einem Design und Code Review verwendet werden zu können.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

User interfaces are one of the fastest changing fields in software engineering. Not long ago *Command Line Interfaces (CLI)* were the only available input method. *Graphical User Interfaces (GUI)* changed the interaction between humans and computers dramatically by introducing a new paradigm: the use of graphical symbols to display the interface elements the user can interact with. This paradigm shift improved the usability of user interfaces as a more intuitive interaction was possible. Today, this development towards more natural and intuitive user interfaces has resulted in the development of so-called *Natural User Interfaces (NUI)* that allow users to use natural and habitual finger and hand movements while interacting with multi-touch devices.

Multi-touch devices are devices with a touchscreen that is able to simultaneously handle at least three distinct touch points [Sys]. Today, multi-touch interfaces are particularly used on mobile devices, such as the iPhone,[1] and tables, such as the Microsoft Surface.[2] These devices have the capability to fundamentally change the way people interact with digital content. They provide intuitive possibilities to handle digital information similar to how we interact with real world content today. In combination with suitable applications, multi-touch interfaces can alleviate the exploration and analysis of complex data and situations. Since multi-touch devices can be used collaboratively they are perfectly suited for discussing complex facts in a group of experts.

While this new technology is being adapted in various domains, for example customer servicing ([Relc] and [Mic]), very little research has been performed on how it can be used beneficially in the context of software engineering.

1.2 Goal

The goal of this thesis is to explore the potential of multi-touch interfaces in the context of software engineering. Therefore this thesis will examine how multi-touch interfaces can be used beneficially in design and code reviews. In order to achieve this, traditional reviewing methods and visualization principles are analyzed and new interaction paradigms that follow well-known design principles are developed. These new interaction paradigms facilitate the interaction with complex, abstract and difficult to understand data during a review process.

[1] `http://www.apple.com/iphone`
[2] `http://www.microsoft.com/surface`

In order to assess these paradigms, we developed an application called *SmellTagger*, that demonstrates how multi-touch technology can be used to collaboratively find and tag code smells in a software project during a design and code review. Afterwards, an empirical evaluation will show that the prototype application can meet these requirements.

1.3 Structure

Chapter 1: Introduction explains the motivation and goal of this thesis. In ***Chapter 2: Related Work*** selective research work in the field of visualization techniques, design principles and design and code review techniques are presented. ***Chapter 3: Background*** provides an overview about the history of multi-touch interfaces with the focus on the Microsoft Surface as well as the services, techniques and principles used to implement our *SmellTagger* application. This prototype application is explained in detail in ***Chapter 4: Augmenting Design and Code Reviews with Multi-Touch Technology***. ***Chapter 5: Implementation Details*** presents interesting implementation details of our *SmellTagger* application and explains how the implementation was validated. In ***Chapter 6: Evaluation*** the evaluation that was performed after the successful implementation of the prototype application is described and its results are discussed. Finally in ***Chapter 7: Final Remarks*** the results of this thesis are reiterated and ideas how to improve our *SmellTagger* application are presented.

Chapter 2

Related Work

This chapter presents related work in the field of design and code reviews, information visualization techniques and approaches, as well as multi-touch user interfaces. First, an overview about information visualization techniques and principles is presented. The second part covers multi-touch interfaces and their influence on collaborative programming. In the last part, principles in a design and code review process are described.

2.1 Information Visualization Techniques and Principles

Chen [Che10] defines the term *Information Visualization* as *"computer generated interactive graphical representations of information"*. These graphical representations are used to transform complex, abstract and difficult to understand data into meaningful visual representations that can be analyzed much easier. Therefore, visualizations are powerful means to simplify complex structures: *"Visualization has been accepted as an useful means to understand complex data, because visual displays allow the human brain to study multiple aspects of complex problems in parallel"* [PGFL05]. That is why visualizations are often used to display large amounts of data in order to simplify data evaluation.

Stasko *et al.* localize the advantage of using visualizations in the *"use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software"* [SDBP98]. Therefore, visualizations have big advantages in displaying large data sets compared to other more traditional approaches such as lists or tables. Ware [War00] sums up the three most important advantages of information visualizations:

- Visualizations allow the viewer to see and comprehend huge amounts of data at the same time.
- Reoccurring patterns in visualizations can be recognized very fast.
- Visualizations facilitate the analysis of both large-scale and small-scale features of the data.

Although, visualizations are very helpful to simplify complex and abstract data, it is not a trivial task to develop an useful visualization, as Tufte [Tuf86] states. Therefore, in order to alleviate this task, he lists six basic principles that should be fulfilled by every feasible visualization:

- **Tell the truth:** It is important that graphical displays preserve their graphical integrity. This means that it is fundamental that a graphical display accurately represent the data. In order to achieve this, it is necessary to follow simple principles. For example it is important that visualizations represent data variation and not design variation. In that way it can be avoided that the viewer's eye mixes up changes in the design with changes in the data. By using clear, detailed, and thorough labeling, graphical distortion and ambiguity can be avoided.

- **Show the data:** The single purpose of a visualization is to display quantitative information. An artful design is nice, but it should not be an essential part of the graphic. That is why all unnecessary non-data information should be reduced to a minimum so that the viewer can focus on the data itself.

- **Present many numbers in a small space:** Since visualizations are often used to display large amounts of data, it is necessary to present a lot of information in a finite space. By using *"multi-functioning graphical elements"* it is possible to display complex and multivariate data in a small space. Multi-functioning graphical elements are single graphical elements that hold more than one piece of information. But it is important to use them carefully. If not designed well, they can lead to graphical confusion that can only be understood by the designer itself.

- **Help the viewer think about the information rather than about the methodology and the design:** *"Data graphics should draw the viewer's attention to the sense and substance of the data, not something else"* [Tuf86]. The design of a visualization should help the user to find the necessary information and facilitate the navigation through large and complex data sets. Additionally, it should help to draw conclusions about the information represented by the visualization. But the visualization itself should not be the focal point. By removing redundant data and unnecessary design elements, it is possible to help the viewer to concentrate on the most important information.

- **Encourage the eye to compare the data:** By comparing one piece of data in a complex set of information with another piece of data from the same set, it is possible to gain more information about the data set. That is why a visualization should encourage the viewer to compare data and therefore visualizations should provide possibilities to compare different aspects of data to each other.

- **Make large data sets coherent:** Frequently it is necessary to simplify large and complex data sets to make them comprehensible. But nevertheless the emerging data have to be coherent. This can be achieved by using multiple layers of information. Tufte proposes to use *"multiple viewing depths"* to create multiple layers of information. This means each visualization can be designed in a way so that at least three different viewing depths are available. Typically these three viewing depths are: the overall structure, detail information and the implicit context.

Tufte's six basic principles are very general rules. Shneiderman [Shn96] approaches the problem of creating a feasible visualization from a more practical viewpoint. Therefore he mentions seven concrete design hints that should be implemented in any feasible visualization:

- **Overview:** The visualization should provide the viewer with a general overview about the data that is represented.

- **Zoom:** The viewer should have the possibility to examine interesting elements of the data set in detail.

- **Details-on-demand:** Not all details should be displayed in the first place in order to avoid that too much information is presented at the same time. Instead the viewer should have the possibility to request detailed information about interesting objects.

- **Relate:** If there are any relationships between elements in the visualization, these relationships should be represented somehow.
- **History:** Especially in very large and complex visualizations, it is necessary that user actions are recorded in a way so that everyone can see what has been changed. Optimally it is also possible to undo these changes.
- **Extract:** It should be possible to extract information displayed in the visualization in some way. For example it could be possible to store interesting information in a file.

From these principles Shneiderman deduced a general pattern how a potential user interacts with visualizations. First, the user tries to get a general overview. If he finds an interesting detail, he sets the focus to this part of the visualization (*zoom*) and tries to get more information about that element (*details-on-demand*). However, as others have mentioned, this simple interaction scheme is not always true in reality [War00]. Instead, Ware suggests that a feasible visualization should support all of these principles mentioned by Shneiderman in any order.

Stephen Few also contributes to the research of general design principles. In his work [Few04] he mentions the *Gestalt principles*. The Gestalt principles describe how the human brain perceives *"pattern, form and organization in what we see"* [Few04]. These principles are relevant to the design of visualizations, since they explain how a potential viewer will perceive and organize different elements of the visualization. In the context of visualizations, the six most important *Gestalt principles* are:

- **Principle of proximity:** Elements that are close to each other are seen as belonging to one group.
- **Principle of similarity:** Elements that are similar in color, size, shape or orientation are often grouped together.
- **Principle of enclosure:** Elements that are enclosed by a visual border are often seen as belonging together.
- **Principle of closure:** Whenever possible the viewer tends to see complete figures, even if a part of that figure is missing.
- **Principle of continuity:** Humans have a preference for continuing figures. If elements are aligned with one another, they are perceived as belonging together.
- **Principle of connection:** Elements that are connected together, for example by a line, are received as being part of the same group.

While designing a feasible visualization it is important to keep these principles in mind since they explain how a future user will perceive various elements of the visualization. A part of these principles also explain how users interact with a visualization. Therefore such design principles can be used as a good starting point when developing a new visualization.

2.2 Multi-touch User Interfaces

Studies have shown that when working collaboratively programmers can produce higher-quality software in a shorter amount of time compared to situations when they are working alone [Nos98]. Additionally, a lot of programmers are thinking that programming collaboratively is more enjoying than working alone [WKCJ00]. However, collaborative programming is not a new invention and was already widely used years ago, for example in the form of *Pair Programming* or *Extreme Programming* [WKCJ00]. But until today collaborative programming has had a major drawback: the traditional input devices programmers are working with, namely mouse and keyboard, normally can not be used by multiple users at the same time. That is why in most cases of collaborative programming one of the programmers is actively working in the front of a traditional computer while the others are supporting him, for example by observing the actual work, thinking of better alternatives, looking up resources or watching for defects [WKCJ00].

Since a multi-touch interface allows developers to work on the same device at the same time without interfering each others, researchers have high expectations how multi-touch interfaces could change the way developers work together in a software project. There are various studies available that have tried to measure the influence of multi-touch interfaces on collaborative work. Peltonen *et al.* [PKS$^+$08] installed a large multi-touch display in the center of Helsinki and observed how people interact with each other while viewing pictures. They noticed that strangers work mostly separately, but in conflict situations, the problems were most often solved collaboratively. Another key observation was that newly arriving users were able to learn how to interact with the display from earlier users.

Izadi *et al.* [IBR$^+$03] used a multi-touch interface during a group discussion. The goal of this group discussion was to create an interactive poster on the multi-touch surface. The study showed that the participants had not any problem to learn the new interaction paradigms. But it also became evident that simultaneous interactions can lead to problems. Users are able to manipulate other user's data without their permission or destroy their work by removing it. In order to overcome these issues the study participants rapidly established social protocols of interaction to avoid such behavior.

Akerman *et al.* [APH$^+$10] contributed to the research by connecting two night clubs in Finland together using two interactive tables. On both ends the users were able to communicate to each others using free-hand drawings. The study showed that almost no interaction between the users took place. Most users drew their own images while preserving the others. Only on very rare occasions users worked together drawing one image. Nevertheless most participants were positively surprised by this form of collaboration.

Frisch *et al.* [FHD09] performed a study that examined how people use multi-touch and pen input on interactive surfaces. For that purpose the study participants were asked to perform spontaneous gestures for creating, moving and deleting diagram elements. The results showed that one-handed gestures and pen interaction were preferred compared to two-handed gestures. Another interesting point of the study was that *"most of the participants often stuck to the desktop paradigm"* [FHD09]. They were not able to perform a spontaneous gesture for specific tasks but rather mentioned that a button would do the work better. Some participants also suggested very unusual gestures like drawing "X" or "d" on the surface when trying to delete a node. Other participants dragged elements off the canvas in order to delete them. These behaviors are very interesting since most current devices do not support such interaction. Therefore these principles are suggestions for new interaction paradigms on multi-touch interfaces.

Wang *et al.* [WM08] developed a prototype application for a tabletop that can be used to *"support collocated and distributed agile planning meetings"* [WM08]. Various forms of interaction like finger touch, handwriting, gestures and voice control are supported. An evaluation of this prototype application examined if the prototype application does facilitate agile planning meetings and how the prototype application changes the behavior in agile planning meetings compared to a traditional pen&paper approach. The results showed that the participants described the interaction with the tabletop as very natural and easy to learn. Gestures were considered as more useful than traditional buttons, although some participants had difficulties to remember all the gestures. Voice control was classified as not very useful, particularly because it was not reliable enough and because the headset that was necessary for accurate voice recognition was not as intuitive as using finger gestures. Another interesting result was that handwriting text was not well accepted. A lot of the participants preferred to use the keyboard for text input because it was considered faster, more accurate and more readable.

All of the studies mentioned in this chapter have proven that multi-touch devices provide an easy to learn and natural interface that allows new forms of interaction with other users. Nevertheless some users seem to have difficulties to forget the desktop paradigm and switch to a new form of interaction. Therefore an important task when developing new multi-touch interfaces will be to facilitate the transition from traditional computer interfaces to sophisticated and natural multi-touch interfaces.

2.3 Design & Code Review

Among others, code reviews are a part in the software developing process that is suitable to be performed in a collaborative environment. A code review is the systematic examination of source code. Its goal is to find errors that have not been found during the development process [HK]. That is why it is an important part of the quality assurance of software.

Practically, the various forms of code reviews can be classified into three main categories: self code reviews, peer code reviews and tutor code reviews. Self code reviews require a software developer to perform a self-critical reflection of his own work. Tutor code reviews use *"the traditional hierarchical approach in which the expert considers the work of the novice"* [FC01]. But most often code reviews are performed as peer reviews, since it is generally accepted as the most practical form [WYCL08]. During a peer review *"a work product (such as requirements specification, design document, or source code) is examined by its designer (author) and one or more colleagues in order to evaluate its technical content and quality"* [Gar10].

In some cases a code review is already implemented in the software development process. For example the *Extreme Programming* paradigm supports the use of *Pair Programming* that is a special version of peer review. But the various review approaches do not vary only on the procedure but also on the degree of formality. For formal code reviews, the IEEE (Institute of Electrical and Electronics Engineers) [IEE08] provides a standard that defines structures, roles and procedures for various forms of code reviews. In less formal code reviews, the process is very similar to Pair Programming: two developers are working together and both of them are monitoring the work of each other.

But independent of the form of a code review it is necessary that the reviewers work collaboratively in order to reach their goal. In that point a multi-touch interface can provide advantages compared to traditional technologies since a multi-touch interface provides the users with a very natural and intuitive way to interact simultaneously with digital content.

Design reviews are very similar to code reviews. The same techniques can be used but the scope is a different one. As Parnas and Weiss state, *"the purpose of all design reviews is to find errors in the design and its representation"* [PW85]. Therefore multi-touch interfaces can support design reviews in the same way as they support code reviews.

2.4 Conclusion

Various studies have shown that multi-touch interfaces have the ability to foster the collaboration between the users. Most of the new interaction paradigms that are required to interact with multi-touch interfaces are easy to learn if the transition from the traditional desktop paradigm to new natural multi-touch interfaces works successfully.

Design and code review processes are suited to be supported by multi-touch interfaces since most review approaches require a high amount of collaboration between the reviewers. In order to design such an application that supports reviewers it is necessary to adhere basic design guidelines such as Tufte's design principles or Shneiderman's design hints. These guidelines are used as a staring point for our *SmellTagger* application.

Chapter 3

Background

In this chapter all the necessary background information to fully understand our *SmellTagger* application is presented. First, a short overview about the history of multi-touch interfaces is provided. Particular attention is paid to the Microsoft Surface that was used to develop this application. The Chapter *Using Software Metrics to Detect Code Smells* explains how code metrics can be used to detect code smells in a software project. Finally this chapter addresses how the necessary structural information and code metrics of a software project were retrieved from a web service in order to perform a code smell analysis.

3.1 The History of Multi-touch User Interfaces

The user interface is the place where the interaction between humans and machines takes place. Most user interfaces allow bidirectional interactions. This means the user has the possibility to enter input in order to manipulate and control the system and the machine has the possibility to create output in order to inform the user about the effects of the user's inputs. The basic goal of all user interfaces is to require minimal input in order to produce the desired output while simultaneously minimizing undesired output.

One of the first user interfaces used by computers was the *Command Line Interface (CLI)*. Using this interface, the user is able to use the keyboard to write commands that are interpreted by a command-line interpreter. This has major disadvantages: The user has to remember a lot of different commands that are not always intuitive. Additionally complex operations can be very difficult to remember. The output from the user interface is limited to basic text output that often includes error codes that are difficult to decode [RO09]. In order to overcome these issues, a new user interface paradigm, the *Graphical User Interface (GUI)* was developed. GUIs use graphical symbols to display the elements the user can interact with. These elements can be selected with a mouse or a similar input device. Since GUIs use metaphors for specific functions, for example a rubbish bin symbol to delete files, the operation of such a user interface is easy to learn. Generally their learning curve is not as flat as the learning curve for command line interfaces. This renders them interesting especially for less experienced users. Today, most user interfaces are based on a GUI. In some cases (for example *Matlab*) a CLI is integrated into the GUI in order to profit from the advantages of both interfaces. The next step in the development of user interfaces are *Natural User Interfaces (NUI)*. NUIs are interfaces which can be directly controlled by one or more senses. This covers a wide range of interface such as voice recognition interfaces, motion sensors or touch interfaces. This thesis focus on multi-touch interfaces that react on movements. Traditional input devices such as the mouse or keyboard are no longer necessary because the user can directly interact with the touch-sensitive surface of the user interface. This leads to a very intuitive handling since natural and habitual movements can be used to interact with the device.

Due to the increasing proliferation of touchscreens, NUIs became more and more important during the past few years. However, touch technologies have a much longer history than people realize, with development beginning in the 1970's [Sai10]. Development started with *PLATO IV*,[1] the first touch screen, developed in 1972. Ten years later, the first multi-touch system was developed. A frosted-glass panel and a camera were used to capture the shadows of fingers and other shapes [Meh82]. During the following years, multi-touch technologies were enhanced and implemented in a lot of different systems varying from large-scale screens to small mobile devices. However, prior to 2007, multi-touch technology was primarily used by researchers and in the education field. With the release of Apple's iPhone[2] multi-touch technology became widely known from a broader consumer perspective. In the same year, Microsoft presented their Microsoft Surface which was *"a key indication of this technology making the transition from research, development and demo to mainstream commercial applications"* [Sai10].

Most recent developments in the multi-touch area are combined in Microsoft's Surface 2.0.[3] This device is based on the Microsoft Surface 1.0 and ThinSight [HIB$^+$07] but has major enhancements. As a result, the Surface 2.0 is *"more than just a multi-touch surface. Since pixels have integrated optical sensors, the whole display is also an imager"*, that can *"see what is placed on it, including shapes, bar codes, text, drawings, etc."* [Bux11]. The examples mentioned in this chapter indicate that *"touch technologies are going to continue to evolve in terms of what they can sense and how they are used"* [Bux10]. However, not only technology developments are important. Of equal or perhaps greater importance is the progress of the conceptual models. That is to say, the way the user interface is used, and appropriate applications that exploit the possibilities the new technologies offer.

3.2 Microsoft Surface

The Microsoft Surface is a multi-touch device that allows users to interact with natural hand and finger gestures. It has a table-like form and its 360 degree interface allows user to interact from any side. The vision system consists of five cameras and it is responsible to *"sense objects, hand gestures, and touch on the screen and then it processes that input"* [Liba]. Based on concepts from 2001, the development process of the Microsoft Surface started in 2003 [Mic07]. Four years later, on May 30, 2007, Steve Ballmer, the CEO of Microsoft, unveiled Microsoft Surface at the *D: All things digital* conference in Carlsbad, California [Relb]. Since 2008 the Microsoft Surface unit can be purchased by customers. Until now, it is most commonly used in the commercial domain of customer servicing, namely in hotels, restaurants and casinos.

Figure 3.1 shows a schematic illustration of a Microsoft Surface unit. It consists of an acrylic surface with a diffuser (1). The diffuser is mainly responsibly for turning the surface into a multi-touch surface. The vision system (2) uses a near-infrared light spectrum (850 nanometer wavelength) to recognize the light that is reflected back when hands, fingers or other objects touch the screen. Microsoft Surface uses a more or less common desktop computer (3) with a slightly customized version of Microsoft Vista. A projector (4) is responsible to project the output image to the screen. The projector works on a resolution of 1024 x 768 pixels. For a 30 inch (76.2 centime-

[1]Programmed Logic for Automated Teaching Operations
[2]http://www.apple.com/iphone/
[3]http://www.microsoft.com/surface/

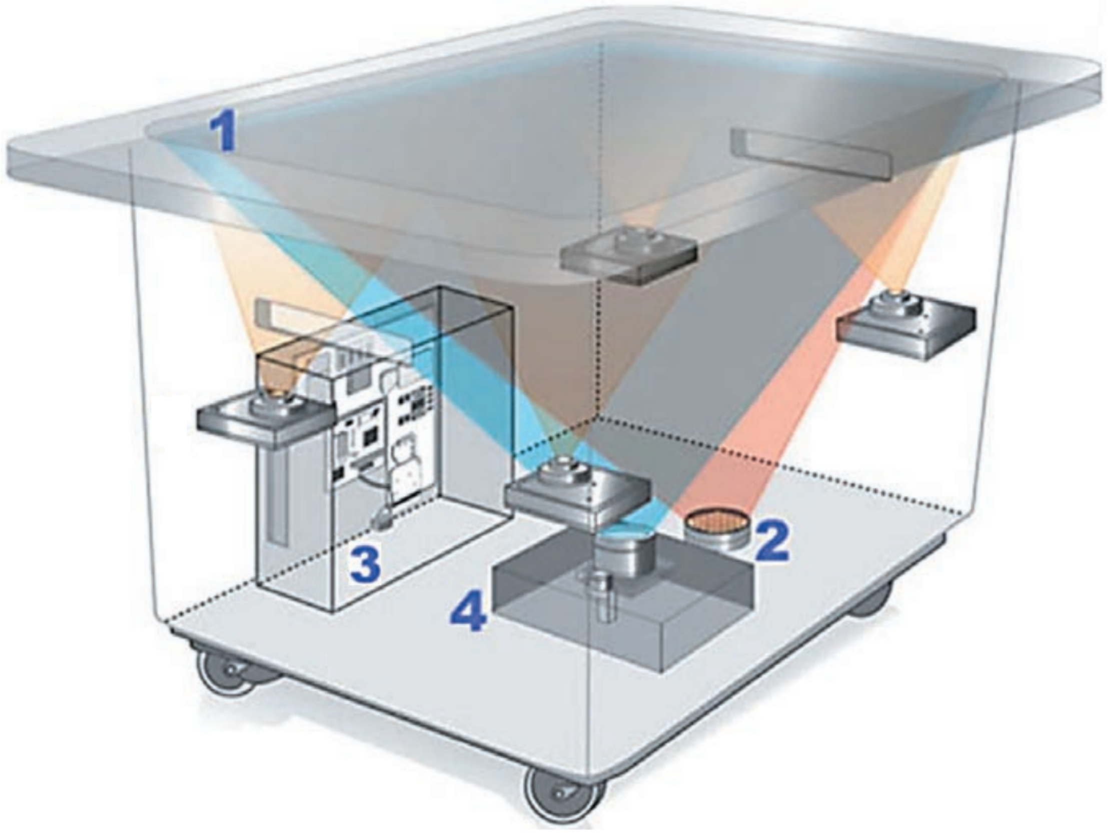


Figure 3.1: Schematic overview of a Microsoft Surface unit [New07]

ters) wide screen this resolution sometimes results in blurred images.

Normally, developing applications for Microsoft Surface is performed using the .NET framework and the Microsoft Surface SDK which enhances the .NET framework to provide access to the specific features of Microsoft Surface. It also includes a simulator that replicates the Microsoft Surface user interface on a workstation and allows developers to run their applications on their development machines.

Therefore is is not necessary to use a Surface unit during development. On the other hand, simulating special inputs on the Surface simulator, for example drag and drop operations, is not easy. For the development of our *SmellTagger* application both the Surface simulator and the Surface unit were used. More background information about the basic approach this application is built on is presented in the next chapter.

3.3 Using Software Metrics to Detect Code Smells

Refactoring is an important method in object-oriented software engineering to facilitate further development and maintainability [MT04]. The key idea is to change *"a software system in such a* *way that it does not alter the external behavior of the code, yet improves its internal structure"* [Fow00]. Therefore, the goal is to improve the nonfunctional attributes of the software, for example improved quality, better readability or less complexity. A critical point in the refactoring process is to decide when and what to refactor. To address this issue, Fowler introduced the term *Code Smell* [Fow00]. A code smell is any indication in the source code that may be caused by a deeper problem. Thus whenever a software engineer encounters a code smell, he should see it as a hint that something is wrong with this part of the code and he should consider to refactor it. In order to classify code smells, researchers have published a collection of code smells. Typically these collections consist of a description of a code smell, an example and suggestions how the code smell can be resolved. Examples of well known code smells are *Feature Envy*, *Shotgun Surgery* [Fow00] or *Combinatorial Explosion* [Ker01].

In order to find code smells in a software project, Fowler [Fow00] suggests to look for them manually by doing code inspections. Inspections are the most common kind of review practice in the field of software engineering. When following a well defined process, code inspections are very reliable approaches to find defects in code. On the other hand, Marinescu [Mar01] states that finding code smells manually is very time consuming, unrepeatable and non-scalable. For these reasons Marinescu proposed another approach that uses software metrics to detect code smells.

From a very general point of view, a metric is *"the mapping of a particular characteristic of a measured entity to a numerical value"* [LM10]. Metrics are used in almost all engineering disciplines, since most manufacturing outputs of an engineering process have to fulfill precise specifications. Metrics can be used to determine if those specifications are achieved. In other words, metrics can be used to control quality. That is why, *"accurate measurement is a prerequisite for all engineering disciplines, and software engineering is not an exception"* [LLL08]. In software projects, there are a lot of different metrics available. They vary from metrics that measure the size of a whole software project to metrics that are only available in the scope of a single operation like for example the *Nesting Depth*[4] of an operation. In particular, code metrics can be used to measure the size, the complexity and the quality of a software project. By using reference points, for example statistical thresholds, it is possible to interpret the measured values and to determine if they are in an acceptable range. From this, statements about the complexity, quality and size of a software project can be derived.

Although Fowler [Fow00] states that there is not a set of precise metrics that can be used to identify the need for refactoring, there are various approaches to use code metrics to detect design flaws in a software project. Basically, the detection can be done manually as suggested by Fowler [Fow00] or by applying automated heuristics. Others suggest alternatives, for example Emden and Moonen [vEM02] or Moha *et al.* [MGDM10]. Rosenberg *et al.* [RSG99] present an approach to automatically detect possible Code Smells. They suggest to thoroughly analyze every class that meets at least two of the following five criteria:

- Weighted methods per class (WMC)[5] > 100
- Coupling between object classes (CBO)[6] > 5
- Response for class (RFC)[7] > 100
- Number of methods (NOM)[8] > 40

[4]The metric *Nesting Depth* measures the maximum number of encapsulated scopes inside the body of a method [NDe]
[5]Sum of the McCabe Cyclomatic Complexity for all methods in a class
[6]Number of classes coupled to a given class
[7]Number of methods that can potentially be executed in response to a message received by an object of that class
[8]Total number of methods defined in the selected code

- Response for class (RFC) > 5 * Number of methods (NOM)

This is a very general approach that does not say anything in particular about the design flaw itself, but gives hints where a problem in a software project could be found. Lanza *et al.* addressed this issue and defined a more specific approach that uses various code metrics to identify code smells in a software project [LM10].

Lanza *et al.* introduced a method to *"both describe and characterize the structure of an object-oriented system by quantifying its complexity, coupling and usage of inheritance"* [LM10]. The basic idea of this so called *Overview Pyramid* is to *"put together in one place the most significant measurements about an object-oriented system, so that an engineer can see and interpret in one shot everything that is needed to get a first impression about the system"* [LM10]. The Overview Pyramid is the graphical representation of these significant measurements. Figure 3.2 shows a schematic illustration of this Overview Pyramid. Figure 3.3 shows an Overview Pyramid with actual values. As one can see these values are colored in blue, green and red. This color scheme indicates if the measurements are close to a low, medium or high statistical averages. For example one can see that the project displayed in Figure 3.3 has only one metric that is in an average statistical range (Number Of Methods). All the other values are either lower or higher than the average values of similar projects.

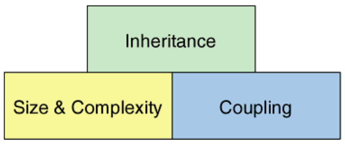


Figure 3.2: Schematic *Overview Pyramid* for object-oriented software systems

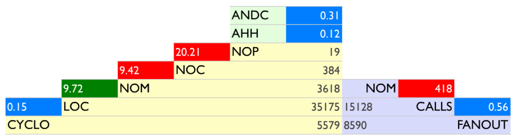


Figure 3.3: *Overview Pyramid* with actual values

The *Size & Complexity* part of the Overview Pyramid is used to describe how big and complex a software system is. Metrics in this part are well-known and widely used and can be directly calculated from the source code. In the example shown in Figure 3.3 one can see that the intrinsic complexity of the methods of the system is rather low (0.15). The size of the methods is in an average range (9.72 statements per method) but with 9.42 methods per class and 20.21 classes per package, the system is large. In the second part of the Overview Pyramid, the level of coupling in the software system is represented. *"Coupling is the measure of the strength of association established by a connection from one module to another. Strong coupling complicates a system since a module is harder to understand, change, or correct by itself if it is highly interrelated with other modules"* [SMC99]. That is why it is important that engineers have a close look at the coupling metrics of a system. In the Overview Pyramid coupling is measured by two direct metrics, *CALLS* and *FANOUT*. The CALLS metric measures the total number of distinct operation calls in the whole software project. The FANOUT metric counts the number of classes that are referenced by a class. An analysis of the example in Figure 3.3 shows that the components of the system are intensively coupled (4.18 calls per method). But this coupling is rather localized since the methods particularly call many methods from few different classes.

The top part of the Overview Pyramid uses two direct code metrics to provide an overview about the usage of inheritance in the software project. Lanza *et al.* argue that these metrics will help to detect how much usage of class hierarchies and polymorphism is present in the system. The *ANDC* metric measures the average number of derived classes and the *AHH* metric measures the average hierarchy height. Therefore, ANDC shows the amount of usage of inheritance relations, while AHH measures how deep these relationships are. In the example presented in Figure

3.3 one can see that each class has 0.31 subclasses on average. This value is rather low so that one can conclude that there is no broad hierarchy in the system. Additionally the hierarchy is not either deep since the average hierarchy height is only 0.12.

With an appropriate reference point, for example statistical thresholds, it is possible to interpret the Overview Pyramid. This allows an engineer to compare the actual system with a statistical base and analyze if the values are in the same range. But like the approach of Rosenberg *et al.*, this Overview Pyramid provides only a very general overview, and it only indicates in which part of the software system a problem may be located. To overcome this issue, Lanza *et al.* suggested a second approach that uses *"a composed logical condition, based on metrics, that identifies those design fragments that are failing the condition"* [LM10]. This composed logical condition is called a detection strategy since it can be used to detect code smells in a software project. Thus with the help of a detection strategy it is possible to make design rules quantifiable and therefore design problems detectable.

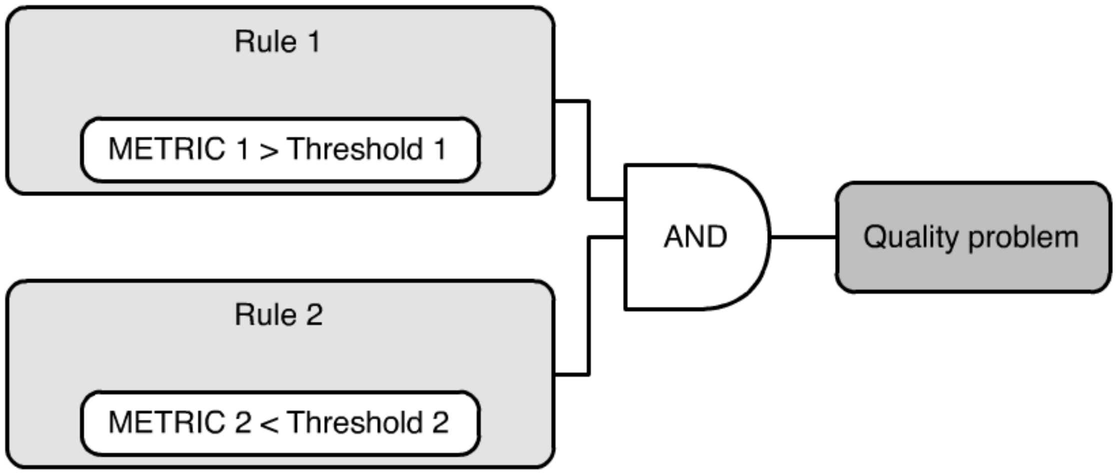


Figure 3.4: Schematic illustration of a detection strategy with two rules [LM10]

Figure 3.4 shows the basic design of a detection strategy. A detection strategy consists of rules and one or more logical gates. A rule is fulfilled if the corresponding equation is true. Afterwards the logical function can be analyzed and if it evaluates to true, a quality problem is detected. The example in Figure 3.4 consists of two rules that both have to be fulfilled in order to identify the quality problem. For a more practical example the code smell *God Class* is described by Riel [Rie96] and Fowler [Fow00] as follows:

- Classes that have many accessor methods defined in their public interface. This may imply that data and behavior is not being kept at the same place.
- Classes having methods that only operate on a proper subset of the instance variables.
- Classes that are large and complex. This may imply that these classes implements more than one responsibility.

In order to define rules that allow one to build a detection strategy to identify a God Class, these properties have to be mapped to proper code metrics that can be used to measure these characteristics. The mapping of God Class properties to the appropriate code metrics is shown in Table 5.1.

Symptom	**Metric**
Class that has many accessor methods and accesses a lot of external data	ATFD is more than a few
Class that is large and complex	WMC is high
Class that has a lot of methods that only operate on a proper subset of the instance variable set	TCC is low

Table 3.1: *God Class* symptoms and their mapping to appropriate metrics

Finally, these metrics have to be correlated into a composed logical condition. From the description of Fowler and Riel, it can be assumed that all these symptoms have to coexist. Therefore it is necessary to use a logical AND gate to connect the individual rules. From this information, the final detection strategy, shown in Figure 3.5, can be inferred.

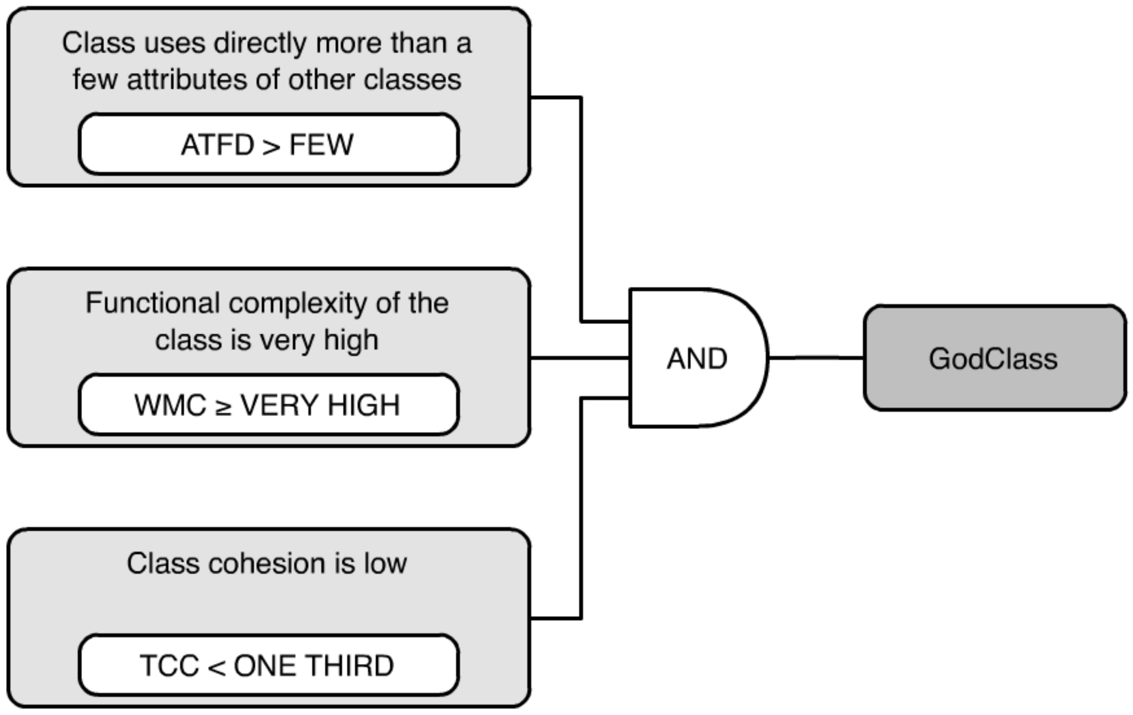

Class uses directly more than a few attributes of other classes
ATFD > FEW
Functional complexity of the class is very high
WMC ≥ VERY HIGH
Class cohesion is low
TCC < ONE THIRD
AND
GodClass

Figure 3.5: *God Class* detection strategy [LM10]

In order to find all God Classes in an object-oriented software system, this detection strategy has to be applied to all classes in the system. The result is a list of code smell candidates that have to be further analyzed in detail to rule out false positive results. This approach of Lanza *et al.* is used in our *SmellTagger* application. In Chapter 5.2 the implemented detection strategies and the further analysis are discussed in detail.

3.4 Retrieving Information from the SOFAS Web Service

In order to analyze a software project, it is necessary to have information about it. For example information about the structure of the software project (packages, classes and methods), the relationship between different entities of the software project or the plain source code of code entities. Additionally, in order to use the automatic detection strategies described in Chapter 3.3 it is also necessary to have software metrics about the classes and methods in the project. For these reasons, our *SmellTagger* application uses the *SOFAS* web service [Ghe10], to retrieve this information. This web service in particular allows one to analyze software projects online and then offers information about the structure, the metrics and the source code of these projects. This information is available in the form of a FAMIX model [TDDN00] and as an example can be accessed by sending SPARQL (SPARQL Protocol and RDF Query Language)[9] queries to the RESTful interface of SOFAS. A RESTful interface is an interface that complies with the REST (Representational State Transfer) constraints. Listing 3.1 shows an exemplary SPARQL query that retrieves the names of all packages of a software project from the web services.

Listing 3.1: SPARQL query to retrieve the names of each package in a software project

```
PREFIX java: <http://evolizer.org/ontologies/seon/2009/06/java.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT ?url ?name
WHERE {
  ?url rdf:type java:Package .
  ?url rdfs:label ?name
}
```

Using such SPARQL queries, the following information about the analyzed software project are retrieved during the startup phase of our *SmellTagger* application:

- Every package of the software project
- Every class and interface of each package
- All implementation-relationships between interfaces and classes
- All invocation-relationships between classes
- All inheritance-relationships between ancestors and descendants
- All the methods that are defined in classes or interfaces
- All call-relationships between methods
- The metrics defined in Table 3.2 for all interfaces and classes
- The source code for all interfaces, classes and methods

Table 3.2 provides an overview about those code metrics that can be retrieved from the SOFAS web service. As one can see there are three levels of scope available: metrics on class level, metrics on package level and metrics that represent values for the whole project.

[9] http://www.w3.org/TR/rdf-sparql-query/

Class	**Package**	**Project**
FANIN	CALLS	CALLS
FANOUT	NOM	AHH
CALLS	NOP	ANDC
HIT (only root classes)		NOC
NDC		NOP
NOA		
NOM		
NOPAR		

Table 3.2: Metrics provided by the SOFAS webservice

In the next chapter it is described how the approaches and principles presented in this chapter are used to actually implement our *SmellTagger* application.

Chapter 4

Augmenting Design and Code Reviews with Multi-Touch Technology

This chapter describes our *SmellTagger* application. This description covers a presentation of a typical use case of the application, the user interface, as well as the mechanisms that are used to interact with the application. Additionally, the design of our *SmellTagger* application is compared with the design principles described in Chapter 2.1.

4.1 Use Case

Our *SmellTagger* application supports software engineers when they have to find code smells in a software project. Since code smells are indicators for a deeper problem in the source code, they can be used as hints for software developers to decide if it is necessary to refactor part of the software in order to improve its quality. Therefore our application will mostly be used during the development phase.

In order to use the application beneficially it is necessary to first define a set of individual code smells that should be searched for. For each of these code smells a detection strategy has to be defined. It is possible to use well-known code smells such as God Class or Feature Envy as well as to define individual code smells. After the definition, the software project can be analyzed and the reviewers will get a set of code smell candidates. These candidates have to be further analyzed in detail to ensure that false positive results are ruled out. It is the responsibility of the reviewers to assess the findings and define strategies how the located design flaws can be resolved. Our *SmellTagger* application provides means to facilitate this task.

The review report (described in Chapter 4.2.4) that is generated after a review session can be used as a starting point for software developers which want to perform a refactoring after the review in order to resolve the detected design flaws. Ideally the resulting design is examined again for further code smells that indicate the need of further refactoring. Therefore our *SmellTagger* application is well fitted into a development process with a short feedback cycle.

4.2 Application

In this chapter, our *SmellTagger* application is described. First the requirements for this prototype application are presented. Afterwards the prototype itself is explained in detail. This covers the user interface as well as the way the user interacts with our application.

4.2.1 Requirements

Our *SmellTagger* application enables users to perform a code or design review session on a multi-touch user interface. For this purpose six key features are necessary:

- **Choosing input:** It is necessary that the user can choose a software project that should be analyzed during the review session.
- **Generate output:** The user has to be provided with a possibility to store the results of a review session.
- **Code smells analysis:** A heuristic is used to automatically detect code smells. These code smells are later manually inspected by the user to verify the automatic results. In order to be able to do that the user needs appropriate features to analyze potential code smells.
- **Exploit the features of Microsoft Surface:** The features of Microsoft Surface such as drag and drop, multi-touch or object-recognition are used to provide the user with a natural and intuitive user interface.
- **Collaboration:** The design and code review process takes place in a collaborative environment. Therefore the user interface has to foster the collaboration between different reviewers.
- **Design principles:** The user interface of the application follows the design principles presented in Chapter 2.1.

In summary it can be said that our *SmellTagger* application provides reviewers with the possibility to choose a project that should be analyzed. During this analysis, heuristics are used to automatically detect code smells. These code smells can be examined in detail with a new multi-touch enabled interface the design of which follows well-known design principles and supports reviewers to solve tasks collaboratively. As soon as the review is completed, it is possible to generate a report that summarize the insights of the review session.

4.2.2 Start-up Screen

During the start-up phase of the application the user has to choose a software project that he wants to analyze. Generally, there are two possibilities. First, it is possible to load a project from the SOFAS webservice described in Chapter 3.4. The second possibility is to load a project that is locally stored on the Surface unit. Figure 4.1 shows an example of this user interface. Here the user can choose to load one of the three remote projects (`argouml`, `findbugs` or `lucene`) or to load a project that was previously stored on the Surface unit (`argouml`, `eclipse` or `findbugs`). It is important to note that it is possible that projects are still locally stored but no longer available on the web service. This is the case if the user has eventually stored a project locally and it was afterwards removed from the web service. The project `eclipse` is an example for that. The projects `findbugs` and `argouml` are available both remotely and locally. This does not imply that both projects are exactly the same. For example a user could store a project locally and afterwards update the project on the web service. Therefore it is important to choose carefully which project should be loaded.

In either case, after choosing a project, the application loads the necessary data. This includes loading all the code elements such as packages, classes and methods, calculating respectively retrieving a predefined set of code metrics and source code, and calculating the relationship between different classes and methods. As soon as this is done, the user can start using our *SmellTagger* application.

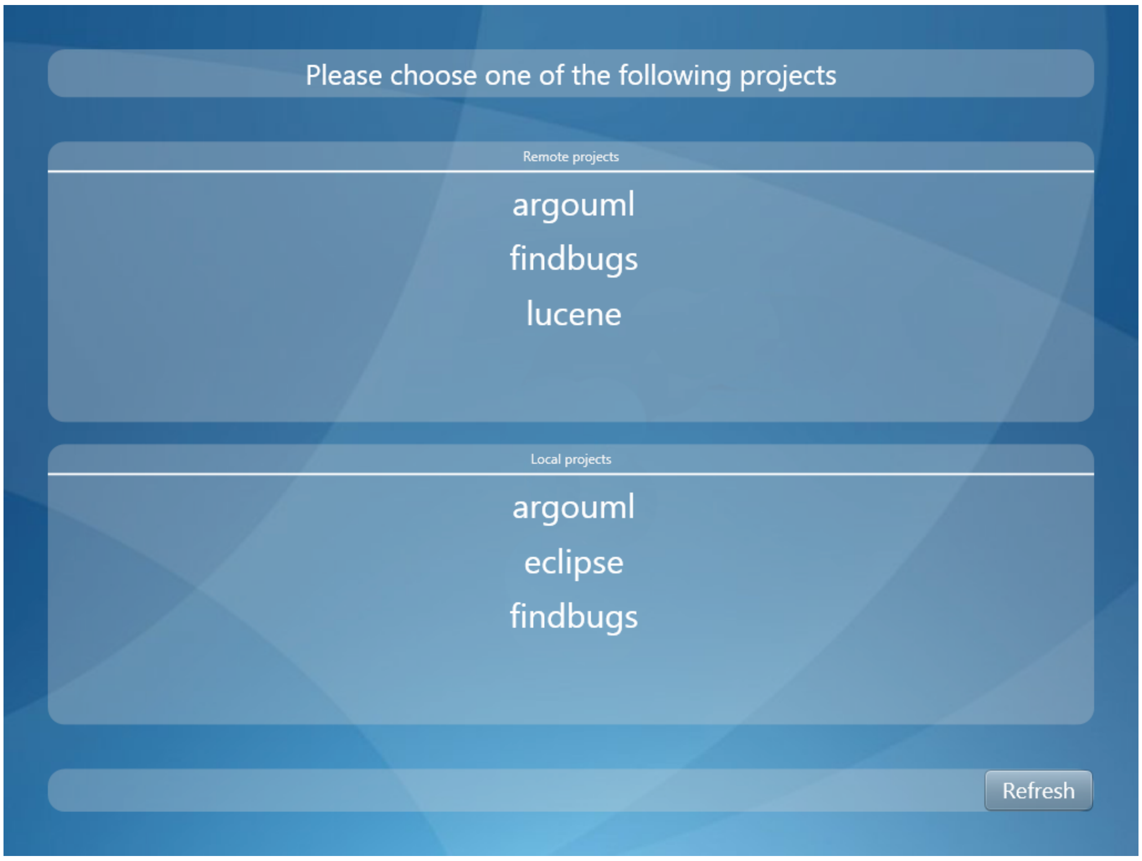
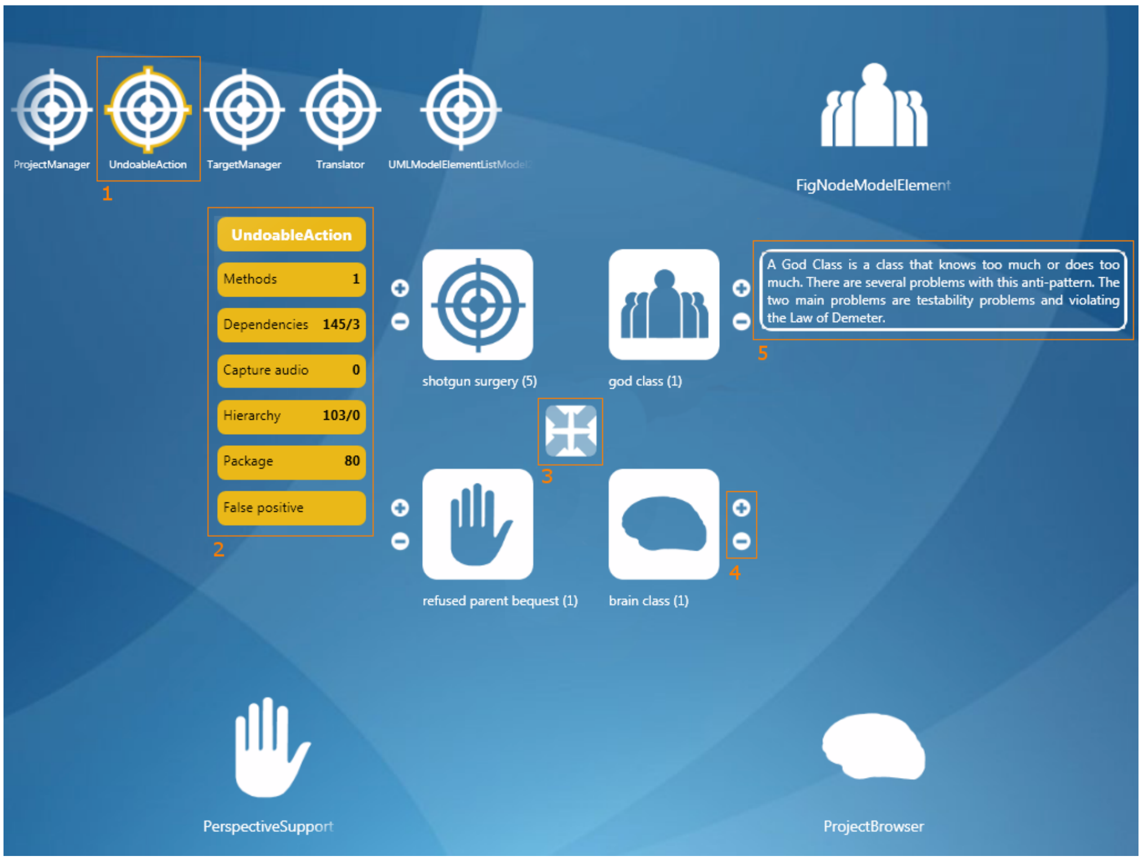


Figure 4.1: Start-up screen of our *SmellTagger* application

4.2.3 Main Window

After the loading process described in Chapter 4.2.2 is finished, the user can start to work with the application. From the *Main Window*, shown in Figure 4.2, it is possible to reach all the available functionalities of the application. The Main Window provides the user with a clearly arranged overview about the detected code smell candidates. For example the class `UndoableAction` (point 1 in Figure 4.2) is a class of the imported project that is a possible *Shotgun Surgery*. By selecting one of this code smell candidates, a small menu pops up that allows the user to choose which further information about the currently selected code smell candidate is displayed (point 2 in Figure 4.2). These features are described in the Chapter 4.2.4. While all these features are only available for a single class, it is also possible to get more information about the whole software project that is analyzed. By selecting the icon in the middle of the screen (point 3 in Figure 4.2) a similar menu is displayed that provides further possibilities to get information about the project. These possibilities are also described in Chapter 4.2.4. The buttons near each code smell logo (point 4 in Figure 4.2) can be used to zoom in or out if there are so many code smells detected that not all of them can be displayed in a reasonable size. Finally each code smell category has a small description (point 5 in Figure 4.2) that can be displayed by demand.



A God Class is a class that knows too much or does too much. There are several problems with this anti-pattern. The two main problems are testability problems and violating the Law of Demeter.

Figure 4.2: Main user interface. It is important to note that the numbers and frames colored in red are not part of the interface but are used to indicate different parts of the UI.

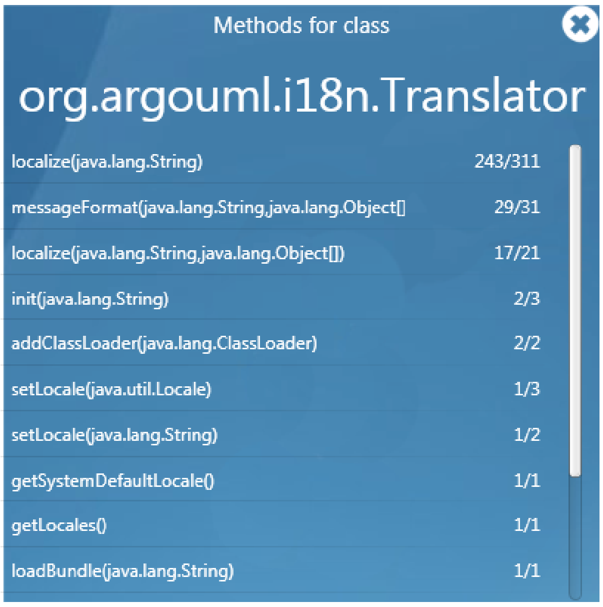
4.2.4 Features

In this chapter all the features of our *SmellTagger* application are described in detail. Basically there are two different kind of features: one that is available for each class and one that is only available for the whole project. The features that are available for each class are *Methods*, *Dependencies*, *Capture audio*, *Hierarchy*, *Package* and *False Positive*. The functionalities *Overview Pyramid*, *Packages Overview* and *Metrics Overview* are only available for the scope of the whole project. In order to explain these functionalities a lot of examples are used in this chapter. All these examples use data retrieved from an analysis of *ArgoUML*,[1] an open source tool for UML modeling.

Methods

In order to provide the user with an overview about the methods that are contained in a detected code smell or any other class, it is possible to list and explore all the methods of a class. As an example Figure 4.3 shows all methods defined in the Shotgun Surgery candidate `Translator`. The methods are sorted by the number of *Changing Classes* respectively the number of *Changing Methods*. The term Changing Methods subsumes those methods that call the actual method. The term Changing Classes subsumes the classes in which the methods that call the actual method are defined in. This allows for example to recognize at first glance which methods are responsible for triggering the cascading changes in a *Shotgun Surgery*. These are those methods that have a lot of Changing Methods. In the example shown in Figure 4.3 one can see that the method `localize(java.lang.String)` is affected by 311 other methods that are defined in 243 different classes. This is almost one-sixth of all classes in the whole project. Therefore it is obvious that this method is responsible for triggering the cascading changes in a Shotgun Surgery.

By selecting on of these methods it is also possible to get more information about the method itself. This information includes the method's source code, a list of methods that are defined in the same class, as well as a list of all Changing Methods and Changing Classes. Figure 4.4 shows the method `init(java.lang.String)` of the class `Translator`. One can see that this method has two Changing Classes. The class `Translator` in which this method is defined itself, and the class `Main` where the method `initTranslator()` is located. In order to increase the usability, the view in which the source code is displayed supports syntax highlighting and code folding. These two features and their implementation are explained in detail in Chapter 5.3.

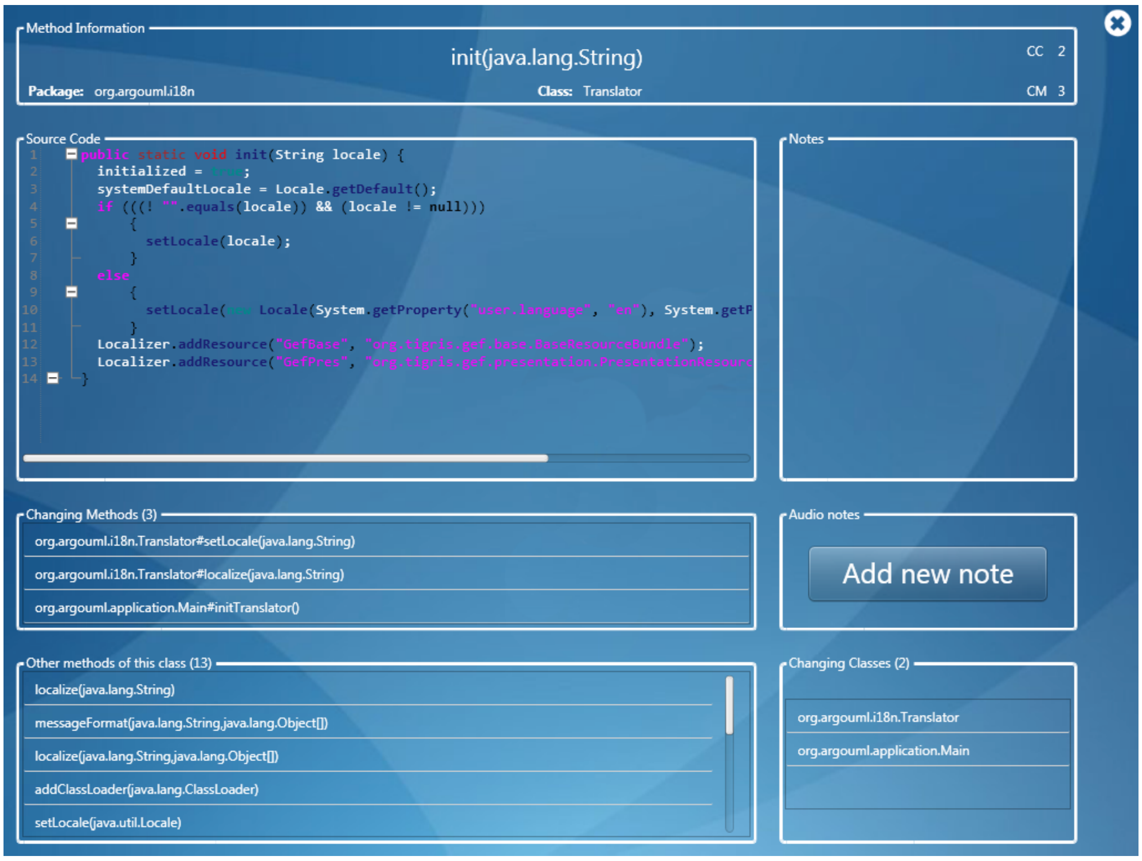

Methods for class

org.argouml.i18n.Translator

localize(java.lang.String)	243/311
messageFormat(java.lang.String,java.lang.Object[]	29/31
localize(java.lang.String,java.lang.Object[])	17/21
init(java.lang.String)	2/3
addClassLoader(java.lang.ClassLoader)	2/2
setLocale(java.util.Locale)	1/3
setLocale(java.lang.String)	1/2
getSystemDefaultLocale()	1/1
getLocales()	1/1
loadBundle(java.lang.String)	1/1

Figure 4.3: *Methods View* of class `Translator`

[1]http://argouml.tigris.org/


Method Information

init(java.lang.String)

CC 2

Package: org.argouml.i18n **Class:** Translator CM 3

Source Code

```
public static void init(String locale) {
    initialized = true;
    systemDefaultLocale = Locale.getDefault();
    if (((! "".equals(locale)) && (locale != null)))
        {
          setLocale(locale);
        }
    else
        {
          setLocale(new Locale(System.getProperty("user.language", "en"), System.getP
        }
    Localizer.addResource("GefBase", "org.tigris.gef.base.BaseResourceBundle");
    Localizer.addResource("GefPres", "org.tigris.gef.presentation.PresentationResourc
}
```

Notes

Changing Methods (3)

org.argouml.i18n.Translator#setLocale(java.lang.String)
org.argouml.i18n.Translator#localize(java.lang.String)
org.argouml.application.Main#initTranslator()

Audio notes

Other methods of this class (13)

localize(java.lang.String)
messageFormat(java.lang.String,java.lang.Object[])
localize(java.lang.String,java.lang.Object[])
addClassLoader(java.lang.ClassLoader)
setLocale(java.util.Locale)

Changing Classes (2)

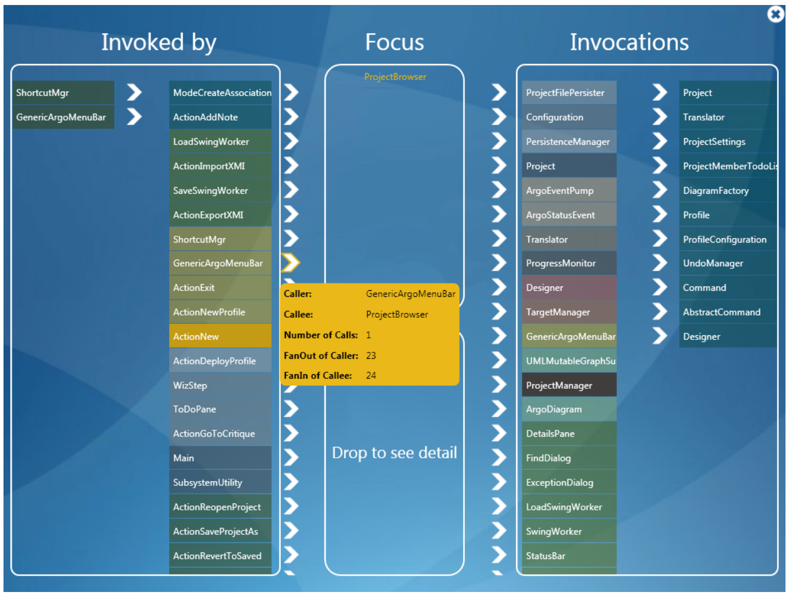
org.argouml.i18n.Translator
org.argouml.application.Main

Figure 4.4: *Methods View* of `Translator#init(String)` with detailed information

Dependencies

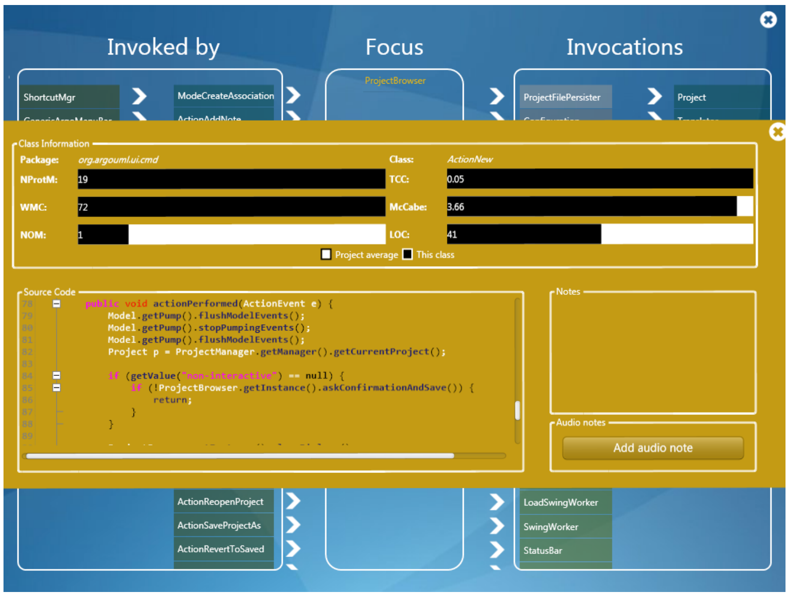
Often during a review session it is necessary to examine classes that are invoked by a specified class or classes that invoke the specified class. Using the view displayed in Figure 4.5 it is possible to see these dependencies for the next two hierarchy levels directly above or below the actual class. In this example the class `ProjectBrowser` is examined. In the right part of the screen in the first list one can see all the classes that are called by the class `ProjectBrowser`. The second list on the right part shows the invoked classes of class `ProjectManager`. In a similar way the left part of the screen displays the classes that are calling the class `ProjectBrowser`. By dragging class elements and dropping them on the appropriate place the focus can be switched to the dropped class so that the dependencies of this class are displayed. In the same way it is possible to display additional details of a class (Figure 4.6). These details include the source code, written notes, code notes and a selected set of metrics that are compared to the project average.

In Figure 4.5, all classes are colored in different colors. These colors represent the packages the classes belong to. That means if two classes are colored in the same color, they both belong to the same package. Additionally it is also possible to examine the relationship between two classes by selecting the arrow that represents this relationship. By selecting one of the arrows displayed in Figure 4.5 a new element is displayed that contains information about the call relationship between these two classes. In the example shown in Figure 4.5, the relationship between the classes `GenericArgoMenuBar` and `ProjectBrowser` is illustrated. One can see that there is just one call between these two classes. As an additional information it is apparent that the class `GenericArgoMenuBar` calls another 22 classes (*FanOut of Caller*) and that the class `ProjectBrowser` is called by another 23 classes (*FanIn of Callee*).



Caller:	GenericArgoMenuBar
Callee:	ProjectBrowser
Number of Calls:	1
FanOut of Caller:	23
FanIn of Callee:	24

Figure 4.5: *Dependencies View* of class `ProjectBrowser`



Class Information

Package:	*org.argouml.ui.cmd*	Class:	*ActionNew*
NProtM:	19	TCC:	0.05
WMC:	72	McCabe:	3.66
NOM:	1	LOC:	41

□ Project average ■ This class

Source Code

```
public void actionPerformed(ActionEvent e) {
    Model.getPump().flushModelEvents();
    Model.getPump().stopPumpingEvents();
    Model.getPump().flushModelEvents();
    Project p = ProjectManager.getManager().getCurrentProject();

    if (getValue("non-interactive") == null) {
        if (!ProjectBrowser.getInstance().askConfirmationAndSave()) {
            return;
        }
    }
```

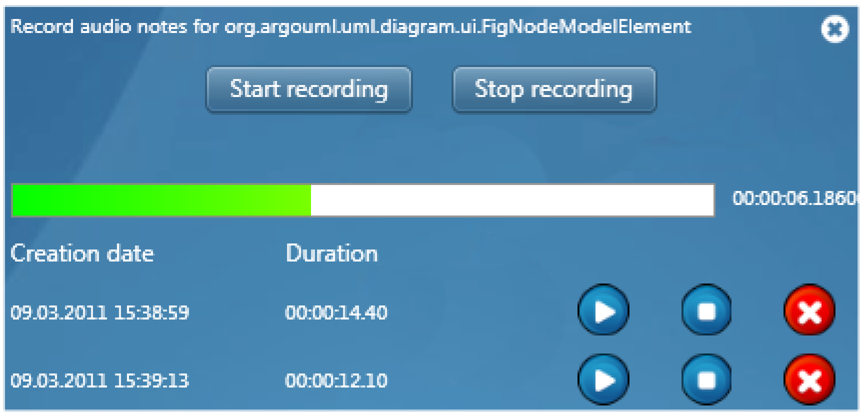
Figure 4.6: *Class Detail View* for class `ActionNew`

Capture audio

The Microsoft Surface provides the user with the possibility to use a virtual keyboard to enter text input. But a lot of people do not like to use virtual keyboards for various reasons. The most common ones are:

- Ergonomic issues since typing on a virtual keyboard requires to keep the wrists in an inconvenient angle.
- Space screen limitations since a full-size QWERTY virtual keyboard consumes a lot of space on the screen.
- A lot of virtual keyboards do not provide tactical feedback when pushing keys down.
- Very often it is more difficult to accurately hit a key on a virtual keyboard than on a traditional hardware keyboard.
- Virtual keyboards do not match with the Surface concept, except if it would be possible to use multiple mini keyboards.

In order to overcome these issues, our *SmellTagger* application provides a feature to record audio notes for each class. Figure 4.7 shows an example with two different audio notes for the class `FigNodeModelElement`. These audio notes are recorded in the *WAVE* format using the open source audio library NAudio.[2] In order to save memory the audio files are converted into the *MP3* format after recording. To do this the MP3 encoder *LAME*[3] is used.



Record audio notes for org.argouml.uml.diagram.ui.FigNodeModelElement

Creation date	Duration
09.03.2011 15:38:59	00:00:14.40
09.03.2011 15:39:13	00:00:12.10

Figure 4.7: User interface that allows to capture audio

With these audio notes it is possible to record notes much faster than using traditional approaches as for example typing or handwritten notes. Unfortunately the Microsoft Surface does not have a built-in audio recorder. That is why in order to use this feature it is necessary to use an external microphone. As an additional feature the audio notes are linked to the appropriate class when the user decides to generate a review report (Chapter 4.2.4). In Chapter 7.2.5 it is described how audio notes could be converted automatically to text to make this feature even more useful.

[2]http://naudio.codeplex.com/
[3]http://lame.sourceforge.net/

Hierarchy

In order to analyze a code smell candidate it is important to know its relationship to other classes. There are various kind of relationships between classes but one of the most significant is inheritance. It is important to know if a class implements an interface, extends a super class or is a base class for a lot of subclasses. Especially when analyzing code smells that suffer from *"classification disharmonies"* [LM10], that is classes that are not in harmony with their ancestors and their descendants, it is important to have the possibility to take a closer look at the inheritance status of a class. The code smell *Refused Parent Bequest* presented in Chapter 5.2.2 is an example of a classification disharmony.

A view that addresses this requirement is shown in Figure 4.8. In this example, the class `ActionNewAction` is located in the center. On top of this class one can see its super class called `AbstractActionNewModelElement`. The six direct descendants are shown at the bottom of the view. The class `ActionNewAction` implements one interface called `ModalAction`. It is located on the left side of the view. Each rectangle that represents a class is colored in a specific color that represents the package the class belongs to. Therefore classes that are in the same package have the same color. In this example one can see that the class `ActionNewAction` and its descendants are in the same package while its ancestor and the interface `ModalAction` are in different packages.

Using drag and drop it is possible to change the class that should be in the focus of this view. When dropping the dragged class on the marker called *Focus* the view is redrawn and the descendants and ancestor of the dropped class are displayed. Additionally it is also possible to view the details of a class as described in Chapter 4.2.4 by dropping a class or interface over the marker called *Detail*. For interfaces there is also a third option. When placed on the marker *Implementations* it is possible to browse through all classes that also implement this interface.

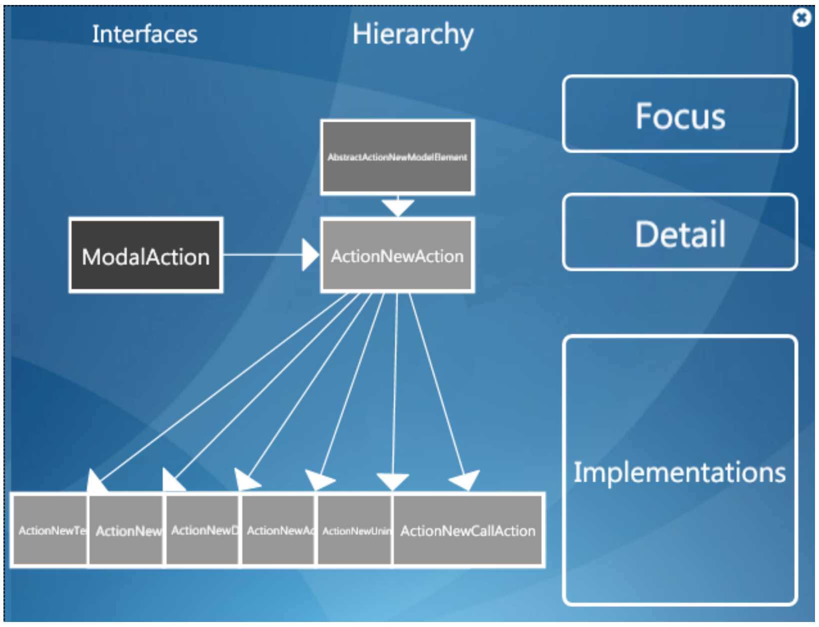


Figure 4.8: *Hierarchy View* of class `ActionNewAction`

Package

According to Lanza & Marinescu [LM10] design entities of a software project such as classes, methods or interfaces can be described by three elements. The first element, the *Having* element describes the scope of a design entity. For example a class belongs to a package and a method is defined in a class. The *Using* element describes the relationship between the measured entity and other entities. A class that is communicating with another class would be an example for that element. The last element, the *Being* element describes the properties of a design entity. The size of a class or the visibility of a method are examples for that element. While the features and views described so far address only one or two of these aspects, the *Package View* combines all three elements.

The example in Figure 4.9 shows the scope of the class `ProfileConfiguration` in the package `org.argouml.kernel`. The scope represents the *Having* element and it includes all other classes that also belong to the package `org.argouml.kernel`. In this view, classes are visualized using circles. These circles can be used to address the *Being* element: various metrics of the displayed class can be used to define the size of the circle and the color of the font. In this example the number of lines of code of a class is used to calculate the radius of a circle and McCabe's cyclomatic complexity is used to calculate the font color of a circle. The lines between the circles are used to visualize the *Using* element. If two classes communicate with each other, they are connected with a line. The line color indicates the communication intensity. Therefore the number of calls between two classes is mapped to a finite number of colors between black and white. The brighter a line is, the more communications take place between the two involved classes.

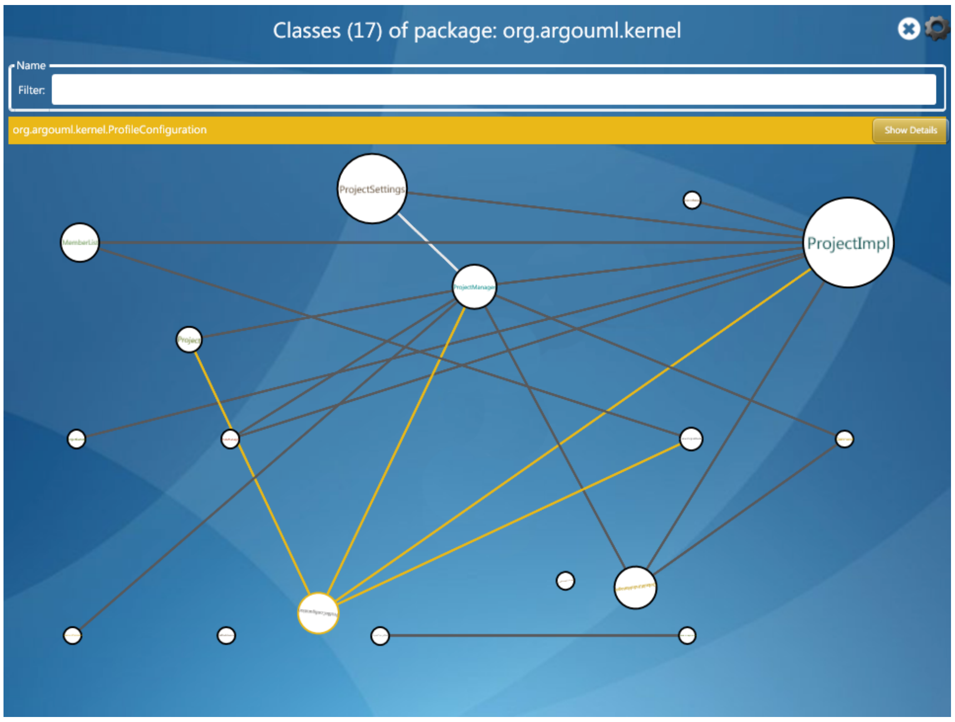

Classes (17) of package: org.argouml.kernel
Name
Filter:
org.argouml.kernel.ProfileConfiguration
Show Details
ProjectSettings
MemberList
ProjectImpl
ProjectManager
Project

Figure 4.9: *Package View* with circular layout. The individual classes are visualized as circles. The lines that connect the circles indicate call relationships.

In order to make the application more customizable and provide the user with already well-known metaphors, our *SmellTagger* application uses supplementary visualization approaches for this feature. The first approach shown in Figure 4.13 uses a house metaphor that was described in a three-dimensional variation by Gall and Boccuzzo [BG07]. *"The idea of this metaphor is to represent software entities such as classes as houses. A well-designed class then looks like a well-shaped house, whereas a problematic class results in a miss-shaped house"* [BG07]. Our *SmellTagger* application uses three different metrics to create a house. Two define the width and the height of the body of the house and one is used to calculate the height of the roof. In the example shown in Figure 4.13, McCabe's cyclomatic complexity is used to define the height of the roof and the number of lines of code and the number of methods are used to calculate the width respectively the height of the house's body. The reference house that represents a well-shaped house is calculated by using average metrics from all classes that are located in the displayed package. It is obvious at first glance that in this example there is not a class that is problematic. All houses are more or less well-shaped. The houses of the classes `ProjectSettings` and `ProjectImpl` are a little bit too wide, indicating that these two classes contain a lot of code. On the other hand the houses of classes `Command`, `ProjectFactory` or `NonUndoableCommand` are relatively high. This is a hint that these classes implement a lot of methods that are not very long.

The second approach uses kiviat diagrams [vM77] to visualize code metrics. Kiviat diagrams are suited to visualize multivariate data, since in contrast to other polymetric views the number of attributes that can be represented is theoretically not limited. However it is not a good choice to display more than ten attributes at the same time. Figure 4.10 shows an example of a kiviat diagram. The six metrics *metric1, metric2, ... , metric6* are mapped on circular axis. For each metric the value is plotted on the corresponding axis. These points are then connected to the adjacent points with a straight line. Sometimes the resulting polygon is filled with a color as shown in Figure 4.11. It is also possible to map more than one data series into the same diagram. Figure 4.12 shows a kiviat diagram where two data series are mapped.

Pinzger [Pin05] describes an approach called *ArchView* that uses kiviat diagrams to visualize and compare code metrics of different releases of the same project. A similar approach is shown in Figure 4.14. In that example kiviat diagrams are used to represent classes. Each kiviat diagram visualizes two data series with five metrics. The first series, shown in dark-green, visualizes the averages for the whole project. The second series, shown in light-green, represents the values for the actual class. Thereby this visualization provides a possibility to compare the properties of a class directly with other classes as well as with the average values for the whole project.

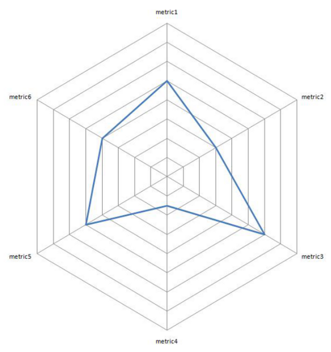


Figure 4.10: Kiviat diagram with one data series

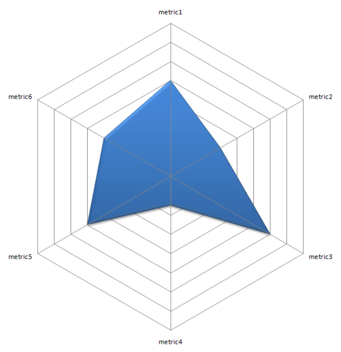


Figure 4.11: Filled kiviat diagram

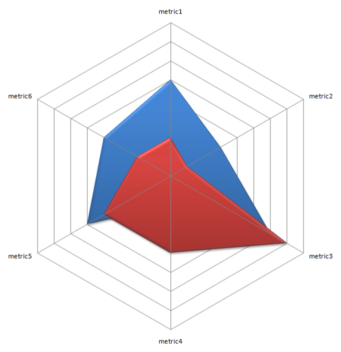


Figure 4.12: Kiviat diagram with two data series

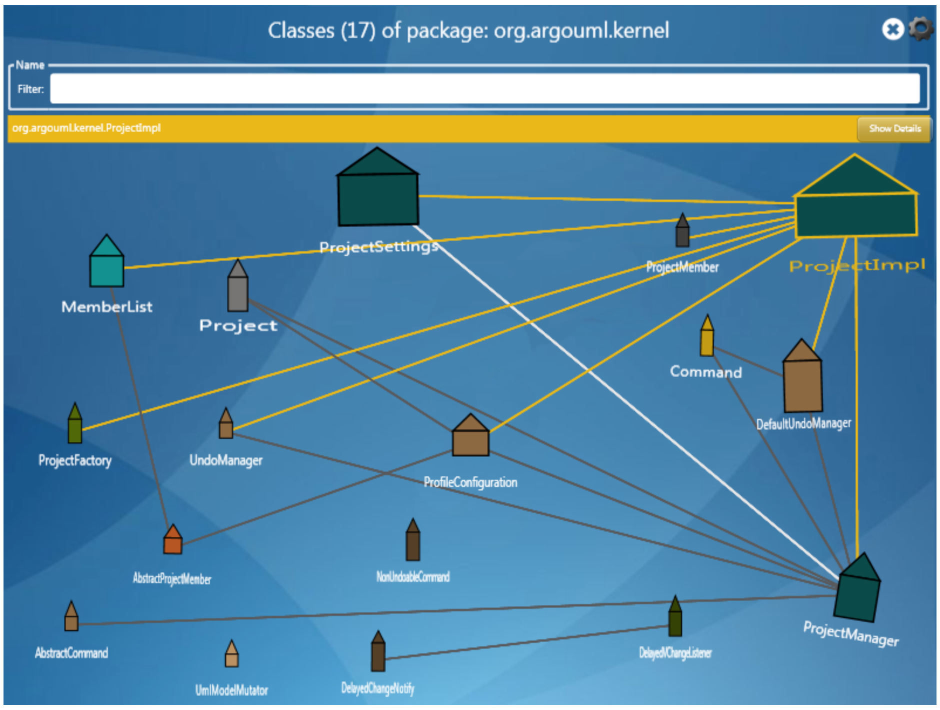


Figure 4.13: *Package View* with house layout

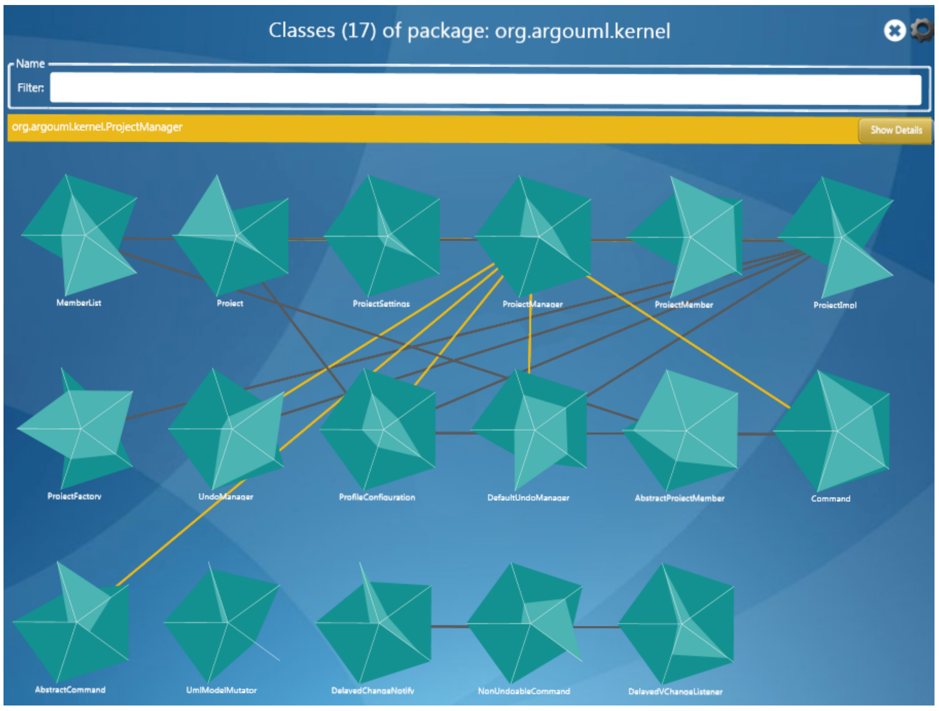


Figure 4.14: *Package View* with *Archview*-based layout

False Positive

The code smell classes presented in our *SmellTagger* application are found by detection strategies based on code metrics. How these detection strategies are working in general is described in Chapter 3.3. A more detailed description about the detection strategies used for this application can be found in Chapter 5.2. It is important to note that each detected code smell is only a candidate. Each candidate has to be inspected manually to verify if it is a code smell or a false positive result of the detection strategy. In order to provide the user with a possibility to dismiss false positive results, it is possible for each code smell to set a flag that indicates a false positive result. In this case the visual representation of that code smell is faded out, as shown in Figure 4.15 for the class `TargetManager`. Code smells that are marked as false positive results are nevertheless mentioned in the review report (Chapter 4.2.4) so that it stays traceable which code smells were sorted out.

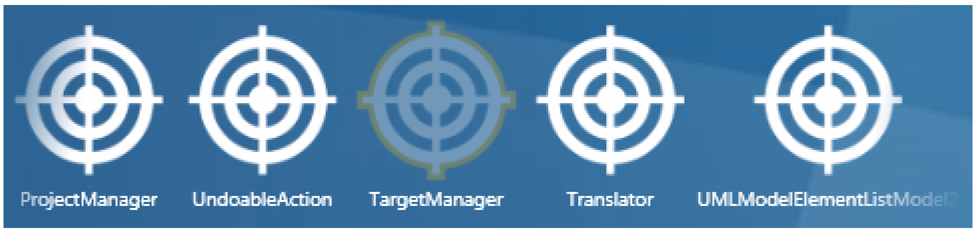


Figure 4.15: Code smell candidate `TargetManager` is marked as false positive.

Overview Pyramid

In Chapter 3.3 it is explained how the *Overview Pyramid*, developed by Lanza and Marinescu [LM10], provides means to both visualize and compare the properties of a software system. For example from the Overview Pyramid shown in Figure 4.16 one can conclude that *ArgoUML* is intensely using inheritance. With a value of 0.54, the average number of derived classes (ANDC) is far above the average value (0.41) for a Java project as noticed by Lanza and Marinescu [LM10]. Also the average hierarchy height (AHH) is explicitly higher than the statistical average (0.21). This allows to draw the conclusion that *ArgoUML* has both a broad and deep hierarchy. Classes tend to have a lot of descendants and the inheritance relationship spans many hierarchy levels. Additionally it is also possible to draw conclusions about the size and complexity of a software project. In the *ArgoUML* project each class defines 6.08 methods on average. This is a hint that *ArgoUML* has an average complexity since this value is very close to the statistical average [LM10]. On the other hand each method contains 13.47 lines of code on average. Compared to the statistical average this is a hint for a rather high complexity.

Cohesion and coupling are the two fundamental attributes that have a major impact on the quality of a software project [Alg07]. *"Coupling refers to the degree of interdependence between software system components"* [Alg07]. Class cohesion measures how closely the methods in a class are related to each other. The general goal is to achieve high cohesion and low coupling. A class with high cohesion follows the *Single Responsibility Principle* that states that each class should do one thing well but nothing else. Therefore it is important that a software engineer is aware of the cohesion and coupling in a system. The two ratios Calls/NOM and FANOUT/Calls are indicators for the system coupling. The coupling intensity (Calls/NOM) shows that each method calls

2.88 other methods on average. The coupling dispersion (FANOUT/Calls) indicates that 86% of all calls are not calls between methods that are in the same class. These two ratios indicate that *ArgoUML* is intensively coupled. To make it even worse the calls causing this intensive coupling are mostly not local calls, but involve methods from a lot of different classes.

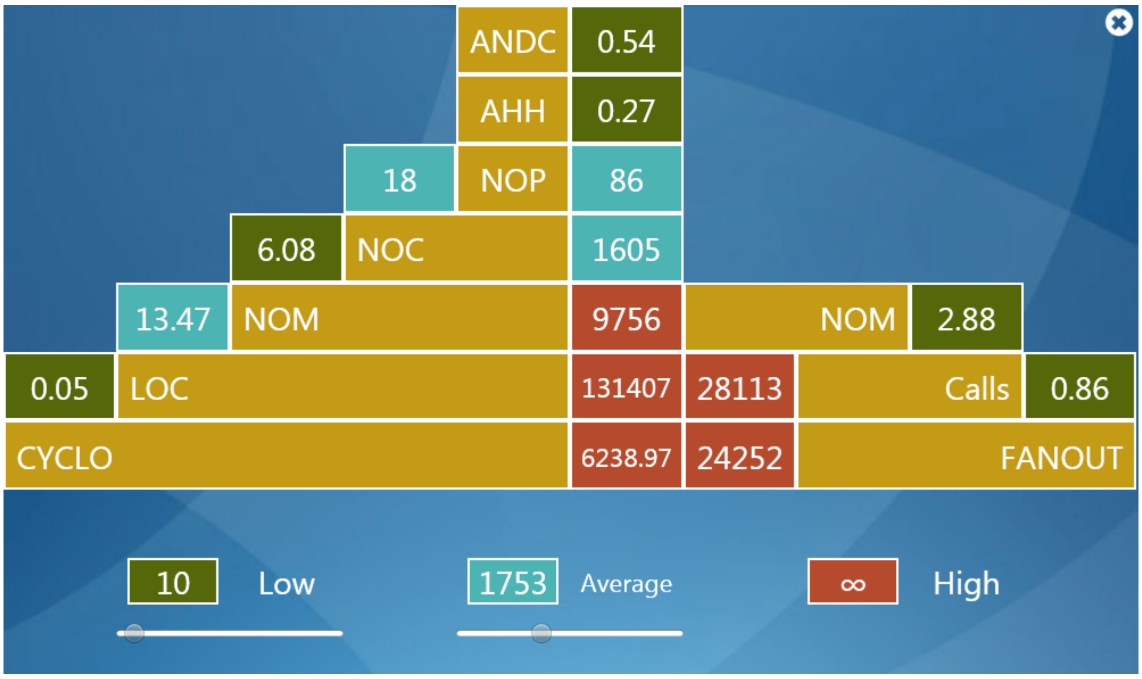


Figure 4.16: *Overview Pyramid* of *ArgoUML*. Green values indicate metrics that are close to the low statistical threshold. Blue values indicate that the metric is close to the average statistical threshold and red indicates that the value is close to the high statistical threshold.

Packages Overview

In Chapter 4.2.4 it is described that the *Package View* is particularly used to visualize the scope of a class. This is done by displaying all classes that are in the same package as the actual class. In a similar way the *Packages Overview* can be used to provide the user with an overview about the whole project. In Figure 4.17 on the left side one can see all the packages of the actual project. In this example there are 86 packages. The packages are visualized using circles and as in the example of a single package, the number of lines of code of a package defines the radius of the circle that is used to visualize the package. McCabe's cyclomatic complexity defines the font color of the text in the circle. The hierarchical relationship between the packages are modeled by lines.[4] In Figure 4.17 one can see that the package `org.argouml.uml.diagram.ui` is highlighted. Accordingly all relationships of this package are also highlighted (yellow lines). On the right side of the figure the classes that belong to the highlighted package are visualized. In this example there are 126 classes defined in the package `org.argouml.uml.diagram.ui`.

[4]It is important to note that according to the Java specifications there is no hierarchical relationship between packages [GJSB05]. But since a lot of developers use packages in that way, we decided to follow this principle.

Similar to selecting and highlighting a package it is also possible to set the focus on a class. In Figure 4.17 the focus concentrates on the class `FigNodeModelElement`. Accordingly all its relationships to other classes are also highlighted. In order to provide consistent functionality it is not only possible to use circles to visualize packages and classes but also the previously mentioned house metaphor and a kiviat based diagram. If it is necessary to display a lot of classes in this view, it can become difficult to differentiate individual classes. For example the package `org.argouml.uml.ui.foundation.core` contains 146 classes that have to be displayed. But to overcome this issue, it is possible to filter out classes and packages by name. For example in Figure 4.18 one can see how all classes that do not contain *me* in their names are filtered out. In that way the view becomes way more perspicuous.

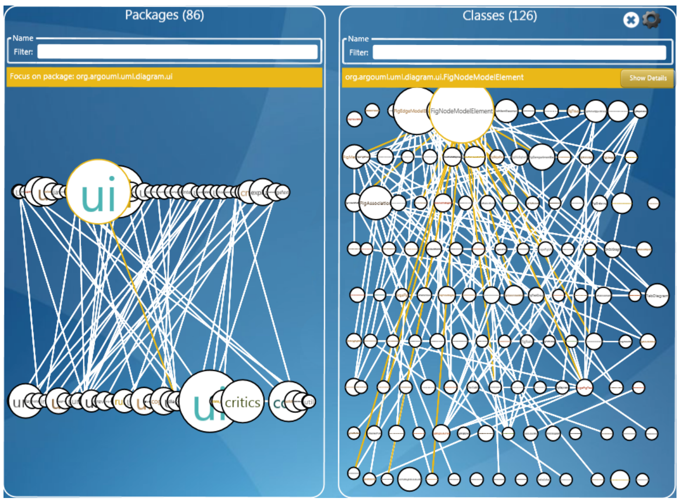

Packages (86)
Name
Filter:
Focus on package: org.argouml.uml.diagram.ui
ui
critics
Classes (126)
Name
Filter:
org.argouml.uml.diagram.ui.FigNodeModelElement
Show Details
FigNodeModelElement

Figure 4.17: *Packages Overview*

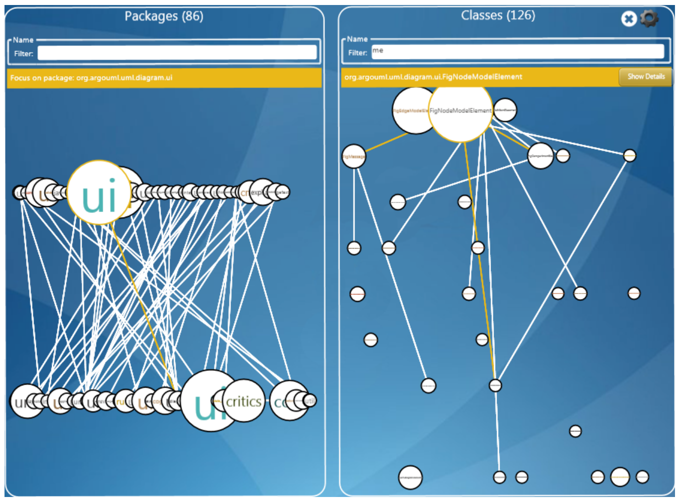
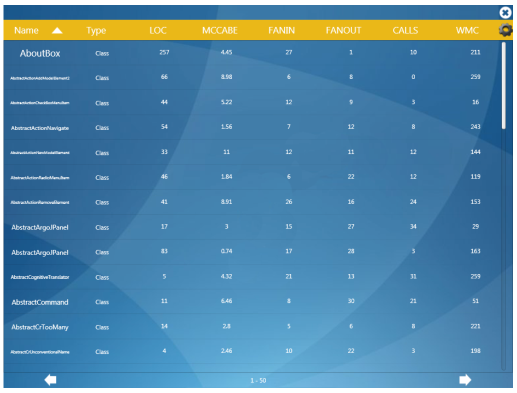


Figure 4.18: *Packages Overview* with enable name filter

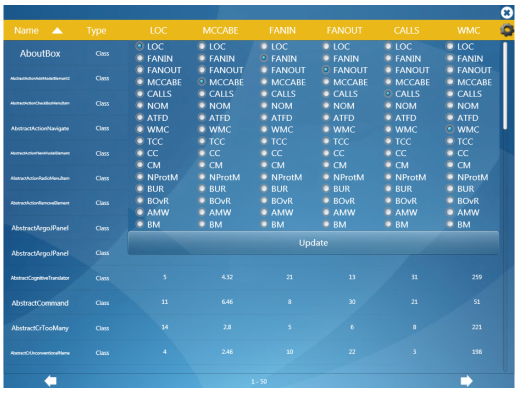
Metrics Overview

The *Package View* does not provide a possibility to find for example the class with the highest cyclomatic complexity. In order to overcome this issue our *SmellTagger* application offers another feature to browse trough all classes and interfaces in the form of a traditional list. This list can be sorted by the name of the entity, the type of the entity or by various code metrics. In the example shown in Figure 4.19 the list is sorted by the names of the entities. In this example the six code metrics *LOC*, *McCabe*, *FANIN*, *FANOUT*, *CALLS* and *WMC* are displayed, but it is possible to customize this view so that other code metrics can be seen as well. This customization tool is shown in Figure 4.20. For each column the user can choose the metric that should be displayed.



Name	Type	LOC	MCCABE	FANIN	FANOUT	CALLS	WMC
AboutBox	Class	257	4.45	27	1	10	211
AbstractActionAddModelElement2	Class	66	8.98	6	8	0	259
AbstractActionCheckBoxMenuItem	Class	44	5.22	12	9	3	16
AbstractActionNavigate	Class	54	1.56	7	12	8	243
AbstractActionNewModelElement	Class	33	11	12	11	12	144
AbstractActionRadioMenuItem	Class	46	1.84	6	22	12	119
AbstractActionRemoveElement	Class	41	8.91	26	16	24	153
AbstractArgoJPanel	Class	17	3	15	27	34	29
AbstractArgoJPanel	Class	83	0.74	17	28	3	163
AbstractCognitiveTranslator	Class	5	4.32	21	13	31	259
AbstractCommand	Class	11	6.46	8	30	21	51
AbstractCrTooMany	Class	14	2.8	5	6	8	221
AbstractCrUnconventionalName	Class	4	2.46	10	22	3	198

Figure 4.19: *Overview List*

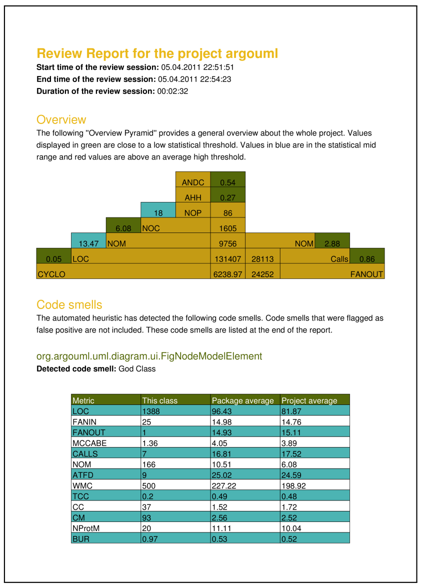


Name	Type	LOC	MCCABE	FANIN	FANOUT	CALLS	WMC
AboutBox	Class						
AbstractActionAddModelElement2	Class						
AbstractActionCheckBoxMenuItem	Class						
AbstractActionNavigate	Class						
AbstractActionNewModelElement	Class						
AbstractActionRadioMenuItem	Class						
AbstractActionRemoveElement	Class						
AbstractArgoJPanel	Class						
AbstractArgoJPanel	Class						
AbstractCognitiveTranslator	Class	5	4.32	21	13	31	259
AbstractCommand	Class	11	6.46	8	30	21	51
AbstractCrTooMany	Class	14	2.8	5	6	8	221
AbstractCrUnconventionalName	Class	4	2.46	10	22	3	198

LOC	MCCABE	FANIN	FANOUT	CALLS	WMC
● LOC	○ LOC	○ LOC	○ LOC	○ LOC	○ LOC
○ FANIN	○ FANIN	● FANIN	○ FANIN	○ FANIN	○ FANIN
○ FANOUT	○ FANOUT	○ FANOUT	● FANOUT	○ FANOUT	○ FANOUT
○ MCCABE	● MCCABE	○ MCCABE	○ MCCABE	○ MCCABE	○ MCCABE
○ CALLS	○ CALLS	○ CALLS	○ CALLS	● CALLS	○ CALLS
○ NOM	○ NOM	○ NOM	○ NOM	○ NOM	○ NOM
○ ATFD	○ ATFD	○ ATFD	○ ATFD	○ ATFD	○ ATFD
○ WMC	○ WMC	○ WMC	○ WMC	○ WMC	● WMC
○ TCC	○ TCC	○ TCC	○ TCC	○ TCC	○ TCC
○ CC	○ CC	○ CC	○ CC	○ CC	○ CC
○ CM	○ CM	○ CM	○ CM	○ CM	○ CM
○ NProtM	○ NProtM	○ NProtM	○ NProtM	○ NProtM	○ NProtM
○ BUR	○ BUR	○ BUR	○ BUR	○ BUR	○ BUR
○ BOvR	○ BOvR	○ BOvR	○ BOvR	○ BOvR	○ BOvR
○ AMW	○ AMW	○ AMW	○ AMW	○ AMW	○ AMW
○ BM	○ BM	○ BM	○ BM	○ BM	○ BM

Figure 4.20: *Overview List* customization

Review Report



Review Report for the project argouml
Start time of the review session: 05.04.2011 22:51:51
End time of the review session: 05.04.2011 22:54:23
Duration of the review session: 00:02:32

Overview

The following "Overview Pyramid" provides a general overview about the whole project. Values displayed in green are close to a low statistical threshold. Values in blue are in the statistical mid range and red values are above an average high threshold.

Code smells

The automated heuristic has detected the following code smells. Code smells that were flagged as false positive are not included. These code smells are listed at the end of the report.

org.argouml.uml.diagram.ui.FigNodeModelElement
Detected code smell: God Class

Metric	This class	Package average	Project average
LOC	1388	96.43	81.87
FANIN	25	14.98	14.76
FANOUT	1	14.93	15.11
MCCABE	1.36	4.05	3.89
CALLS	7	16.81	17.52
NOM	166	10.51	6.08
ATFD	9	25.02	24.59
WMC	500	227.22	198.92
TCC	0.2	0.49	0.48
CC	37	1.52	1.72
CM	93	2.56	2.52
NProtM	20	11.11	10.04
BUR	0.97	0.53	0.52

Figure 4.21: First page of a review report

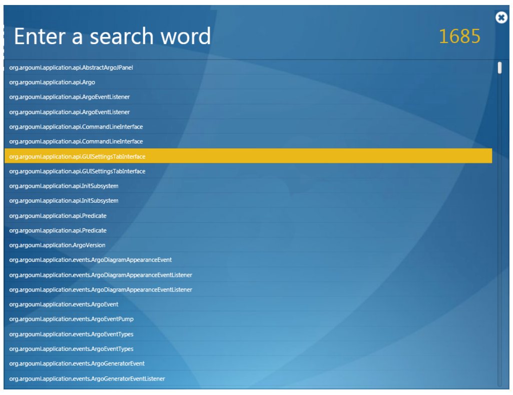
In order to provide the user with a possibility to store the insights of a review session, our *Smell-Tagger* application allows users to generate a review report. This review report can be generated at any time during the review session by placing a specific tagged object on the screen of the Surface unit (Chapter 4.2.5). The information that is contained in a review report covers the detected code smells including its notes, audio notes, tags and annotated code snippets as well as a list of metrics. Additionally, the code smells that were flagged false positive by the reviewers are mentioned in the report too, in order to keep the process traceable. Figure 4.21 shows the first page of a review report. One can see that the *Overview Pyramid* defined by Lanza and Marinescu is also included in the review report.

The review report can be used as a starting point for a refactoring that is performed after the review session in order to resolve the design flaws that are listed in the review report.

Search Function

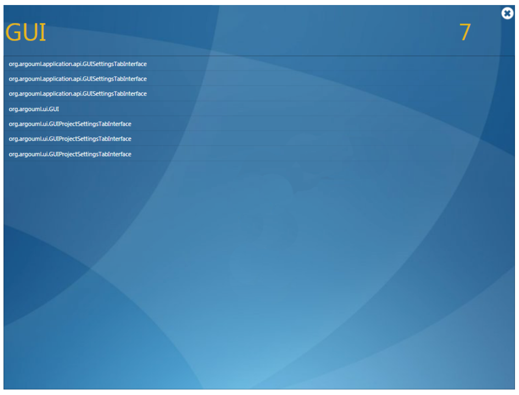
During the evaluation described in Chapter 6, one-fifth of all participants suggested to implement a search feature. Therefore we implemented this feature after the evaluation. It offers a search functionality that allows the users of the application to browse trough a list of all classes in the project and filter this list by class names. In that way it is possible to search for a specific class by its name. Figure 4.22 shows this class list. In this example no name filter is enabled and that is why all of the 1685 classes of the *ArgoUML* project are displayed. In Figure 4.23 one can see that the name filter is enabled and only classes with the keyword *GUI* in their names are displayed. In this example this specification is fulfilled by only seven classes.

This feature improves the usability of our *SmellTagger* application to a great extent. In order to find a specific class it is no longer necessary to know the package in which the class is defined in and then use the *Package Overview* to navigate to this package and search the specific class in the list of all classes that are defined in this package. Instead, it is now possible to directly navigate to the search function and use the class name as the search keyword.


Enter a search word 1685

org.argouml.application.api.AbstractArgoJPanel
org.argouml.application.api.Argo
org.argouml.application.api.ArgoEventListener
org.argouml.application.api.ArgoEventListener
org.argouml.application.api.CommandLineInterface
org.argouml.application.api.CommandLineInterface
org.argouml.application.api.GUISettingsTabInterface
org.argouml.application.api.GUISettingsTabInterface
org.argouml.application.api.InitSubsystem
org.argouml.application.api.InitSubsystem
org.argouml.application.api.Predicate
org.argouml.application.api.Predicate
org.argouml.application.ArgoVersion
org.argouml.application.events.ArgoDiagramAppearanceEvent
org.argouml.application.events.ArgoDiagramAppearanceEventListener
org.argouml.application.events.ArgoDiagramAppearanceEventListener
org.argouml.application.events.ArgoEvent
org.argouml.application.events.ArgoEventPump
org.argouml.application.events.ArgoEventTypes
org.argouml.application.events.ArgoEventTypes
org.argouml.application.events.ArgoGeneratorEvent
org.argouml.application.events.ArgoGeneratorEventListener

Figure 4.22: *Search Function View*

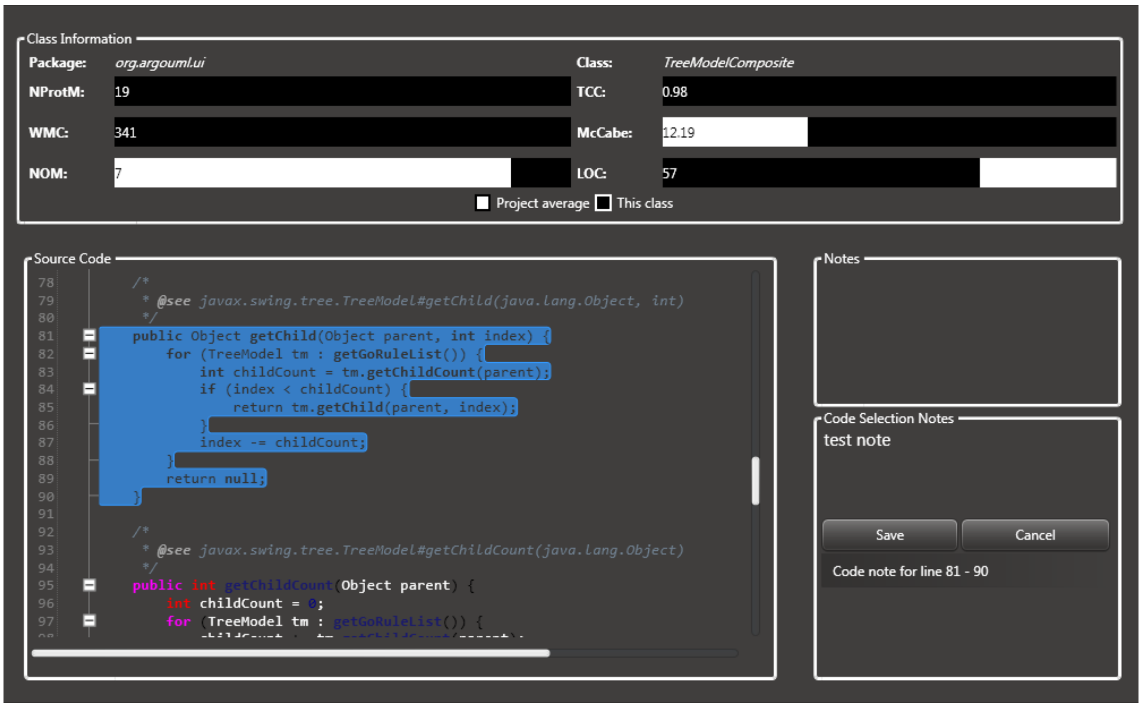

GUI 7

org.argouml.application.api.GUISettingsTabInterface
org.argouml.application.api.GUISettingsTabInterface
org.argouml.application.api.GUISettingsTabInterface
org.argouml.ui.GUI
org.argouml.ui.GUIProjectSettingsTabInterface
org.argouml.ui.GUIProjectSettingsTabInterface
org.argouml.ui.GUIProjectSettingsTabInterface

Figure 4.23: *Search Function View* with enabled name filter

Tag Source Code

Another suggestion of improvement that was frequently mentioned by the participants of the evaluation (Chapter 6) was the idea to not only allow to add a note for a class or a method but also for a code snippet. In order to enable this feature it is necessary that the user can mark a part of the source code. We implemented this function after the evaluation. An example is shown in Figure 4.24 where the method `TreeModelComposite#getChild(Object parent, int index)` is marked. An example of a note is also shown in this figure. It is possible to store those notes and add them with the corresponding source code snippet to the review report. But not only whole methods can be marked and annotated, rather it is possible to mark each single line of the source code.


Class Information

Package:	*org.argouml.ui*	**Class:**	*TreeModelComposite*
NProtM:	19	**TCC:**	0.98
WMC:	341	**McCabe:**	12.19
NOM:	7	**LOC:**	57

Project average This class

Source Code

```
78      /*
79       * @see javax.swing.tree.TreeModel#getChild(java.lang.Object, int)
80       */
81      public Object getChild(Object parent, int index) {
82          for (TreeModel tm : getGoRuleList()) {
83              int childCount = tm.getChildCount(parent);
84              if (index < childCount) {
85                  return tm.getChild(parent, index);
86              }
87              index -= childCount;
88          }
89          return null;
90      }
91
92      /*
93       * @see javax.swing.tree.TreeModel#getChildCount(java.lang.Object)
94       */
95      public int getChildCount(Object parent) {
96          int childCount = 0;
97          for (TreeModel tm : getGoRuleList()) {
```

Notes

Code Selection Notes

test note

Code note for line 81 - 90

Figure 4.24: Marked and annotated source code snippet

4.2.5 User Interaction

There are various ways how the user can interact with our *SmellTagger* application. The most common way is to use the fingers to manipulate elements of the user interface. But there are also other possibilities that can be used. Tagged objects are used to enable special features that do not have to be available all the time. Gestures are particularly used to navigate between different windows.

Tagged Objects

The Microsoft Surface SDK provides the possibility to create so called *Tag Visualizations*.[5] TagVisualizations are user interfaces that appear on the Microsoft Surface screen whenever a user places a tagged object on the screen. Tagged objects are special objects that are *"marked with a special pattern of dots called tags. A tag consists of a geometric arrangement of infrared reflective and absorbing*

[5]http://msdn.microsoft.com/en-us/library/ee804826%28Surface.10%29.aspx

areas" [Libb]. Each of these byte tags stores a unique binary code so that they are distinguishable. Since byte tags store 1 byte of data there are 256 unique tags available.

Figure 4.25 shows an example of a byte tag. The black background is infrared-absorbing so that the white (infrared-reflecting) circles can be recognized by the Microsoft Surface vision system. The big circle in the middle of the tag is used to define the center of the tag and the small three white circles are used to define the orientation of the tag. Additionally each tag can have up to eight data bits that are represented by small white circles that are located around the center of the tag. Figure 4.26 shows an exemplary tag with data bits for the value 0xC6 (198).

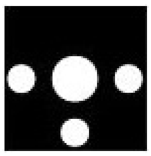


Figure 4.25: Example of a tag

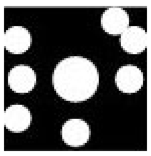


Figure 4.26: Tag 0xC6 (198)

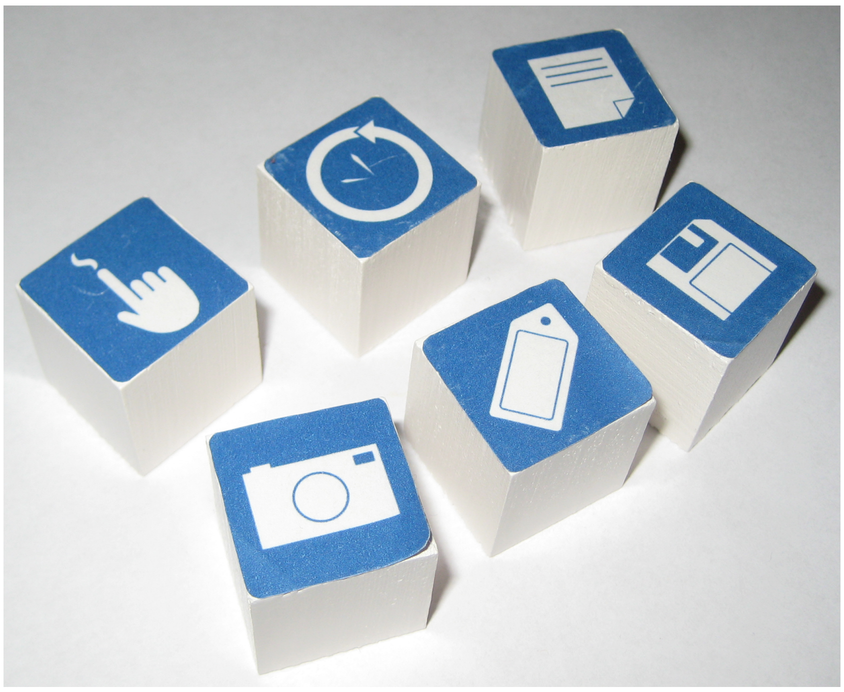


Figure 4.27: The actual tagged objects for our *SmellTagger* application

Tag Visualizations are used in our *SmellTagger* application to provide the user with additional features that have not to be always available but can be used in any situation. Tag Visualizations are most suitable for that since tagged objects are recognized as soon as they are placed on the Microsoft Surface screen, independent of the state of the application or of the user interface elements that are currently visible on the screen. The six tagged objects shown in Table 4.1 are implemented in our *SmellTagger* application. The real tagged objects are shown in Figure 4.27.

🕒	Whenever the user adds a new note (either a written or an audio note) to an entity or tags a class or a method with a keyword, these actions are recorded. Therefore it is possible to undo them. By placing this tagged object on the screen, an user interface appears where all recent actions are listed. From these actions the user can choose which ones he wants to revoke.
💾	In Chapter 5.5 it is described that loading a project from the SOFAS web services can require a lot of time. In order to overcome this issue it is possible to store a project locally. Whenever this tagged object is placed on the screen, the user is asked where he wants to store the actual project. After providing this information the project is stored locally. This feature can also be used to pause and resume a review session.
📷	Whenever this tagged object is placed on the Microsoft Surface screen, a screenshot is taken and stored as an image file in the local file system.
🏷	This tagged object can be used to associate a keyword or a value with a collection of classes and methods. Whenever it is placed on the screen, a container is displayed. This container holds all the classes and methods that are currently associated with this tagged object. By using drag and drop it is possible to add or remove items from this container.
📄	In order to provide the user with a possibility to store the results of a review process it is possible to generate a review report that lists all findings of the current review session. These findings include a list of the detected code smells with notes, audio notes, code notes, keywords and metrics. It also mentions those code smells that were identified as false positive results. Whenever this tagged object is placed on the screen such a report is generated and stored as a PDF file. If the users wants, he can examine the report directly on the Microsoft Surface.
👆	Gestures can be used to control the application. If this tagged object is placed on the screen, the application switches to a special mode that allows to use gestures. How this exactly works is explained in Chapter 5.1.

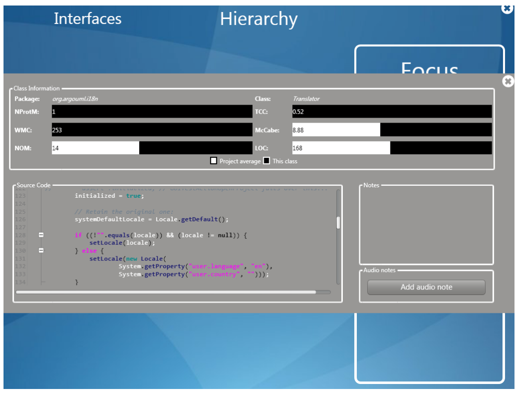
Table 4.1: Tagged objects that can be used in our *SmellTagger* application

Gestures

Gestures are particularly used to manipulate views. It is possible to use gestures to close windows or to display the settings options of a window. But the scope of operation is not limited to this two examples. In Chapter 5.1 it is described in detail how these gestures are implemented in our *SmellTagger* application and how it is possible to add auxiliary gestures.

Other Remarks about the Interaction with the Application

Interaction with our *SmellTagger* application is straight forward. Almost every element of the user interface supports the three most common manipulations: scaling, rotating and translating. In that way it is possible to freely arrange the UI so that it is best suited for the actual task. In order to support collaborative behavior it is also possible to duplicate interesting content, so that more than one user can view and manipulate it concurrently. For example Figure 4.28 and Figure 4.29 show how this is done when viewing source code. It is possible to duplicate the views so that each reviewer has its own code view to work with. These duplicated views can also be resized, rotated and moved.



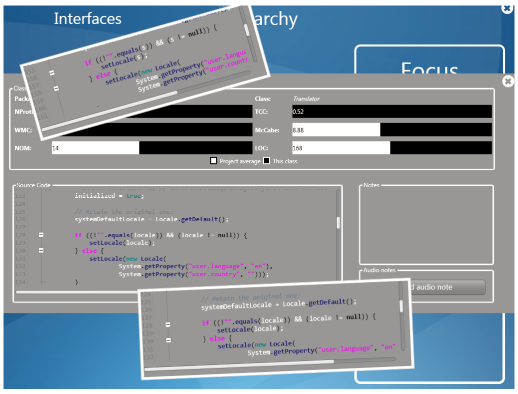
Package: *org.argouml.i18n* Class: *Translator*
NProtM: 1 TCC: 0.52
WMC: 253 McCabe: 8.88
NOM: 14 LOC: 168

```
initialized = true;

// Retain the original one:
systemDefaultLocale = Locale.getDefault();

if ((!"".equals(locale)) && (locale != null)) {
    setLocale(locale);
} else {
    setLocale(new Locale(
            System.getProperty("user.language", "en"),
            System.getProperty("user.country", "")));
}
```

Figure 4.28: *Class Detail View*



Class: *Translator*
TCC: 0.52
McCabe: 8.88
NOM: 14 LOC: 168

```
initialized = true;

// Retain the original one:
systemDefaultLocale = Locale.getDefault();

if ((!"".equals(locale)) && (locale != null)) {
    setLocale(locale);
} else {
    setLocale(new Locale(
            System.getProperty("user.language", "en"),
            System.getProperty("user.country", "")));
}
```

Figure 4.29: *Class Detail View* with two separate code views

4.3 Lesson Learned

Designing a nice looking user interface that is both intuitively easy to learn and offers all functionality the user needs is a hard job. Following well-known design principles mentioned for example by Tufte [Tuf86] or Shneiderman [Shn96] can definitively help. But especially when developing an user interface for a new technology, such as multi-touch interfaces, it is also important to sometimes get rid of established paradigms. For example it is not practical to just rely on buttons and clickable icons as in a traditional desktop application and port this to fit into a multi-touch environment. Instead new ways have to be found to provide the user with the possibility to use natural and habitual finger and hand movements while interacting with the user interface. Ideally the user should be able to interact with digital content in the same way as he is used to interact with real world content.

Some approaches to achieve this were presented in this chapter. The possibility to capture audio notes provides the user with the ability to interact very naturally with the application, since speaking is one of the most common communication forms. Additionally it is possible to rotate, scale and translate almost every element of the user interface. Thereby the user can always adjust every element so that it optimally fits into his field of vision. Using tagged objects to add classes or methods to a list reminds of using a stamp to mark an object with a unique imprint. Even though not all gestures may follow a real world example (for example the 'X' to close windows), it is a very common behavior to manipulate objects using fingers. This assumption is even more obvious when talking about drag and drop operations. The concept to use fingers to move objects (drag) and place (drop) them somewhere else is widely used in the real world.

In Chapter 2.1 design guidelines for a feasible visualization are described. By comparing our *SmellTagger* application with the design hints mentioned by Shneiderman [Shn96] it becomes evident, that all of them are fulfilled.

- **Overview:** Shneiderman states that every visualization should provide the user with a general overview about the data it represents. Our *SmellTagger* application offers a general overview about all detected code smells. In Chapter 4.2.3 the *Main Window* is described that visualizes all code smells that were found in the analyzed project. From there the user can use various features to examine those code smells in detail.

- **Zoom:** It should be possible to examine interesting data in detail. In this application, a lot of views described in Chapter 4.2.4 provide the user with the possibility to examine a class or a method in detail. For example the *Dependency View* or the *Hierarchy View* implement a function that allows users to use drag and drop operations to place a class or a method on a specially marked area on the screen in order to display a lot of detail information about the entity.

- **Details on demand:** Shneiderman states that not all details should be displayed in the first place to avoid confusing because too much information is presented at the same time. Therefore our *SmellTagger* application uses a clean user interface with no information overload. In the first place, when the application is started, the user sees the set of detected code smells and nothing else. Details about those code smells are only displayed if the user explicitly requests it.

- **Relate:** According to Shneiderman, relationships between elements should be represented in the visualization. Our *SmellTagger* application obeys this principle and visualizes the hierarchical relationships between packages as well as the call dependencies between classes.

- **History:** Sneiderman states that it should be possible to record user actions and ideally implement a possibility to undo them. In our *SmellTagger* application, important events such as adding a note to a class or method, or recording an audio note are stored. Using a special tagged object described in Chapter 4.2.5 it is possible to examine the recent history of those events. But it is not only possible to browse through these events but also to undo them.

- **Extract:** According to Shneiderman, it should be possible to extract information from the visualization in some way. There are generally two possibilities to extract information out of our *SmellTagger* application. First, it is possible to store the whole project locally in a XML file. In that way it is not necessary to download all the information from the web service, if the analysis is started again another time. This functionality can also be used to pause and resume a review session. Second, it is possible to generate a review report as described in Chapter 4.2.4. This review report summarizes the most important information about the findings during a review session.

But not only Shneiderman's design hints are fulfilled by our *SmellTagger* application, but also Tufte's design guidelines. As described in Chapter 2.1, Tufte states that a visualization should help the viewer to think about the information rather than about the methodology and the design. This can be achieved by removing redundant data and unnecessary design elements. The evaluation of our *SmellTagger* application, described in Chapter 6, has shown that the application provides a clear user interface with no information overload. As a consequence the important data and features are generally easy to find.

As another advice, Tufte suggests to use *"multi-functioning graphical elements"* to display multivariate data in a small space. Using kiviat diagrams to display multiple code metrics from two or more different data sources obeys this principle. The *Package View* described in Chapter 4.2.4 is another example of a component of our *SmellTagger* application that is directly influenced by one of Tufte's design principles. The Package View encourages the viewer's eye to directly compare classes respectively packages with each other.

In the next chapter it is described how some of the features presented in this chapter are actually implemented in our *SmellTagger* application.

Chapter 5

Implementation Details

In this chapter, three implementation details are described. First the algorithm that is used to recognize gestures is presented, second the rule-based detection strategies used to detect code smells are described in detail. Third, it is described how syntax highlighting and code folding are implemented into the application's code viewer.

5.1 Gestures

Multi-touch interfaces provide a lot of possibilities for various kinds of user interactions. Built-in gestures allow users to manipulate elements of the user interface and to control the application. Therefore we also implemented gestures in our *SmellTagger* application. Currently the three gestures defined in Figure 5.1, Figure 5.2 and Figure 5.3 are supported. White these gestures it is possible to navigate and control the application. The *Delete gesture* (Figure 5.1) can be used to close windows. With the *OK gesture* (Figure 5.2) it is possible to leave the mode in which gestures can be used and switch back to the application. The *Settings gesture* shown in Figure 5.3 is used to access the settings menu of a window if there is one available. In order to make the gestures more customizable, it is possible to define auxiliary gesture templates in XML files. For that purpose a XML schema (Listing 5.1) is defined that can be used to implement additional user-defined gestures.

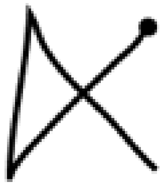


Figure 5.1: Delete gesture

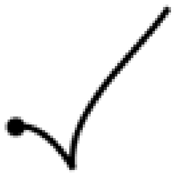


Figure 5.2: OK gesture

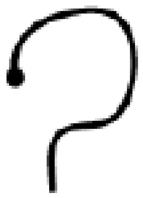


Figure 5.3: Settings gesture

Each element of the user interface that has to be manipulatable by gestures has to implement the interface `Manipulatable` shown in Figure 5.4. In that way it can be assured that only those elements of the user interface are affected by gestures that were capable of handling such events. Additionally the interface defines the methods that have to be implemented by every element of the user interface that is affected by gestures.

Listing 5.1: XML schema for gestures

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
   elementFormDefault="qualified"
   xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="points">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="point">
          <xs:complexType>
            <xs:attribute name="x" type="xs:unsignedShort"
                  use="required" />
            <xs:attribute name="y" type="xs:unsignedShort"
                  use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

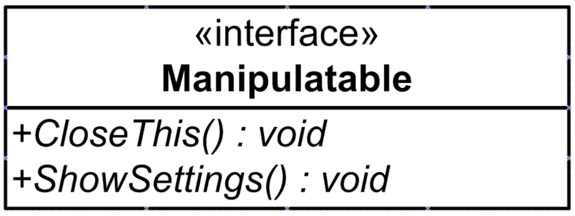


Figure 5.4: `Manipulatable` interface

In order to use custom gestures in an application it is necessary to implement a component that is able to recognize gestures. Implementing a gesture recognizer is not a trivial task. Wobbrock *et al.* discover a substantial problem: *"Although mobile, table, large display, and tabletop computers increasingly present opportunities for using pen, finger and wand gestures in user interfaces, implementing gesture recognition largely has been the privilege of pattern matching experts, not user interface prototypers"* [WWL07]. To overcome this issue Wobbrock *et al.* presented a simple gesture recognition algorithm [WWL07]. This algorithm assumes that a list of points, the so-called point path, that contains all points that build a gesture, is available. Then the algorithm consists of four simple steps:

1. **Resample the point path**
   Since the moving speed of a gesture has a substantial influence on the number of sample points, it is necessary to first resample the point path to achieve an evenly distributed sample. Otherwise it would not be possible to compare gestures with different movement speeds. For that reason the M original path points are sampled into N new points that have the same distance to each other.

2. **Rotate sample around a indicative angle**
   "Template matching techniques are sensitive to orientation. Therefore, for rotation invariant recognition it is necessary to first rotate the patterns into the same orientation" [Kar04]. To find this rotation angle that rotates the patterns into the same orientation one could use a brute force method and rotating the gestures by 1° for 360°. In order to save time, Wobbrock *et al.* rotate the candidate gestures before trying to match them with a template gesture. The rotation angle that is used in Wobbrock's approach, the so-called *indicative angle*, is the angle between the center of the gesture and the gesture's first point. The gesture is rotated in a way that this angle is set to 0°. Wobbrock *et al.* argue that this approximates finding the best angular match between a template and a candidate gesture and will therefore speed-up the posterior matching process described in step 4.

3. **Scale and translate**
   In order to facilitate the matching process it is also necessary that both the gesture template and the gesture candidate have approximately the same size and position. For that reason, the gesture is scaled to a predefined reference square and translated to a predefined reference point.

4. **Find the optimal angle for the best match**
   In this step the actual recognition is done. Using the average distance d_i, defined in Equation 5.1, it is possible to compare a candidate gesture C to each stored template gesture T_i.

   $$d_i = \frac{\sum_{k=1}^{N} \sqrt{(C[k]_x - T_i[k]_x)^2 + (C[k]_y - T_i[k]_y)^2}}{N} \tag{5.1}$$

   This comparison has to be done using the best angular match between the candidate gesture and the template gesture. In step 2, the *indicative angle* was used to find an approximation, but nevertheless it is necessary to search the whole angle space. The *indicative angle* just provides a good starting point. As soon as this best match is found, the minimum average path distance can be calculated. Afterwards this value is mapped into the [0,1] range. In that way the minimum average path distance for each gesture template is calculated. The template with the smallest average path distance matches best with the candidate gesture. Using statistical thresholds it can be verified if the best match is good enough to be a valid recognition.

   In order to find the best angular match between a candidate and a template gesture various search algorithms, for example hill-climbing, can be used. However, for performance reasons, Wobbrock *et al.* use a *Golden Section Search* algorithm, initially described by Press *et al.* [PTVF92].

The implementation in our *SmellTagger* application is based on the source code of Marshall [Mar] who has already implemented the approach of Wobbrock *et al.* on a Windows 7 based HP TouchSmart. Although studies [WWL07] have shown that this approach is working correctly, it has limitations. For example the algorithm does not support interactive correction or multistroke gestures [AW10].

5.2 Rule-based Detection Strategies

Lanza and Marinescu [LM10] proposed an approach that uses detection strategies to find code smells in a software project. This approach is used in our *SmellTagger* application to detect four well-known code smells in an arbitrary software project: *God Class*, *Refused Parent Bequest*, *Shotgun Surgery*, and *Brain Class*. In order to make the application more customizable the rules in a detection strategy can be defined using XML files. In that way it is easy to add new rules, manipulate already existing rules or delete rules. The XML schema that has to be used for these XML rule files is shown in Figure 5.6.

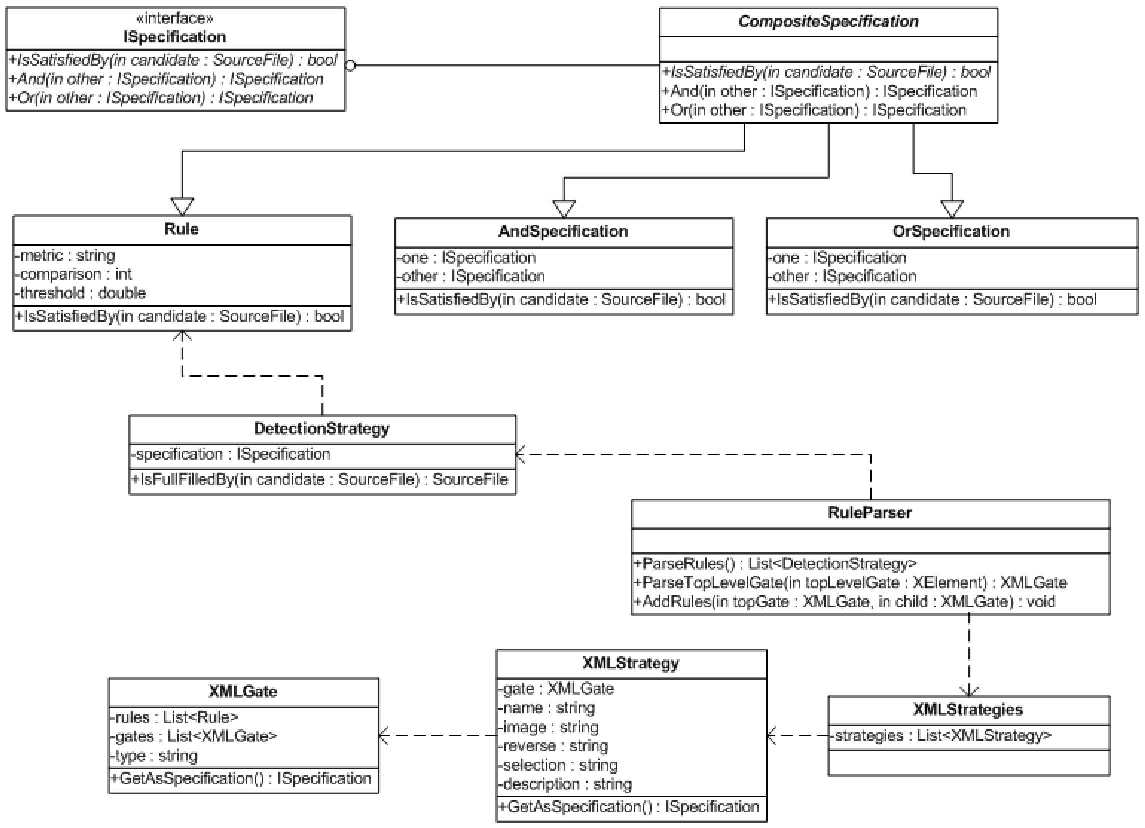

«interface»
ISpecification
+IsSatisfiedBy(in candidate : SourceFile) : bool
+And(in other : ISpecification) : ISpecification
+Or(in other : ISpecification) : ISpecification
CompositeSpecification
+IsSatisfiedBy(in candidate : SourceFile) : bool
+And(in other : ISpecification) : ISpecification
+Or(in other : ISpecification) : ISpecification
Rule
-metric : string
-comparison : int
-threshold : double
+IsSatisfiedBy(in candidate : SourceFile) : bool
AndSpecification
-one : ISpecification
-other : ISpecification
+IsSatisfiedBy(in candidate : SourceFile) : bool
OrSpecification
-one : ISpecification
-other : ISpecification
+IsSatisfiedBy(in candidate : SourceFile) : bool
DetectionStrategy
-specification : ISpecification
+IsFullFilledBy(in candidate : SourceFile) : SourceFile
RuleParser
+ParseRules() : List<DetectionStrategy>
+ParseTopLevelGate(in topLevelGate : XElement) : XMLGate
+AddRules(in topGate : XMLGate, in child : XMLGate) : void
XMLGate
-rules : List<Rule>
-gates : List<XMLGate>
-type : string
+GetAsSpecification() : ISpecification
XMLStrategy
-gate : XMLGate
-name : string
-image : string
-reverse : string
-selection : string
-description : string
+GetAsSpecification() : ISpecification
XMLStrategies
-strategies : List<XMLStrategy>

Figure 5.5: UML class diagram of the Specification Pattern

All XML rule files are parsed during the start-up phase of the application. In order to verify if a class of the analyzed project fulfills the specifications defined in the XML rule files and is therefore a code smell candidate, it is necessary to check each rule against each class in the project. In order to solve this task the *Specification pattern*, first presented by Evans and Fowler [EF97], is used. This design pattern allows to specify complex boolean logic conditions and making them explicit in the model. Berther states that *"its primary use is to select a subset of objects based on some criteria"* [Ber]. In the same way it is used in our *SmellTagger* application. Based on the criteria defined in the XML rule files, a subset of classes is selected that are possible code smells. Figure 5.5 shows an UML class diagram of that part of the code that is responsible for parsing the XML rule files and verifying if a source code file meets the specifications in these rule files.

The class `RuleParser` parses the XML file that contains all the detection strategies and creates `XMLGate` and `XMLStrategy` objects. The class `XMLStrategy` holds all the information that was parsed out of the XML file for one detection strategy. A `XMLGate` object is created for each composed logical condition in a detection strategy. These two objects are used to cre-

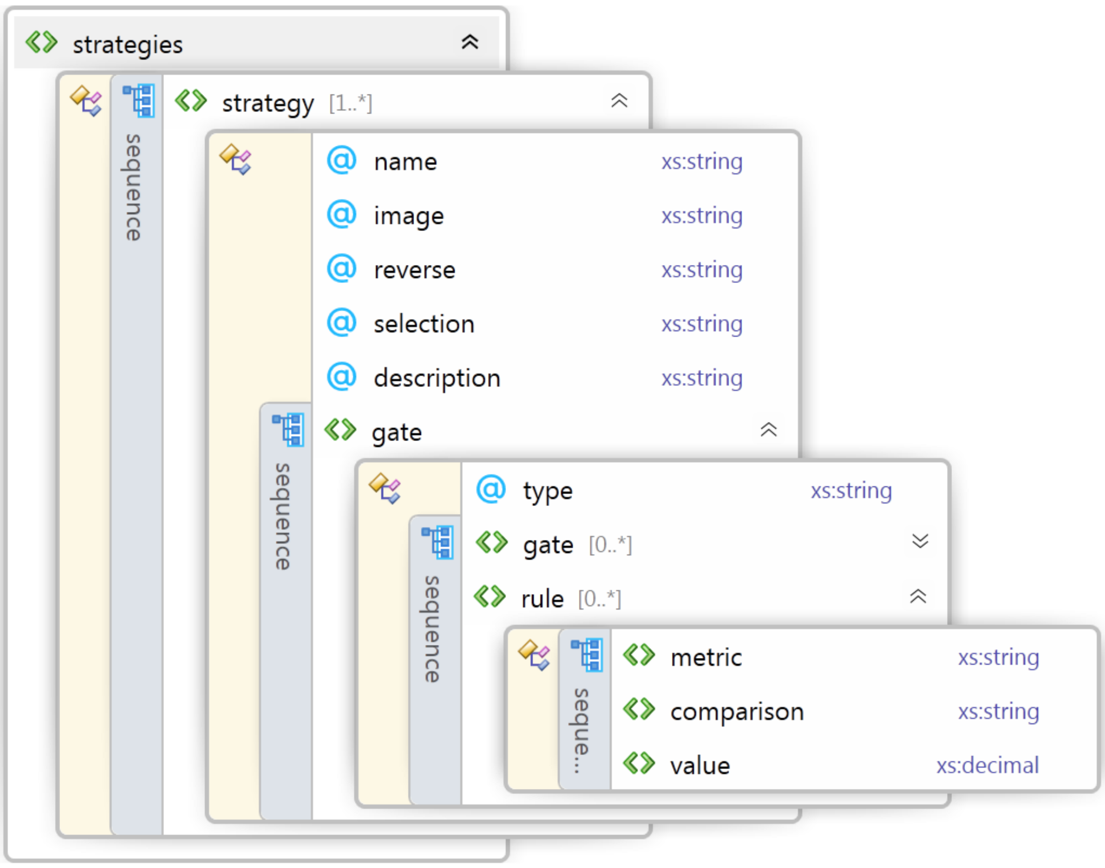


Figure 5.6: XML schema for customizable detection strategies

ate a `DetectionStrategy` object that has a reference to an `ISpecification` object. This `ISpecification` contains all the rules that have to be fulfilled by the corresponding detection strategy. A rule is defined in the class `Rule` and consists of a metric, a threshold and a comparison symbol. Whenever is is necessary to check if a class of the analyzed project fulfills a specific detection strategy, it can be checked by using the method `DetectionStrategy#IsFullFilledBy( SourceFile candidate)`. This method checks if all rules that are contained in this detection strategy are fulfilled by the candidate class. During the start-up phase of the application each class is checked against each detection strategy. The detected code smell candidates are then presented to the user as described in Chapter 4.2.3.

The advantages in using the specification pattern are the high flexibility even though there are not a lot of classes necessary to implement the pattern. Adding new conditions is very easy, because it does not require to modify the object that is checked itself. As an additional benefit the specification pattern supports logical conditions so that it is very easy to map a detection strategy that consists of composed logical conditions to objects in a specification pattern.

The following section describes how the detection strategies for the four code smells that are predefined in our *SmellTagger* application (*God Class*, *Refused Parent Bequest*, *Shotgun Surgery* and *Brain Class*) are implemented.

5.2.1 God Class

A *God Class* is a class that knows too much and does too much. This violates the object-oriented design principle that every class should only have one responsibility (*Single Responsibility Principle* [Mar02]). Often, a God Class is also surrounded by a *Data Class*[1] whose data containers are used by the God Class. In most cases, a God Class is created accidentally by adding more and more functionality to a class while the software project evolves. However, not all big classes that have a lot of functionalities are bad per se. In some cases, for example a parser, it is very difficult to decompose a big class into smaller units [Tri05]. In any cases, God Classes have some major drawbacks. Because of their size, they tend to be very difficult to understand. Additionally, because of their complexity and the amount of functionality they provide, God Classes will be changed more frequently than other classes during maintenance and refactoring processes. This increases the likelihood of implementing defects in the code. For these reasons it is important to find potential God Classes in a software system and try to refactor them.

According to Lanza and Marinescu [LM10] a God Class has typically three main characteristics that can be easily measured by using code metrics. These main characteristics are shown in Table 5.1. From these characteristics and the metrics that are able to measure them, a detection strategy can be inferred and implemented in a XML structure. From the description of the symptoms of a God Class it can be assumed that all three characteristics apply at the same time. The XML code of this detection strategy is shown in Listing 5.2. As one can see in the listing, each class that access more than eight attributes from unrelated classes and that has a weighted method count above 397 and that has a tight class cohesion below 0.33 is a God Class candidate. These values are the default thresholds that are used in our *SmellTagger* application. The thresholds have to be adapted before every review session.

Listing 5.2: XML detection strategy for a *God Class*

```
<strategy name="God Class" ...>
   <gate type="AND">
     <rule>
       <metric>ATFD</metric>
       <comparison>greater than</comparison>
       <value>8</value>
     </rule>
     <rule>
       <metric>WMC</metric>
       <comparison>greater than</comparison>
       <value>397</value>
     </rule>
     <rule>
       <metric>TCC</metric>
       <comparison>smaller than</comparison>
       <value>0.33</value>
     </rule>
   </gate>
  </strategy>
```

[1]Data Classes are classes that passively store data and no behavior.

Main characteristic	Code metric
God Classes heavily access data of other simpler classes, either directly or using accessor methods.	**ATFD (Access To Foreign Data):** The number of attributes from unrelated classes that are accessed directly or by invoking accessor methods.
God Classes are large and complex.	**WMC (Weighted Method Count):** The sum of the statical complexity of all methods of a class. McCabe's cyclomatic complexity (CYCLO) metric is used to quantify the method's complexity.
God Classes have a lot of non-communicative behavior, i.e., there is low cohesion between the methods belonging to that class.	**TCC (Tight Class Cohesion:)** The relative number of method pairs of a class that access in common at least one attribute of the measured class.

Table 5.1: *God Class'* main characteristics and code metrics to measure them

5.2.2 Refused Parent Bequest

The *Refused Parent Bequest* code smell addresses wrong inheritance relationships in a software project. In a well designed hierarchy, subclasses implement functionality and data of their base classes. Lanza and Marinescu summarize this problem: *"But if a child class refuses to use this special bequest prepared by its parent then this is a sign that something is wrong within that classification relation"* [LM10].

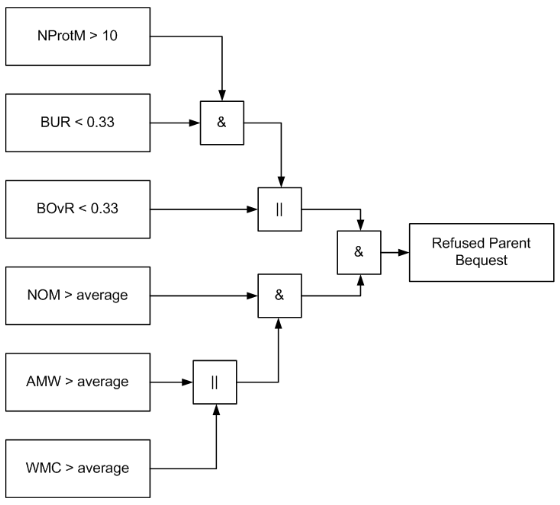


Figure 5.7: Detection strategy for a *Refused Parent Bequest* code smell

Trifu describes two different types of this code smell: *"the one that concerns inherited functionality and the more critical case of refused interface, when a subclass privately inherits interface methods, or overrides them with empty methods"* [Tri05]. The main problem of this code smell is that it introduces code duplications and causes incoherent and non-cohesive class interfaces. According to Lanza and Marinescu, this code smell has two main characteristics that can be detected by analyzing the metrics shown in Table 5.2 [LM10]. It is possible to infer a detection strategy from this description. But it is important to note that, in contrast to the God Class detection strategy, not all characteristics have to be fulfilled at the same time. The result of this observation is that not all rules are conjuncted using logical AND-conjunction. For example, as one can see in Figure 5.7 the rules *AMW* > *average* and *WMC* > *average* are conjuncted using an OR-conjunction.

Main characteristic	**Code metric**
Child class ignores bequest	
Parent class provides more than a few protected members.	**NProtM (Number of Protected Members):** The number of protected methods and attributes of a class.
Child class uses only little of parent's bequest.	**BUR (Base Class Usage Ratio):** The number of inheritance-specific members used by the measured class, divided by the total number of inheritance-specific members from the base class.
Overriding methods are rare in child class.	**BOvR (Base Class Overriding Ratio):** The number of methods of the measured class that override methods from the base class, divided by the total number of methods in the measured class.
Child class is not too small and simple	
Functional complexity is above average.	**AMW (Average Method Weight):** The average static complexity of all methods in a class. McCabe's cyclomatic number is used to quantify the method's complexity.
Class complexity is not lower than average.	**WMC (Weighted Method Count):** The sum of the statical complexity of all methods of a class. McCabe's cyclomatic complexity metric is used to quantify the method's complexity.
Class size is above average.	**NOM (Number of Methods):** The number of methods of a class.

Table 5.2: *Refused Parent Bequest* symptoms and code metrics to measure them

5.2.3 Shotgun Surgery

A *Shotgun Surgery* is a typical anti-pattern in object-oriented software design. A method that is affected by a Shotgun Surgery has many other classes and methods depending on it. This causes the major problem that a change on that method requires many other changes in a lot of different classes and methods. That is why this code smell is *"an indicator of excessive low-level couplings, in the sense of strong afferent (incoming) coupling"* [OCBZ09]. As a result this code smell can cause maintenance problems, since when changing a method affected by Shotgun Surgery, it is necessary to check thoroughly if all affected operations and classes are changed correctly. But fortunately, a Shotgun Surgery is relatively easy to detect. Lanza and Marinescu [LM10] suggest to first search all methods that have a strong change impact and then from these methods those with a high dispersion of changes are probably affected by Shotgun Surgery. This approach can be implemented by using the characteristics and metrics described in Table 5.3. The XML detection strategy that is build based on these metrics is shown in Listing 5.3.

Main characteristic	Code metric
Operation is called by too many other operations.	**CM (Changing Methods):** The number of distinct methods that call the measured method.
Incoming calls are from many different classes.	**CC (Changing Classes):** The number of classes in which the methods that call the measured method are defined in.

Table 5.3: *Shotgun Surgery* symptoms and code metrics to measure these symptoms

Listing 5.3: XML detection strategy for a *Shotgun Surgery*

```
<strategy name="Shotgun Surgery" ... >
   <gate type="AND">
     <rule>
       <metric>CM</metric>
       <comparison>greater than</comparison>
       <value>10</value>
     </rule>
     <rule>
       <metric>CC</metric>
       <comparison>greater than</comparison>
       <value>100</value>
     </rule>
   </gate>
 </strategy>
```

5.2.4 Brain Class

Brain Classes and God Classes have some very similar characteristics. Similar to God Classes, a Brain Class tends to be very complex and combine a lot of different functionalities. But since Brain Classes do not use as much data from foreign classes as a God Class, Brain Classes are slightly more cohesive [OCS10]. Nevertheless the impact of a Brain Class on a software system can be significant. For example, Brain Classes tend to ignore the basic object-oriented principle that every class should have one single responsibility. Additionally Brain Classes tend to violate the *Law of Demeter* [LHR88] which states that a class should not be coupled to closely with other classes. However this violation is less pronounced compared to God Classes since Brain Classes typically access less foreign data directly.

From these observations a detection strategy for Brain Classes can be derived. Lanza and Marinescu use another code smell, called *Brain Method* in order to detect Brain Classes. A Brain Method is a method that *"tend to centralize the functionality of a class, in the same way as a God Class centralizes the functionality of an entire subsystem, or sometimes even a whole system"* [LM10]. This leads to the assumption that a Brain Class can be detected by examining all classes that are very complex and not cohesive and that contain either one *Brain Method* and are extremely large and complex, or contain more than one *Brain Method* and are just very large. The metrics used to measure the complexity, size and cohesion of a potential Brain Class are described in further details in

Table 5.4. Based on this description, our *SmellTagger* application implements a specific detection strategy that is shown in Figure 5.8.

Main characteristic	**Code metric**
Class contains more than one Brain Method and is very large	
Class contains more than one Brain Method.	**BM (Brain Methods):** The number of detected Brain Methods in a class.
The total size of methods in class is very high.	**LOC (Lines of Code):** The number of lines of code of an operation.
Class contains only one Brain Method but is extremely large and complex.	
Class contains only one Brain Method.	**BM (Brain Methods):** The number of detected Brain Methods in a class.
The total size of methods in class is extremely high.	**LOC (Lines of Code):** The number of lines of code of an operation.
The functional complexity of a class is extremely high.	**WMC (Weighted Method Count):** The sum of the statical complexity of all methods of a class.
Class is very complex and non-cohesive	
The functional complexity of a class is very high.	**WMC (Weighted Method Count):** The sum of the statical complexity of all methods of a class.
Class cohesion is low.	**TCC (Tight Class Cohesion):** The relative number of method pairs of a class that access in common at least one attribute of the measured class.

Table 5.4: *Brain Class* symptoms and code metrics to measure these symptoms

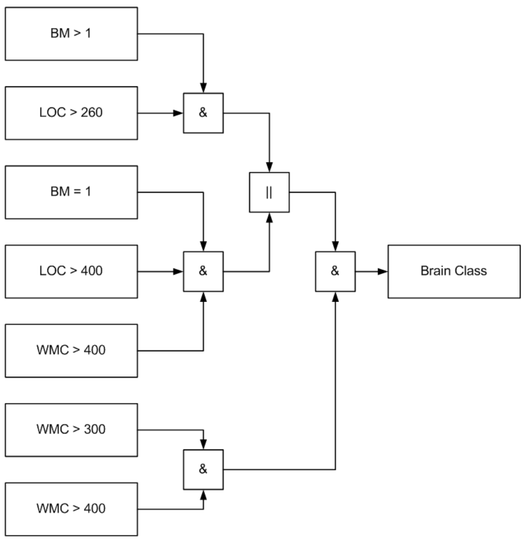


Figure 5.8: Detection strategy for a *Brain Class*

5.3 Syntax Highlighting and Code Folding

Chapter 4.2.4 describes that the views in our *SmellTagger* application that display source code are able to support syntax highlighting and code folding. It is also possible to select and mark a part of the source code using finger touches. In order to enable these features, we modified *Avalon Edit*,[2] the WPF-based text editor of *SharpDevelop 4*,[3] so that it can also be used in a multi-touch environment. These modifications particularly include changing the event handling in order to enable recognizing events from a multi-touch surface. Therefore the original event handlers that register mouse clicks or keyboard inputs are removed and new event handlers that react on touch input are added. But it is also necessary to adapt other parts of the text editor. The original scrollbars are removed and replaced by Surface-enabled scrollbars in order to allow the user to scroll through the source code using a finger gesture. The default code folding strategy that is optimized for displaying XML file is changed so that the code is folded at curly braces. In that way the code folding feature is more useful for displaying Java source code.

Particularly syntax highlighting improves the usability of the source code view because syntax highlighting has the following positive effects:

- Improved readability of source code
- Simpler recognition of structures
- Faster identification of typing errors
- Simpler differentiation between source code and comments

5.4 Validation

In order to ensure the quality of our *SmellTagger* application some tests are implemented and performed on a regular basis. Basically two different kind of tests are implemented. Unit tests to verify the correctness of important and complex functions and tests that verify that the user interface is working correctly.

5.4.1 Unit Testing

The *Visual Studio Unit Testing Framework* is used to generate unit tests for each class of the application's model component. The model component is responsible for representing the real state content [Mar02]. Other components, for example the view component that is responsible for representing the user interface, are tested using different test techniques described in Chapter 5.4.2. With these unit tests it is possible to ensure that the tested operations and classes behave as intended. Per definition unit tests are independent from each other. Therefore it is only possible to show the correctness of individual parts of the whole application.

5.4.2 Interface Testing

In Chapter 3.2 it is mentioned in passing that the Microsoft Surface SDK contains a simulator that replicates the Surface user interface on a workstation. With the help of the *Surface Simulator Au-*

[2]http://wiki.sharpdevelop.net/AvalonEdit.ashx
[3]http://www.icsharpcode.net/OpenSource/SD/

tomation API[4] it is possible to programatically simulate contact input for the Surface. This allows to test an application that is running on the Surface Simulator by using a Windows Forms[5] project. In order to successfully connect the test project to an application that is running on the Surface Simulator, it is necessary that the application is started using the Launcher from the Surface Shell. Listing 5.4 shows how the test project can be connected to the Surface Simulator.

Listing 5.4: Method that connects the test environment to the Surface Simulator

```
private void StartTestButton_Click(object sender, EventArgs e) {

    try {

      /// Gets the active group of SimulatedContact objects.
      /// If this group is null, the tests can not be executed.
      if (SimulatorAutomation.PrimarySimulatedContactGroup == null) {
         throw new CommunicationException();
      }

      /// Blocks until communcation with the Simulator is available.
      if (!SimulatorAutomation.WaitUntilReady()) {
         throw new CommunicationException("Communication Error");
      }

      /// Creates test cases with the active SimulatedContact objects.
      using (PrimarySimulatedContactGroup simulator =
        SimulatorAutomation.PrimarySimulatedContactGroup) {
         SimulatorAutomationAPI apiComm
           = new SimulatorAutomationAPI(simulator);

         /// Executes all test cases.
         StartTest(apiComm);
      }
    }

    /// Informs user about exceptions.
    catch (SystemException se) {
      MessageBox.Show(se.Message, "Simulator Problem",
        MessageBoxButtons.OK);
        this.Close();
    }
}
```

This approach is used to test the main functionalities of the user interface graphically. In that way it is possible to decrease the probability of an error in the user interface.

[4]http://msdn.microsoft.com/en-us/library/ee804820(v=surface.10).aspx
[5]http://msdn.microsoft.com/en-en/library/dd30h2yb.aspx

5.5 Limitations

Chapter 3.4 describes how the SOFAS web service is used to retrieve the necessary information about a software project that should be analyzed. For that purpose SPARQL queries are sent to the RESTful interface of the web service and the response is parsed appropriately. But this kind of information retrieval causes problems. For example in big software projects thousands of SPARQL queries are necessary to just retrieve the project structure. Additional queries are used to get the source code, the relationships between the entities and the code metrics. This leads to the problem that loading the application takes a lot of time. Unfortunately it is not a solution to load part of the information on demand, since almost all of the information is needed during the start-up phase to decide whether a source file is affected by a code smell detection strategy or not. In order to temper this issue it is only required to load this information once. As it is described in Chapter 4.2.5 it is possible to store a project locally. During the next startup, loading a project from a local storage will decrease the time needed to load the project dramatically.

Chapter 6

Evaluation

This chapter describes the evaluation that was performed to show that it is possible to operate our *SmellTagger* application intuitively. Additionally, the evaluation was used to verify that the application can foster the collaboration between reviewers during a code and design review process. First the study settings are described. In the next part the results of the study are presented and its limitations are discussed.

6.1 Study Settings

We performed an evaluation of our *SmellTagger* application to find answers to the following key questions:

- Is the user interface intuitive and easy to use?
- Are users able to solve tasks without a detailed explanation of the functionalities the application provides?
- Can the application be used beneficially to find code smells in a software project?
- Can multi-touch interfaces be used beneficially for doing code reviews?
- Do multi-touch interfaces facilitate collaborative work during a code review?
- Which interaction possibilities with the application are well-accepted by the users?

In order to find answers to these questions, a questionnaire was designed that consists of six parts. In the first part the participants had to solve typical tasks that occur in a review session such as finding the subclasses of a code smell or examining the relationship between a code smell and other classes. After each task the participants had to answer two questions concerning the easiness to find the necessary information and the amount of navigation through different user interface elements that was necessary to solve this task. The second part of the questionnaire consists of some tasks that each participants had to solve. In contrast to the first part, it was not necessary to explicitly answer a question during the task. But at the end of each task the participants were asked again how easy it was to find the necessary functionality to solve the task. The third part consists of ten questions concerning the general impression the participant had when working with the application and his opinion about multi-touch interfaces in general. The participant's opinion about the features of the prototype was addressed in the fourth part. There it was necessary to answer seven questions about different features and functionalities the prototype provides. In the fifth part the participants were asked about their previous experience with multi-touch devices in order to examine if the results of the questionnaire are biased. In a final step during the sixth part, the participants had the possibility to share any other remarks. In particular they were asked about their likes and dislikes while working with the prototype. Each of the questions during the first five parts could be answered on a Likert scale [Lik32]. Therefore the participants had to rate each statement respectively question with values from 1 ("I strongly disagree") to 5 ("I strongly agree"). The questionnaire can be found in Appendix A.

The evaluation was performed with 15 participants. All the participants were graduates with a background in computer science. On average it took 25 minutes to perform the evaluation with one participant, varying from 18 to 40 minutes.

6.2 Results

An overview about all the results of the evaluation can be found in Appendix B. In the following chapters these results are presented in more detail.

6.2.1 Functionalities

The first two parts of the questionnaire address the features of our *SmellTagger* application (Chapter 4.2.4). For each of its main features, the participants had to solve at least one task. In total the participants had to answer 135 questions. All but one participants were able to solve all the tasks so that there are 134 valid answers that can be examined.

Methods

In order to evaluate this feature the participant has to answer two questions. The first question (Table 6.1) addresses the comparison of method definitions for various code smells. Most participants found the information that was necessary to answer this question. But the relatively high standard deviation (0.95) indicates that there were also some participants that had difficulties to answer the question. The same is true for the second statement of the first question. While most participants think that it was not necessary to navigate trough a lot of different windows, the standard deviation (0.91) is still high compared to other questions. On the other hand the results for the second question (Table 6.2) are very clear. Almost all participants found the information very easily and had not any problems while navigating to the appropriate user interface element. From these results one can conclude that this feature is in general easy to understand and intuitive. This conclusion is supported by the fact that no participant had any suggestions to improve this feature (Chapter 6.2.4).

First Question: *Does the Shotgun Surgery smell occur in the code? If yes, which class that triggers the cascading changes defines the most methods?*

Statement	**Mean**	**Median**	**STDV**
The necessary information was easy to find	4.14	4.00	0.95
To find this information it was necessary to navigate through a lot of different menus	1.71	1.50	0.91

Table 6.1: Results for question 3. (1 = "I strongly disagree" / 5 = "I strongly agree")

Second Question: *How many methods are defined in class FigNodeModelElement?*

Statement	Mean	Median	STDV
The necessary information was easy to find	4.93	5.00	0.26
To find this information it was necessary to navigate through a lot of different menus	1.00	1.00	0.00

Table 6.2: Results for question 8. (1 = "I strongly disagree" / 5 = "I strongly agree")

Dependencies

The *Dependencies* feature is used to visualize the relationship between various classes. In order to evaluate this feature the participants of the evaluation had to detect how many calls are there between the classes `TargetManager` and `ProjectManager`. The results in Table 6.3 indicate that most participants had no problems to find the appropriate functionality and view to solve this task. But the relatively low average value (2.80) and high standard deviation (1.21) for the first statement show that some users had problem to find the necessary information in the view itself.

Question: *How strong (number of calls) is the relationship between class TargetManager and class ProjectManager?*

Statement	Mean	Median	STDV
The necessary information was easy to find	2.80	3.00	1.21
To find this information it was necessary to navigate through a lot of different menus	2.07	2.00	0.80

Table 6.3: Results for question 6. (1 = "I strongly disagree" / 5 = "I strongly agree")

Capture Audio

In order to evaluate the feature to record an audio note, the participants had to solve a simple task. They had to record an audio note for a specific class in the project. As the results in Table 6.4 indicate this functionality seems to be very intuitive and easy to handle. All but one participants rated the questions related to this task with 4 ("I agree") or 5 ("I strongly agree"). This observation is supported by the relatively low standard deviation (0.59).

Task: *Record an audio note for code smell PerspectiveSupport.*

Statement	Mean	Median	STDV
The necessary functionalities to solve this task were easy to find	4.73	5.00	0.59

Table 6.4: Results for task 2. (1 = "I strongly disagree" / 5 = "I strongly agree')'

Hierarchy

The *Hierachy* feature is used to visualize the hierarchy relationship between various classes. In order to evaluate it, the participants had to find the only subclass of a code smell. All participants were able to solve this question and most of them had no problems to find the required information. Although the mean (4.07) and median (5.00) for the first statement in Table 6.5 are high, the standard deviation (1.28) indicates that there were also some participants that answered the question with 2 ("I disagree") or 3 ("I neither agree nor disagree"). A closer look at the results shows that two participants rated the statement with 3 and three participants rated the statement with 2. On the other hand all participants more or less agreed on the fact that it was not necessary to navigate trough a lot of different menus to find the appropriate feature to answer this question.

Question: *Name the direct subclasses of class PerspectiveSupport*

Statement	Mean	Median	STDV
The necessary information was easy to find	4.07	5.00	1.28
To find this information it was necessary to navigate through a lot of different menus	1.80	2.00	0.94

Table 6.5: Results for question 4. (1 = "I strongly disagree" / 5 = "I strongly agree")

Package

Sometimes during a review process it is important to investigate the package in which a code smell is located. That is why the participants had to detect in which package a class is defined in. This can be done using the *Package* feature. Almost all participants had no problems to find the information that is necessary to answer this question. Twelve participants agreed on the fact that it is easy to find the required information. The remaining three participants could not decide if they should agree or not. A very similar pattern can be recognized for the second statement in Table 6.6. Twelve participants disagreed and stated that is is not necessary to navigate through a lot of different menus. Another two participants could not decide and only one participant thought that it was difficult to find the appropriate view to answer this question.

Question: *In which package is class ProjectBrowser located?*

Statement	Mean	Median	STDV
The necessary information was easy to find	4.33	5.00	0.82
To find this information it was necessary to navigate through a lot of different menus	1.67	1.00	0.98

Table 6.6: Results for question 5. (1 = "I strongly disagree" / 5 = "I strongly agree")

False Positive

It can be necessary to mark a code smell candidate as a false positive result of the automatic detection strategies. In order to evaluate this feature, the participants had to mark a code smell as false positive and then judge how difficult it was to solve this task. All but one participants rated the statement in Table 6.7 with 4 ("I agree") or 5 ("I strongly agree") so that on average there is a high agreement (4.27) on this statement. However it is worth mentioning that one-fifth of all participants mentioned suggestions for improvements. In Chapter 6.2.4 this is discussed in detail, but in summary it can be said that the participants suggested a more active feedback if an user marks a code smell as a false positive result.

Task: *Mark code smell UndoableAction as false positive.*

Statement	Mean	Median	STDV
The necessary functionalities to solve this task were easy to find	4.27	5.00	1.22

Table 6.7: Results for task 1. (1 = "I strongly disagree" / 5 = "I strongly agree")

Overview Pyramid

The *Overview Pyramid* can be used to compare various code metrics in the scope of the whole project. Therefore the participants had to find two different metrics that indicate how inheritance is used in the project. All participants were able to answer this question. The high average value (4.33) for the first statement in Table 6.8 and the low average value (1.47) for the second statement indicate that most participants had no problems neither to navigate through the menus nor to find the necessary information.

Question: *What are the values for AHH (Average Hierarchy Height) and ANDC (Average Number of Derived Classes) metric over the whole project?*

Statement	Mean	Median	STDV
The necessary information was easy to find	4.33	5.00	1.11
To find this information it was necessary to navigate through a lot of different menus	1.47	1.00	0.92

Table 6.8: Results for question 7. (1 = "I strongly disagree" / 5 = "I strongly agree")

Package Overview

In order to evaluate this feature, the participants had to find the number of classes that are defined in a specified package. All participants answered the question shown in Table 6.9 without any problems. But one participant complained that it was difficult to find the necessary information since the visualization of class interrelations consists of too much lines so that they viewer's eye gets confused. The exemplary package that the participants had to examine includes 115 classes. That is why it is possible that the visual representation of classes overlap so that it is hard to find a specific class. The participant suggested to use graph layout algorithms (GLA) to reduce the impact of this problem.

Question: *How many classes are located in the package org.argouml.uml.cognitive.critics?*

Statement	Mean	Median	STDV
The necessary information was easy to find	4.47	5.00	0.74
To find this information it was necessary to navigate through a lot of different menus	1.67	2.00	0.82

Table 6.9: Results for question 9. (1 = "I strongly disagree" / 5 = "I strongly agree")

Metrics Overview

The feature *Metrics Overview* can be used to compare all the classes in the analyzed project against each other by comparing their metrics. In order to evaluate this feature, the participants had to find the largest class (by lines of code) in the whole project. As the results in Table 6.10 show, all participants were able to find the information easily. In the open questions part of the questionnaire (Chapter 6.2.4) two participants mentioned suggestions to improve this feature. In summary both suggestions propose to implement the possibility to filter the list for a subset of all classes. This subset of classes can be defined by the user either by using a name filter or by using drag&drop operations to choose interesting classes that should be compared against each other in this view.

Question: *What is the largest (LOC) class in the project?*

Statement	Mean	Median	STDV
The necessary information was easy to find	4.60	5.00	0.63
To find this information it was necessary to navigate through a lot of different menus	1.47	1.00	0.92

Table 6.10: Results for question 1. (1 = "I strongly disagree" / 5 = "I strongly agree")

Tagged Objects

In our *SmellTagger* application, it is possible to use tagged objects to access special features by placing these tagged objects on the Surface screen. This interaction paradigm is described in detail in Chapter 4.2.5. In order to evaluate if tagged objects are an intuitive way to interact with our *SmellTagger* application, the participants had to solve three tasks involving tagged objects. In the first task (Table 6.11) it was necessary to associated a tagged object with a code smell. The second task (Table 6.12) requires to use a tagged object to generate a review report (Chapter 4.2.4) about the current review session. In the last task (Table 6.13) tagged objects were used to manipulate views using gestures (Chapter 5.1). All three tasks could be solved by every participant. Because of the high ratings for each task it is possible to conclude that tagged objects are intuitive possibilities to interact with our *SmellTagger* application.

First task: *Tag class Translator with tag TODO. Then undo the changes.*

Statement	Mean	Median	STDV
The necessary functionalities to solve this task were easy to find	4.07	4.00	1.10

Table 6.11: Results for task 3. (1 = "I strongly disagree" / 5 = "I strongly agree")

Second task: *Generate a review report for the project.*

Statement	Mean	Median	STDV
The necessary functionalities to solve this task were easy to find	5.00	5.00	0.00

Table 6.12: Results for task 4. (1 = "I strongly disagree" / 5 = "I strongly agree")

__Third task:__ *Open the Hierarchy View of class FigNodeModelElement. Then use gestures to close this window again.*

Statement	Mean	Median	STDV
The necessary functionalities to solve this task were easy to find	4.60	5.00	0.51

Table 6.13: Results for task 5. (1 = "I strongly disagree" / 5 = "I strongly agree")

6.2.2 Review Process

The third part of the evaluation consists of questions about the interaction with our *SmellTagger* application itself and questions about multi-touch interfaces in general. The results are shown in Table 6.14.

#	Statement	Mean	Median	STDV
1	The prototype provides the user with all functionalities that are necessary to solve tasks in a review process efficiently.	4.13	4.00	0.99
2	The prototype uses symbols, terms and abbreviations that are easy to understand.	4.07	4.00	0.80
3	The prototype is qualified to be used in a review process.	4.27	5.00	0.96
4	The prototype provides the user with an useful overview about its functionalities.	3.47	4.00	1.30
5	The prototype is easy to operate because of its consistent design.	4.27	4.00	0.70
6	The prototype has a flat learning curve.	2.67	2.00	1.76
7	The prototype does not require to remember a lot of details.	3.87	4.00	1.25
8	It is difficult to learn how to operate the prototype without any help.	2.27	2.00	0.70
9	Multi-touch interfaces are more suitable in code review process than traditional interfaces.	3.40	3.00	0.99
10	Collaboration with a multi-touch interface is easier than with a traditional computer.	3.80	4.00	1.32

Table 6.14: Results for the third part of the questionnaire. (1 = "I strongly disagree" / 5 = "I strongly agree")

In summary it can be said that most participants think that our *SmellTagger* application can be used in a review session. The high average agreement on statement 1 (4.13) and statement 3 (4.27) support this argument. Additionally, it seems that for most participants it was easy to interact with the prototype because of the usage of symbols, terms and abbreviations that are easy to understand (statement 2), its consistent design (statement 5) and because it is not necessary to remember a lot of details (statement 7).

On the other hand statement 4 shows that the participants did not agree on the question that our *SmellTagger* application provides the user with an useful overview about its functionality. The average answer (3.47) is between 3 ("I neither agree nor disagree") and 4 ("I agree") and the high standard deviation (1.30) indicates that there were a lot of different answers. This impression is fortified by the fact that some participants stated that they were not able to find a specific feature. One participants mentioned that *"some interesting features are hidden behind symbols without*

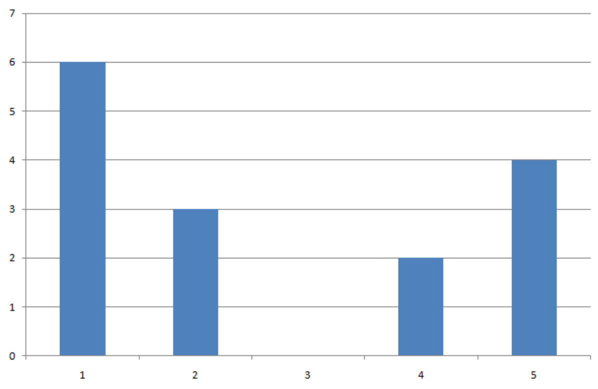


Figure 6.1: Histogram of statement 6. The x-axis shows the ratings and the y-axis the occurrence of each rating.

any indication". Another one complained that it was *"difficult to find out how to handle the various functionalities"*. This indicates that our *SmellTagger* application does not sufficiently inform the users about its functionalities and possibilities.

Another problem that became evident during the evaluation is that our *SmellTagger* application offers no explanation or help function. Although the average rating (2.27) of statement 8 indicates that it is not really difficult to learn how to operate the application without any help, a lot of participants addressed this problem in the last part of the evaluation where they were asked to write down their opinions (see Chapter 6.2.4 for details). More than one-third of all participants stated that they would appreciate a help function in any form.

Statement 9 and 10 are used to evaluate the participants' opinion about multi-touch interfaces in general. While it is not really clear if the participants think that multi-touch interfaces are more suitable in code review processes than traditional interfaces, it became evident that most participants think that multi-touch interfaces foster collaboration between users.

Statement 6 needs to be analyzed with caution. A lot of participants were not sure if a flat learning curve indicates that something is difficult or easy to learn. These doubts are visible in the histogram shown in Figure 6.1. The histogram shows that particularly the extreme ratings are very frequent. This also explains the high standard deviation (1.76). Therefore it is possible that the result for this statement is distorted.

6.2.3 Features

The fourth part of the evaluation consists of questions about the special features our *SmellTagger* application offers. The results of these questions are shown in Table 6.15.

The first statement reveals that tagged objects are widely accepted as useful form of user input. This result fits to the results presented in Chapter 6.2.1 where the participants were asked to solve tasks using tagged objects. All tasks were solved without any problem and the ratings indicate that tagged objects are easy to use. Surprisingly, one of the key feature of our *SmellTagger* application, namely the possibility to freely rotate, scale and translate almost all user interface elements is not accepted as an useful way to improve the interaction with the application. In statement 2

#	Statement	Mean	Median	STDV
1	Tagged objects are useful forms of user input.	4.07	4.00	0.88
2	The ability to freely rotate, scale and translate all elements of the user interface improves the user interaction.	3.73	4.00	1.28
3	Gestures are useful forms of user input.	3.13	3.00	1.36
4	Multi-touch interfaces foster collaboration during a review process.	3.67	4.00	0.72
5	The possibility to records audio notes is an important feature.	3.13	3.00	1.06
6	The possibility to examine the source code of a class or method is helpful.	4.20	5.00	1.37
7	The outcome of a review process (a written report) can be easily used in further development steps.	4.27	5.00	0.96

Table 6.15: Results for the fourth part of the questionnaire. (1 = "I strongly disagree" / 5 = "I strongly agree")

one can see that the average rating (3.73) is only slightly in the positive area of the rating scale. One participant even stated explicitly that he did not like the *"scaling and transforming of screens"*.

The possibility to use gestures to interact with the application is a controversial feature. The histogram in Figure 6.2 shows that seven participants rated gestures negatively while another seven users rated the feature positively. These oppositional opinions are also reflected in the last part of the questionnaire where the user could share their opinions. While one participant noted that gestures are not really useful and that it is too bothersome to activate them first through tagged objects, another participant stated that it would be an improvement to *"increase the types of gesture commands"*. Another feature that was controversially discussed is the ability to record audio notes. The median rating (3.00) shows that most of the participants do not think that it is an important feature.

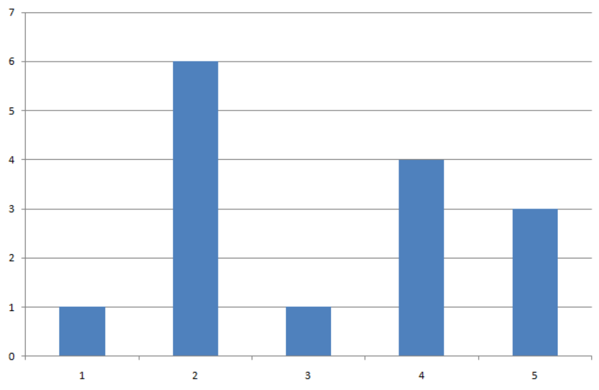


Figure 6.2: Histogram of statement 3. The x-axis shows the ratings and the y-axis the occurrence of each rating.

Statement 6 evaluates if the participants think that it is helpful to browse through the source code of a class or a method. 13 of 15 participants agree on that fact. Two participants also suggested to enhance this feature and implement a possibility to mark or tag source code snippets. A similar positive result for statement 7 indicates that the review report is an useful outcome of a review session performed with our *SmellTagger* application.

6.2.4 Open Questions

In the last part of the evaluation the participants had the possibility to share their opinions in open questions. This part of the questionnaire consisted of four different open questions. Three of them are presented in detail. The first question asked the participants what they did like the most during the evaluation. The second question addressed the aspects that the participants did not like. The third question asked the participants for suggestions to improve our *SmellTagger* application. In order to evaluate the answers, all answers are thematically grouped and the most frequently mentioned groups are presented.

What did you like the most?

Design & GUI

- *"Design and color choices."*
- *"Clear and clean design. No information overload."*
- *"Really nice user interface without information overload."*
- *"Clean user interface."*
- *"Graphical design, clear arrangement of features & functionalities."*

Navigation

- *"Quick navigation and information provided by the system."*
- *"Navigating is easy and very intuitive."*
- *"Easy access to all important info (not too much clicking required to get there)."*
- *"The whole interface that lets you see the smells right away and then allows you to dig in and find all the needed details."*
- *"Even though there is a lot of information, you never feel lost and can go back to the overview with one click".*

Easiness

- *"Very intuitive and incredibly easy."*
- *"Very fast to solve questions."*
- *"Easy learning provides a rather deep insight into the code."*
- *"With a short introduction and a little bit of trail and error it is easy to learn."*
- *"Easy to understand."*

Technology

- *"Simple presentation of code in another format."*

- *"Multi-touch technology."*
- *"Doing a code review on a multi-touch device is really great!"*

Conclusion: Most participants liked the user interface. They thought that it is easy to learn and offers a fast and intuitive navigation to solve typical problems in a review session. A lot of participants were also impressed by the multi-touch technology and mentioned that it is a great experience to do a code review in a non-traditional way. These results coincide with the results of the questions the participants had to answer and the tasks they had to solve, presented in Chapter 6.2.1.

What did you not like?

Gestures

- *"Gestures are not so useful."*
- *"Gestures are sometimes hardly recognized (more of a problem of the device than the app I guess)."*
- *"Gestures do not work good enough. I think it is too bothersome to activate them through the tagged objects."*

Technology

- *"Infrared touch mechanism is very inaccurate."*
- *"The fans of the Surface are too loud."*
- *"The Surface is just too slow."*
- *"Most of the problems are related to the technology of MS Surface itself."*
- *"The reactivity of the MS Surface."*

GUI

- *"Scaling and transforming of screens."*
- *"Screens with class interrelations draws too much lines and classes overlap such that it is hard to find them."*
- *"Graphical differentiation between GUI elements that just display something and those that allow user input."*
- *"Probably the kiviats were a bit unclear."*
- *"Some interesting features are hidden behind symbols without any indication."*

Conclusion: A lot of participants complained about the Microsoft Surface. It became evident that the touch recognition is not always working accurately. But this inaccuracy is caused by the vision system of the Microsoft Surface and not by our implementation. Nevertheless it also influences the evaluation of the gestures since the precise recognition of gestures primarily depends on accurate touch points recognition. Suggestions to improve the graphical user interface were also frequently mentioned.

Suggestions to improve the prototype application

Help

- *"More help functions."*
- *"More explanations."*
- *"A simple 'HELP' link would be useful in early stages."*
- *"Welcome / Splash screen with short tutorial."*
- *"Without your introduction a help system would be nice."*
- *"Some help info via '?' symbol."*

Search function

- *"Implement a searchable list for all classes."*
- *"Name filter / Search feature in 'Metrics Overview'."*
- *"Search function."*

Mark and annotate source code snippets

- *"The possibility to mark source code snippets."*
- *"Can you tag or mark source code?"*

Conclusion: The suggestion to implement a help function was mentioned far more frequently than any other idea for improvement. Nevertheless there were also other suggestions for improvement that were mentioned several times, for example the idea to implement a search box that allows users to search all classes in the project by name or the idea to implement a feature that allows users to mark a code snippet and add a note to it. Chapter 4.2.4 describes how the last two ideas were implemented after the evaluation.

6.3 Limitations and Conclusion

The results of this evaluation could be biased due to the selection of participants. As shown in Figure 6.3 most participants of the evaluation had no previous experience in working with Surface applications. But a lot of them had already previous experience in using other multi-touch devices, such as the iPhone, as shown in Figure 6.4.

There is the assumption that the experience in using multi-touch devices in general and Surface applications in particular will influence the results of the questions the participants had to answer in the first part of the evaluation. But the correlation matrix between those questions and the experience of the participants shown in Table 6.16 does not indicate any dependencies. Event though there are some correlations between single questions, it is not possible to detect a dependency pattern that matches all questions. It is important to note that it was not possible to calculate values for the second statement of the eight questions since the answers for this statement do not vary at all.

The evaluation has shown, that the participants are generally confident that multi-touch technology can be used beneficially in a code and design review process. In respect of our *SmellTagger* application it can be said that most participants like the intuitive user interface that is very easy to learn. Although there are a lot of suggestions for improvements, the participants are generally positive that the prototype can be used beneficially in a review session.

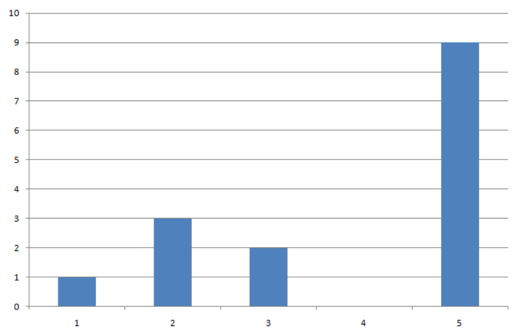


Figure 6.3: Histogram for the the statement "I am not experienced in using MS Surface applications" (1 = "I strongly disagree" / 5 = "I strongly agree")

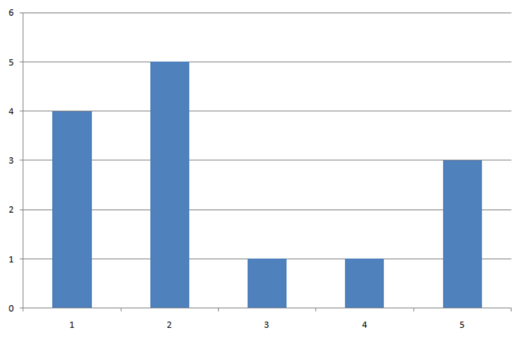


Figure 6.4: Histogram for the the statement "I am not experienced in using other multi-touch devices" (1 = "I strongly disagree" / 5 = "I strongly agree")

Question / Statement	Additional Question 1	Additional Question 2
Question 1 / Statement 1	-0.36	-0.11
Question 1 / Statement 2	0.10	0.20
Question 2 / Statement 1	-0.21	-0.44
Question 2 / Statement 2	0.21	-0.11
Question 3 / Statement 1	-0.56	-0.05
Question 3 / Statement 2	0.56	0.16
Question 4 / Statement 1	0.08	0.05
Question 4 / Statement 2	0.08	0.09
Question 5 / Statement 1	0.04	-0.06
Question 5 / Statement 2	0.36	0.34
Question 6 / Statement 1	-0.17	-0.13
Question 6 / Statement 2	0.78	0.38
Question 7 / Statement 1	-0.36	-0.77
Question 7 / Statement 2	0.31	0.72
Question 8 / Statement 1	-0.21	0.11
Question 8 / Statement 2	NA	NA
Question 9 / Statement 1	0.06	-0.40
Question 9 / Statement 2	-0.15	0.23

Table 6.16: Correlation matrix between the two additional questions in the questionnaire that measure the previous experience of each participant and the nine questions about tasks the participants had to solve

Chapter 7

Final Remarks

This chapter subsumes the insights on how multi-touch technology can be beneficially used in a design and code review. Additionally, suggestions how to improve our *SmellTagger* application are presented.

7.1 Conclusion

While the new multi-touch technology is widely adapted in various domains such as customer servicing, very little research has been performed on how this technology can be used beneficially in the field of software engineering. Our *SmellTagger* application contributes to overcome this issue. It is designed to exploit the features this new technology offers and to provide the user with a new kind of user interface that allows intuitive and natural interaction. Our application supports software engineers in finding code smells in a software project and provides means to analyze those code smells thoroughly. Therefore our application can be used as a starting point for a refactoring process. The design of the user interface follows established design principles that are enhanced to fit into a multi-touch environment. Additionally, our application exploits the features the Microsoft Surface offers in order to provide the users with a new kind of interaction.

The evaluation of our *SmellTagger* application has shown that the participants are generally confident that multi-touch technology can be used beneficially in a review process. In respect of our *SmellTagger* application it can be said that the participants are generally positive that the application can be used advantageously in a code and design review. Although there are a lot of suggestions for improvements, most participants liked the intuitive and easy to learn user interface.

For future work we plan to implement an interactive help function so that it will become easier for new users to get used with the application. In addition, we intend to not only analyze static project snapshots, but also dynamic information extracted from a version control or bug tracking system.

7.2 Future Work

There are many further features which could be implemented to extend our *SmellTagger* application to improve its usability and functionality. In the following chapters, some extensions are described as an example.

7.2.1 Tutorial and Interactive Help

The evaluation has shown that the users of our application are sometimes a little bit overextended when they are first confronted with the application. For a lot of people working with a multi-touch interface to such an extent was a new experience. That is why it was difficult for them to get used to the application without further help. Consequently, a frequently mentioned suggestion for improvement was to include a tutorial or interactive help function that explains how to use the application. Compared to a more traditional application, well-known interactive help functions, for example *Tooltips*, are difficult to implement in a multi-touch application. That is why it is necessary to use other kind of help functions. Two ideas, tutorials and interactive help, are described in the next two chapters.

Tutorial

This idea is based on the way modern video games help players to understand the control mechanisms and the story of a game. In these games the player often has to complete a short introductory mission that explains how the game works. The player learns how to control and handle the game and very often he also learns facts about the story the game is based on. These principles can also be used in our *SmellTagger* application. Before the first use, the user can optionally take a guided virtual tour through an exemplary project. There he learns how to interact with the application and how to solve typical tasks that have to be solved during a review session.

Interactive Help

The second idea is based on the principle that the user can request help if he needs assistance. In contrast to the tutorial approach the user will only get an explanation for the feature or view he is currently interacting with. The idea is that whenever the user does not know how to interact with a view or how to use a feature, he places a special tagged object on the screen. This object is recognized by the application and a help movie or a written explanation is shown that explains how to use the feature or view the tagged object was placed on. In that way the user does not have to remember all the information he got when doing the tutorial, but can request help whenever he needs it.

7.2.2 Display Dynamic Content

At the moment our *SmellTagger* application displays static content. It is only possible to analyze a project snapshot that was taken at a specific point in time. All modifications that were made later are not regarded. Therefore the analysis of a project is always based on old data. As an additional problem, the single data source used to detect code smells are code metrics. In order to enhance the quality of a review session there are various approaches that allow to include actual data from various sources:

- Information from a bug tracking system, for example *Bugzilla*,[1] can be included. This information help to differentiate between code smell candidates that are really code smells and those that were just marked accidentally as code smells by the detection strategies. This approach is valid since it can be assumed that code smell classes have a higher probability to contain bugs than other classes. Emden supports this assumption with his statement that *"the fewer bugs (or defects) there are present in a piece of code, the higher the quality of that code"* [vEM02].

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

- Version control systems, for example *Subversion*,[2] can provide useful information. With this information it is possible to see how many times a software entity was changed. This can also help identifying code smells since software entities with code smells are more likely to change [KDPG09].
- Using information from version control systems allows to display live information about a system. For example it is possible to show which developer is currently working on a class or an operation. This information is not truly useful when detecting code smells in a software project but it could make sense to display such information in break rooms or meeting rooms of a software development company.

7.2.3 Extend Structural Information

The structural information about a project is currently narrowed down to *Packages*, *Classes*, *Interfaces* and *Methods*. But in most object-oriented languages there are more software entities available such as *Annotations*, *Enumerations*, *Exceptions* or *Anonymous Inner Classes*. By including those software entities in the analysis of a software project it is possible to infer more detailed and meaningful information.

7.2.4 Review Report

The possibility to create a review report is well accepted by the users as the evaluation described in Chapter 6 has shown. Nevertheless some participants of the evaluation suggested two interesting ideas of improving. Currently it is only possible to have a look at the review report on the Surface unit itself. Although it is possible to search the report file in the file system and then distribute it to other devices, it would be a great improvement, if the review report could be distributed to all reviewers and other interested people directly within the application. For this purpose various possibilities are suitable. Sending the review report per email is only one of them. Another even more interesting approach is offered by the possibility to connect Bluetooth-enabled mobile devices to the Surface.[3] This allows users to interact with data or the contacts of the connected device. In that way it is possible to either directly send the data to a mobile device, or to use the contacts that are stored on the mobile device to send the review report per email to selected contacts. This feature can be used for example to send review reports directly to the reviewers mobile phones or other interested persons.

Another interesting idea of improvement that was suggested by the participants of the evaluation is the customization of the review report. At the moment, the user does not have any possibility to choose or manipulate the content of the review report. It is automatically generated by our *SmellTagger* application. But if the user would have the possibility to select the content of the review report, he could choose individually what information is significant for the actual review session. In that way the possibility to create a review report gets even more valuable.

7.2.5 Speech-to-Text

Our *SmellTagger* application already provides the users with the possibility to record audio notes. This feature is primarily implemented because typing with a virtual keyboard can be cumbersome and because speech is one of the most natural way to interact.

[2]http://subversion.apache.org/
[3]http://technet.microsoft.com/en-us/library/ee692087(Surface.10).aspx

Since .NET 3.0 speech recognition is supported and directly build into the .NET framework by the namespace `System.Speech.Recognition`.[4] Speech recognition converts spoken words to written text. This possibility to convert speech into written words can be used to capture notes more beneficially. The audio notes that are currently added as audio files to the review report could be directly converted into text. In that way the notes are more useful when the review report is analyzed, since finding information in text is easier than browsing through an audio file to find the appropriate position.

7.2.6 Microsoft Surface 2.0

At the CES 2011 in Las Vegas, Microsoft announced the next generation of Microsoft Surface [Rela]. This new *Surface 2.0* will use *"PixelSense technology, which gives LCD panels the power to see without the use of cameras"* [Rela]. This will improve and accelerate the recognition of finger, hands and objects that are placed on the screen. There are also other improvements such as the larger screen that has now a 40 inch display with a resolution of 1920 x 1080 pixels, the improved processing power or the refined Windows 7 operating system. Figure 7.1 shows that the Microsoft Surface 2.0 is much thinner than its predecessor. Therefore it is also possible to mount it on a wall. All these modifications do have impact on the way the Microsoft Surface can be used. That is why it is important to revise already existing applications and adapt them to the new environment.

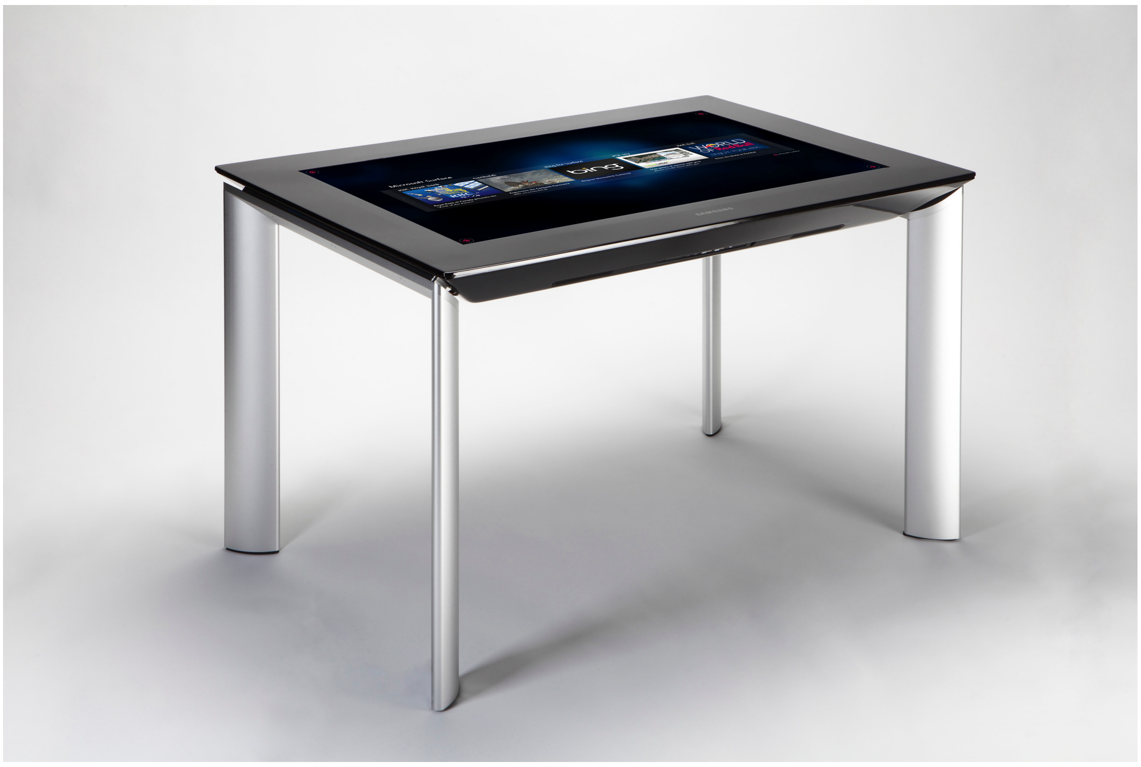


Figure 7.1: Microsoft Surface 2.0

[4]http://msdn.microsoft.com/en-us/library/system.speech.recognition(v=VS.85).aspx

7.2.7 Filtering Classes Based on Various Characteristics

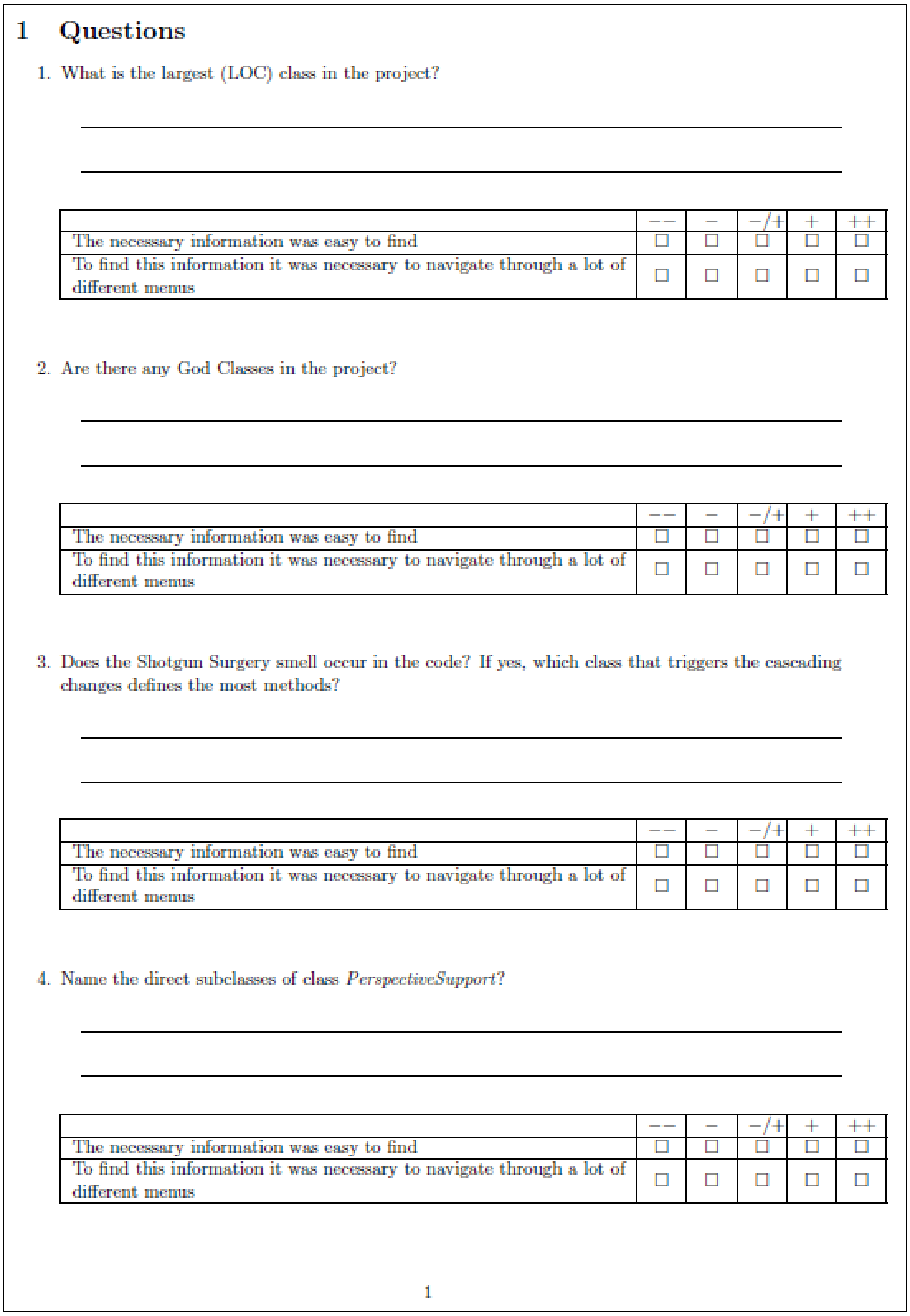
Our *SmellTagger* application allows to filter classes by name. For example if there is a view that has to visualize a lot of classes, it is possible to filter out classes by class names. In a similar way it is also possible to search for classes by names. But this filtering mechanism is not always good enough. By implementing more filters, such as filters that sort out classes by code metrics, it is possible to provide the users with additional selection criteria. Thereby the views become more customizable and therefore useful.

7.2.8 Enhancing Web Service with Rule-based Detection Strategies

In Chapter 5.2 it is described how rule-based detection strategies are used to identify code smells in a software project. The detection strategies need to use code metrics obtained from the web service described in Chapter 3.4. Therefore it is necessary to first retrieve all necessary code metrics from the SOFAS web service and then perform an analysis with the detection strategies. In order to improve and facilitate this process, the web service can be enhanced to provide a service that merges this two subprocesses. As a consequence it will be possible to directly retrieve code smell candidates from the web service without retrieving the code metrics first.

Appendix A

Questionnaire



1 Questions

1. What is the largest (LOC) class in the project?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

2. Are there any God Classes in the project?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

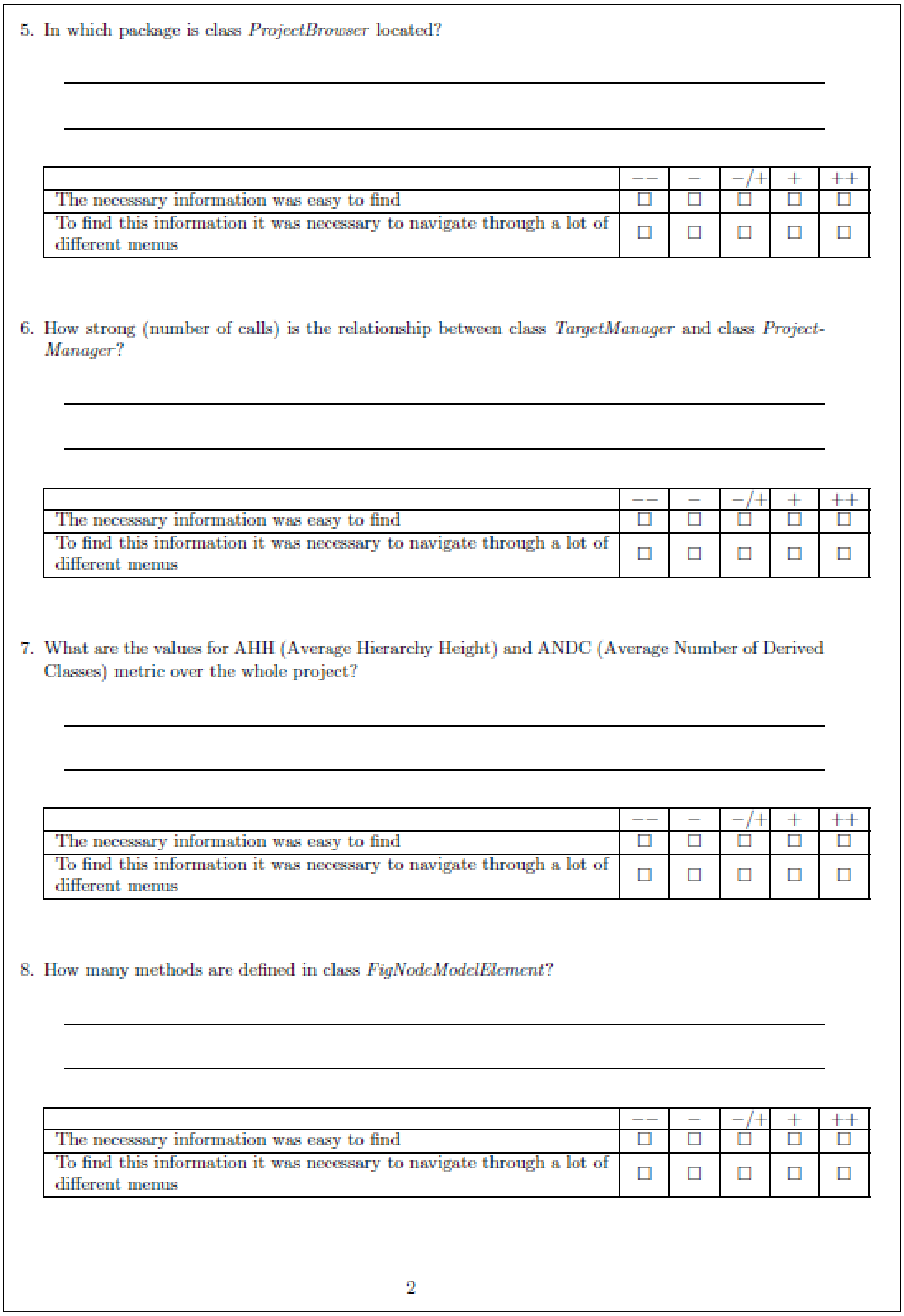
3. Does the Shotgun Surgery smell occur in the code? If yes, which class that triggers the cascading changes defines the most methods?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

4. Name the direct subclasses of class *PerspectiveSupport*?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

1



5. In which package is class *ProjectBrowser* located?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

6. How strong (number of calls) is the relationship between class *TargetManager* and class *Project-Manager*?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

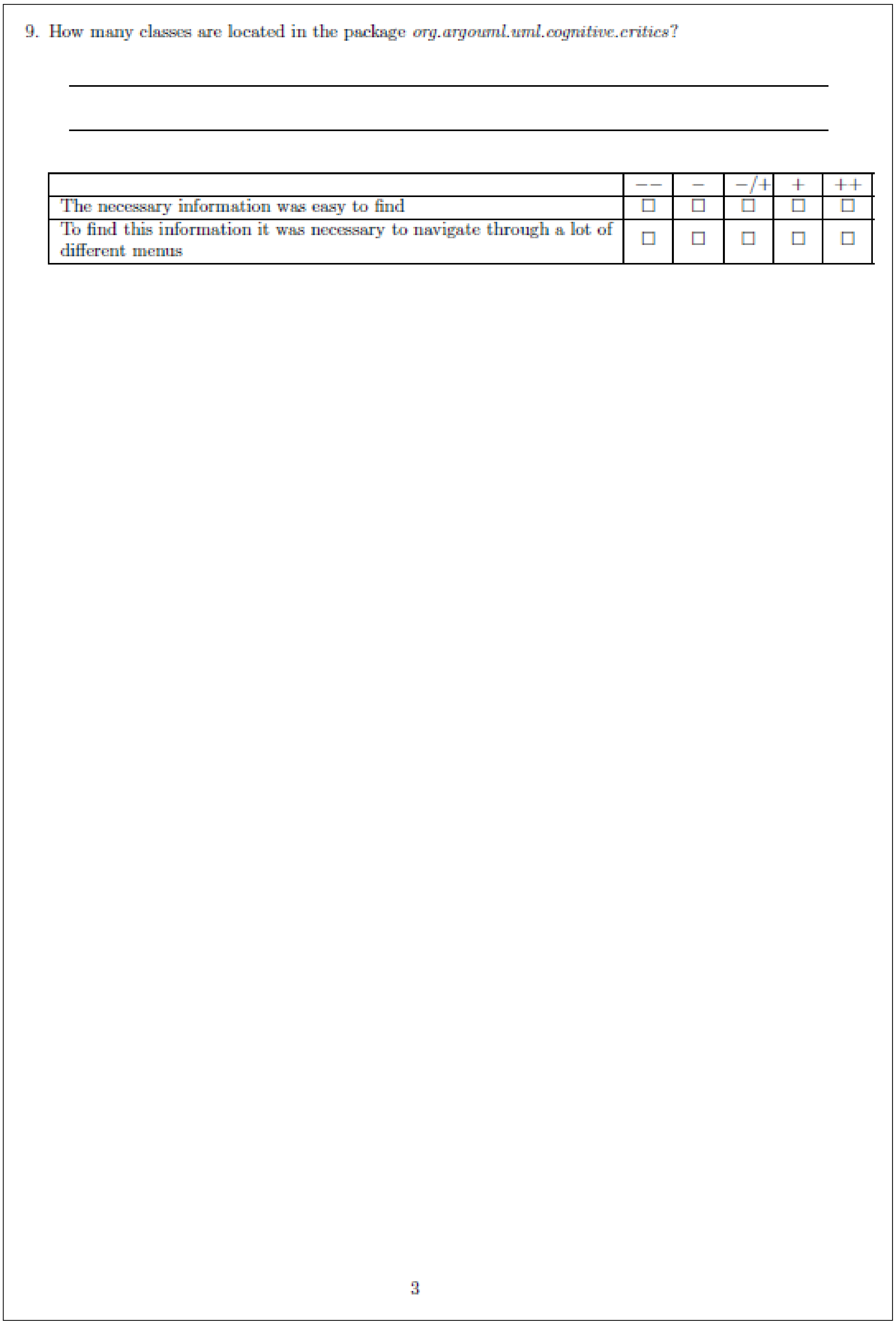
7. What are the values for AHH (Average Hierarchy Height) and ANDC (Average Number of Derived Classes) metric over the whole project?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

8. How many methods are defined in class *FigNodeModelElement*?

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

2



9. How many classes are located in the package *org.argouml.uml.cognitive.critics*?

____

____

	−−	−	−/+	+	++
The necessary information was easy to find	☐	☐	☐	☐	☐
To find this information it was necessary to navigate through a lot of different menus	☐	☐	☐	☐	☐

3

2 Tasks

1. Mark code smell *UndoableAction* as false positive

	−−	−	−/+	+	++
The necessary functionalities to solve this task were easy to find	☐	☐	☐	☐	☐

2. Record an audio note for code smell *PerspectiveSupport*

	−−	−	−/+	+	++
The necessary functionalities to solve this task were easy to find	☐	☐	☐	☐	☐

3. Tag class *Translator* with tag *TODO*. Then undo the changes.

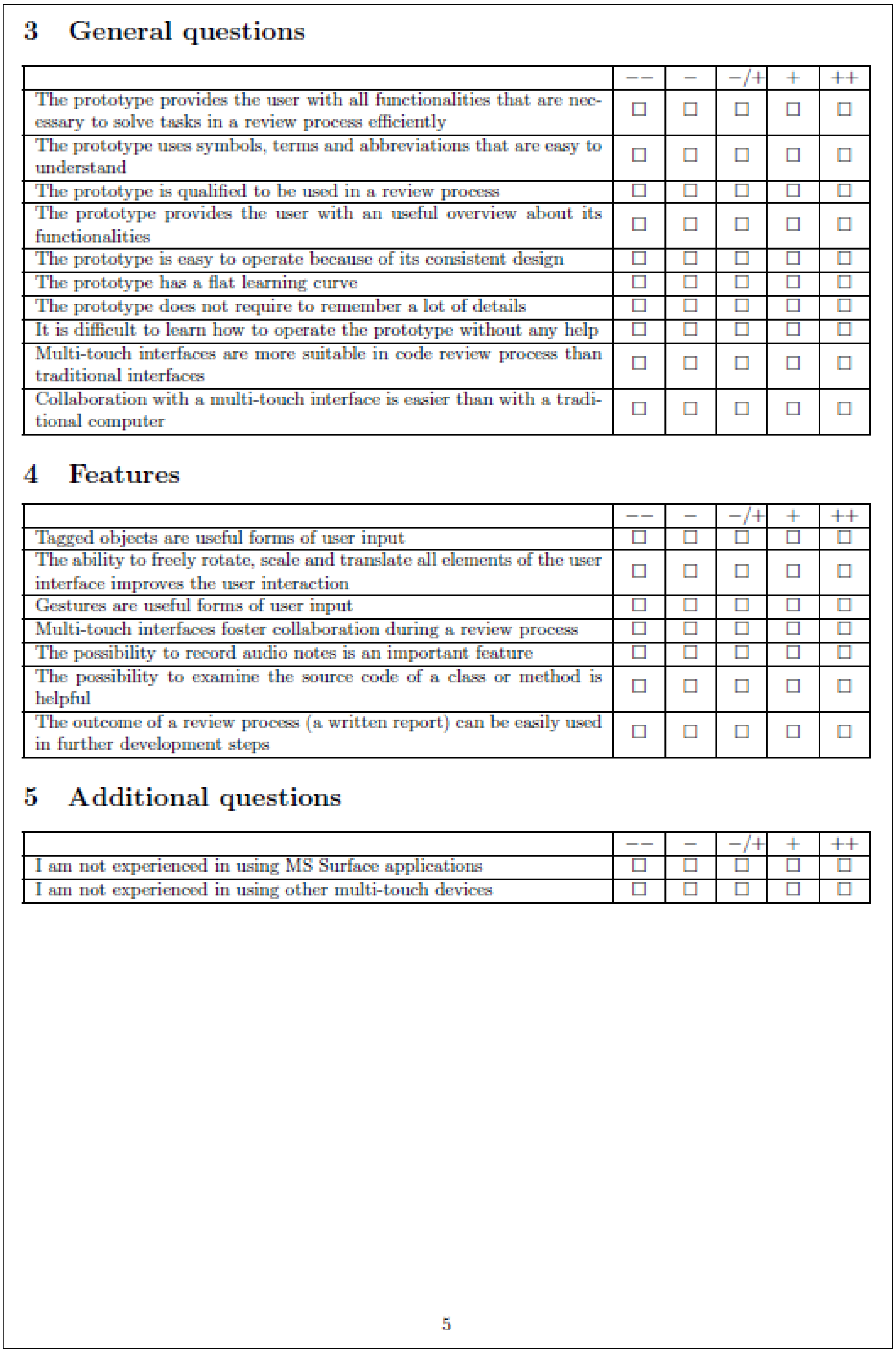
	−−	−	−/+	+	++
The necessary functionalities to solve this task were easy to find	☐	☐	☐	☐	☐

4. Generate a review report for the project

	−−	−	−/+	+	++
The necessary functionalities to solve this task were easy to find	☐	☐	☐	☐	☐

5. Open the *Hierarchy View* of class *FigNodeModelElement*. Then use gestures to close this window again.

	−−	−	−/+	+	++
The necessary functionalities to solve this task were easy to find	☐	☐	☐	☐	☐



3 General questions

	−−	−	−/+	+	++
The prototype provides the user with all functionalities that are necessary to solve tasks in a review process efficiently	☐	☐	☐	☐	☐
The prototype uses symbols, terms and abbreviations that are easy to understand	☐	☐	☐	☐	☐
The prototype is qualified to be used in a review process	☐	☐	☐	☐	☐
The prototype provides the user with an useful overview about its functionalities	☐	☐	☐	☐	☐
The prototype is easy to operate because of its consistent design	☐	☐	☐	☐	☐
The prototype has a flat learning curve	☐	☐	☐	☐	☐
The prototype does not require to remember a lot of details	☐	☐	☐	☐	☐
It is difficult to learn how to operate the prototype without any help	☐	☐	☐	☐	☐
Multi-touch interfaces are more suitable in code review process than traditional interfaces	☐	☐	☐	☐	☐
Collaboration with a multi-touch interface is easier than with a traditional computer	☐	☐	☐	☐	☐

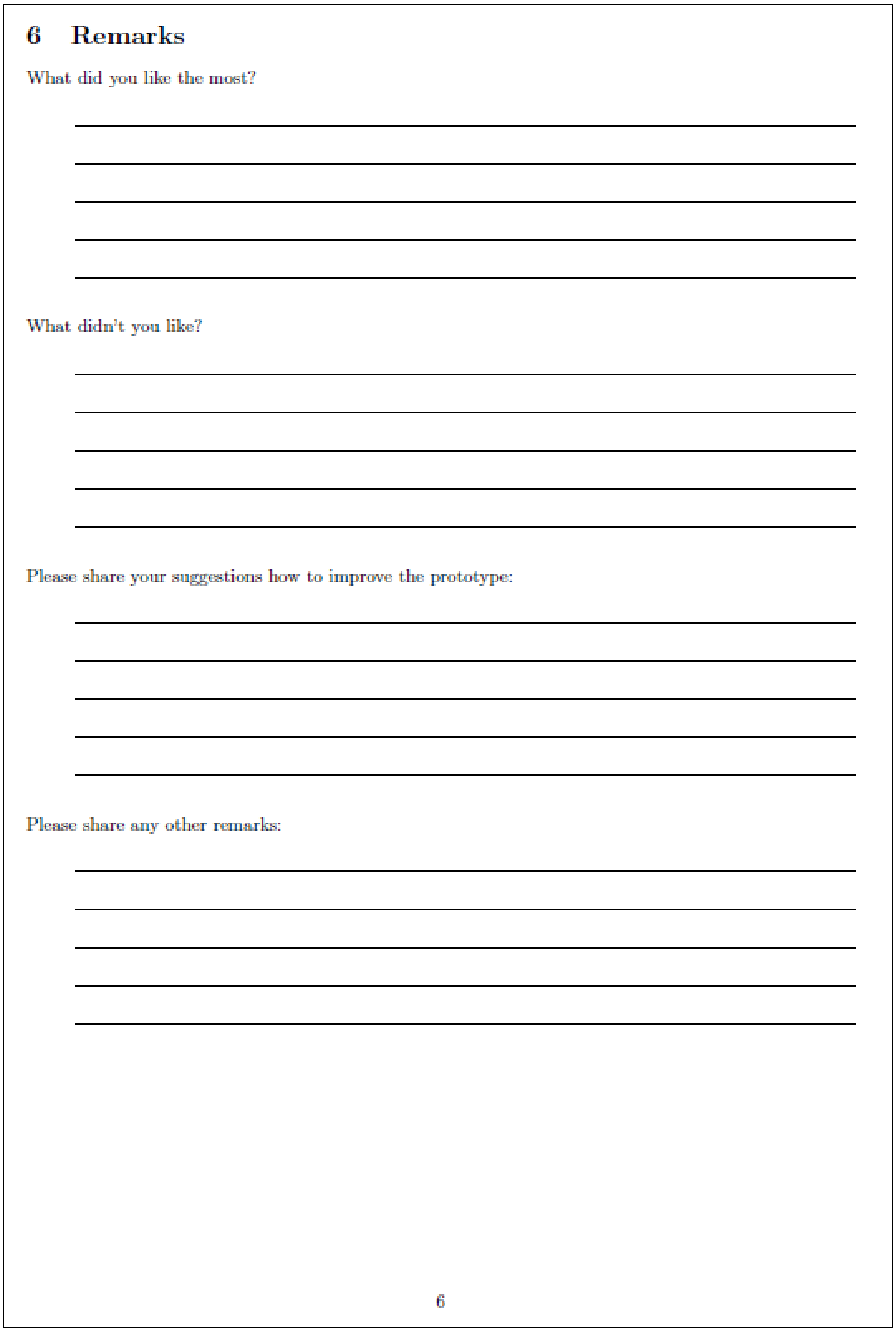
4 Features

	−−	−	−/+	+	++
Tagged objects are useful forms of user input	☐	☐	☐	☐	☐
The ability to freely rotate, scale and translate all elements of the user interface improves the user interaction	☐	☐	☐	☐	☐
Gestures are useful forms of user input	☐	☐	☐	☐	☐
Multi-touch interfaces foster collaboration during a review process	☐	☐	☐	☐	☐
The possibility to record audio notes is an important feature	☐	☐	☐	☐	☐
The possibility to examine the source code of a class or method is helpful	☐	☐	☐	☐	☐
The outcome of a review process (a written report) can be easily used in further development steps	☐	☐	☐	☐	☐

5 Additional questions

	−−	−	−/+	+	++
I am not experienced in using MS Surface applications	☐	☐	☐	☐	☐
I am not experienced in using other multi-touch devices	☐	☐	☐	☐	☐

5



6 Remarks

What did you like the most?

What didn't you like?

Please share your suggestions how to improve the prototype:

Please share any other remarks:

6

Appendix B

Results of the Evaluation

#	Duration	Questions																										
		1			*2*			*3*			*4*			*5*			*6*			*7*			*8*			*9*		
		C	E	N	C	E	N	C	E	N	C	E	N	C	E	N	C	E	N	C	E	N	C	E	N	C	E	N
1	21	x	3	1	x	5	2	x	2	2	x	3	2	x	5	2	x	5	2	x	5	1	x	4	1	x	5	1
2	25	x	4	2	x	3	1				x	2	2	x	3	2	x	2	3	x	2	4	x	5	1	x	3	1
3	40	x	5	1	x	5	1	x	4	1	x	5	1	x	5	1	x	2	2	x	5	1	x	5	1	x	5	1
4	37	x	5	1	x	5	1	x	5	1	x	2	2	x	3	1	x	2	1	x	5	1	x	5	1	x	4	2
5	34	x	4	4	x	5	1	x	4	2	x	5	2	x	4	3	x	3	2	x	5	2	x	5	1	x	4	2
6	21	x	5	1	x	5	1	x	3	2	x	5	1	x	4	2	x	2	3	x	3	2	x	5	1	x	5	1
7	18	x	5	1	x	5	1	x	5	1	x	5	2	x	5	1	x	3	3	x	5	1	x	5	1	x	5	2
8	30	x	4	2	x	5	1	x	3	4	x	3	3	x	3	3	x	1	3	x	5	1	x	5	1	x	5	1
9	20	x	4	3	x	5	1	x	5	3	x	5	3	x	5	4	x	3	3	x	2	3	x	5	1	x	4	4
10	25	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1	x	3	2	x	4	1	x	5	1	x	5	2
11	30	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1
12	25	x	5	1	x	5	1	x	5	1	x	5	1	x	5	1	x	4	1	x	5	1	x	5	1	x	5	2
13	20	x	5	1	x	5	1	x	4	2	x	4	1	x	5	1	x	3	2	x	5	1	x	5	1	x	5	2
14	24	x	5	1	x	5	1	x	4	2	x	5	1	x	4	1	x	3	2	x	4	1	x	5	1	x	4	1
15	35	x	5	1	x	5	1	x	4	1	x	2	4	x	4	1	x	1	1	x	5	1	x	5	1	x	3	2
Results																												
A	27.00		4.60	1.47		4.87	1.07		4.14	1.71		4.07	1.80		4.33	1.67		2.80	2.07		4.33	1.47		4.93	1.00		4.47	1.67
M	25.00		5.00	1.00		5.00	1.00		4.00	1.50		5.00	2.00		5.00	1.00		3.00	2.00		5.00	1.00		5.00	1.00		5.00	2.00
H	40	15	5.00	4.00	15	5.00	2.00	14	5.00	4.00	15	5.00	4.00	15	5.00	4.00	15	5.00	3.00	15	5.00	4.00	15	5.00	1.00	15	5.00	4.00
L	18		3.00	1.00		3.00	1.00		2.00	1.00		2.00	1.00		3.00	1.00		1.00	1.00		2.00	1.00		4.00	1.00		3.00	1.00
D	6.93		0.63	0.92		0.52	0.26		0.95	0.91		1.28	0.94		0.82	0.98		1.21	0.80		1.11	0.92		0.26	0.00		0.74	0.82

#	Tasks					General Questions										Features							AQ	
	1	2	3	4	5	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	1	2
1	5	4	4	5	5	4	4	5	5	4	2	4	2	3	3	4	4	3	3	2	5	5	5	2
2	2	3	3	5	4	2	4	3	2	3	2	5	4	5	4	4	4	2	3	2	5	3	5	5
3	5	5	4	5	5	2	4	2	1	4	1	5	3	2	1	2	2	1	3	3	1	2	5	2
4	1	4	1	5	4	5	5	5	4	5	1	5	2	3	2	3	1	2	4	3	1	5	1	1
5	4	5	3	5	4	4	5	3	4	5	2	4	3	4	3	4	5	5	3	4	5	4	3	2
6	5	5	5	5	5	4	4	5	4	4	1	2	3	4	4	4	4	5	4	3	5	5	5	5
7	5	5	4	5	4	5	4	5	4	4	4	3	2	5	5	5	5	2	5	4	5	3	5	2
8	4	5	5	5	5	4	3	5	3	4	1	5	2	3	5	5	5	4	5	4	4	5	5	1
9	4	5	5	5	4	5	3	4	3	4	4	3	2	4	5	4	5	2	4	2	4	4	5	5
10	5	5	5	5	5	5	3	5	3	5	5	5	2	4	5	5	4	4	4	4	5	5	2	1
11	5	5	5	5	5	5	5	5	5	5	5	5	2	3	4	5	4	4	4	3	5	5	2	1
12	4	5	5	5	5	4	5	5	5	5	1	2	2	3	4	4	3	5	3	4	5	5	2	4
13	5	5	4	5	5	4	5	4	4	5	5	5	2	4	5	5	5	2	4	5	5	5	5	2
14	5	5	4	5	5	5	4	4	4	4	1	3	1	2	5	3	3	4	3	3	4	4	5	3
15	5	5	4	5	4	4	3	4	1	3	5	2	2	2	2	4	2	2	3	1	4	4	3	3
Results																								
A	4.27	4.73	4.07	5.00	4.60	4.13	4.07	4.27	3.47	4.27	2.67	3.87	2.27	3.40	3.80	4.07	3.73	3.13	3.67	3.13	4.20	4.27	3.87	2.60
M	5.00	5.00	4.00	5.00	5.00	4.00	4.00	5.00	4.00	4.00	2.00	4.00	2.00	3.00	4.00	4.00	4.00	3.00	4.00	3.00	5.00	5.00	5.00	2.00
H	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	4.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00	5.00
L	1.00	3.00	1.00	5.00	4.00	2.00	3.00	2.00	1.00	3.00	1.00	2.00	1.00	2.00	1.00	2.00	1.00	1.00	3.00	1.00	1.00	2.00	1.00	1.00
D	1.22	0.59	1.10	0.00	0.51	0.99	0.80	0.96	1.30	0.70	1.76	1.25	0.70	0.99	1.32	0.88	1.28	1.36	0.72	1.06	1.37	0.96	1.51	1.50

Explanations

C = Correctness: Was the questions answered correctly?
E = Easiness: How difficult was it to find the necessary information
N = Navigation: How much navigation was necessary to find the required information?
AQ = Additional questions
A = Average / M = Median
H = Highest value / L = Lowest value
1 = Strongly disagree / 5 = Strongly agree
D = Standard deviation

	What did you like the most?	What didn't you like?	Suggestions to improve the prototype	Any other remarks
1	Design Simple presentation of code in another format	Gestures are not so useful	More help functions	Looks professional Software keyboard would be better than hardware keyboard
2	Multi-touch technology Design and color choices	It was difficult to find out how to handle the various functionalities	More explanations	Problems to understand the concept of code smells
3	Nice GUI	Infrared touch mechanism is very inaccurate	Sort the classes in the Dependency-View alphabetically Filter packages in Package-View according to the full name Implement a searchable list for all classes	
4	Clear and clean design. No information overload Very intuitive and incredibly easy Very fast to solve questions	Scaling and transforming of screens Screens with class interrelations draws to much lines and classes overlaps such that it's hard to find them	Graph layout algorithms (GLA). Maybe diferent GLAs activated by different tagged objects. More active feedback when things changes. E.g. When class is set to "false positive"the action should get better feedback	It made my day. Absolutely awesome!
5	Quick navigation and information provided by the system		A simple 'HELP' link would be useful in early stages Allow users (maybe 2) to use the surface collaboratively to perform 2 tasks or more	Interesting, I enjoyed it
6	Doing a code review on a multi-touch device is really great! Navigating is easy and very intuitive!	Gestures are sometimes hardly recognized (more of a problem of the device than the app I guess)	Generating a list of metrics of an arbitrary number of classes could be done nicely with Drag&Drop	
7	Another perspective on the topic Easy learning provides a rather deep insight into the code Automatic tagging of code smells by heuristic	The fans of the Surface are too loud Gestures don't work good enough. I think it is too bothersome to activate them through the tagged objects.	Color of the background should represent the "quartainary" use philosophy of the prototype. Users should be able to visually diffrenciate between a "toggle" button and a menu button.	
8		Graphical differentiation between GUI elements that just display something and those that allow user input.	Bookmarks that allow to store interesting screens, reports, etc. Welcome / Splash screen with short tutorial Name filter / Search feature in "Metrics Overview"	
9	Table of metrics is very easy to compare the quality of classes.		Increase the types of gesture commands.	
10	Really nice user interface No information overload Easy access to all important info (not too much clicking required to get there)	The Surface is just too slow	Send review report by email Without your introduction a help system would be nice	Very nice work
11	The whole interface that lets you see the smells right away and then allows you to dig in and find all the needed details. Even though there is a lot of information, you never feel lost and can go back to the overview with one click.	There is nothing that I really didn't like. Most of the problems are related to the technology of MS Surface itself. Probably the kiviats were a bit unclear.	Overall I'm impressed by it, it's really cool. The kiviats, packages graphs could probably improved to b more readable for the user. But these are layout issues. It would be nice to be able to pick which smells to display and which to ignore. That is, making it a bit more customizable.	
12	Excellent design! Looks really nice! With a short introduction and a little bit of trial and error it is easy to learn.		Search function Feedback if the user marks a code smell as "false positive" The possibiliy to mark source code snippets	
13	Clean user interface Easy to understand Great set of provided functionality	Some interesting features are hidden behind symbols without any indication. System feedback where the screen doesn't change would be beneficial	Can you tag or mark source code? Some help info via '?' symbol	
14	Graphical design, clear arrangement of features & functionalities Often implemented functionality of "changing the subject of the analysis" - e.g. Dependencies - I start from a code smell class, but can analyze other classes' dependencies.	The reactivity of the MS Surface	Make the functionalities (analyses) of the whole project (the button in the middle of the screen) more visible.	it is fun to work with the prototype, however I don't know exactly if and how that would really support a software engineer.
15	The organisation of the tools.	Tagged objects Closing windows on focus change	Auto gestures without placing a tagged objects on the screen More information about the hidden classes (classes that are not code smells) Information about the importance of the class in the system.	

Appendix C

Contents of the CD-ROM

The following files are stored on the CD-ROM:

- **Zusfsg.pdf**
  German version of the abstract of this thesis
- **Abstract.pdf**
  English version of the abstract of this thesis
- **Masterarbeit.pdf**
  Copy of this thesis
- **Questionnaire.pdf**
  The questionnaire used for the evaluation of the prototype described in this thesis
- **SmellTagger.rar**
  The application described in this thesis
- **Documentation.rar**
  The documentation of the application described in this thesis
- **Questionnaire Results.xlsx**
  The results of the questionnaire

Appendix D

Used Libraries and Tools

- **iTextSharp - http://www.itextsharp.com/**
  Port of the open source Java library iText. Can be used to programmatically create and manipulate Adobe compatible PDF documents on Microsoft's .NET Framework.
- **Lame MP3 Encoder - http://lame.sourceforge.net/**
  High quality MPEG Audio Layer III (MP3) encoder. Used in this thesis to encode WAVE audio files into MP3 files.
- **Ghostscript - http://www.ghostscript.com/**
  An interpreter for the PostScript language and for PDF. Used in this thesis to convert PDF files into PNG images.
- **Microsoft Surface SDK 1.0 SP 1**
  Enhances the .NET framework in order to provide access to the specific features of Microsoft Surface
- **AvalonEdit - http://www.avalonedit.com/**
  WPF-based text editor. Used in this thesis to display source code.
- **Surface Simulator Automation API**
  Provides possibilities to programmatically simulate contact input for Microsoft Surface. Used in this thesis to test the application visually.
- **NAudio - http://naudio.codeplex.com/**
  An .NET audio and MIDI library. Used in this thesis to record audio notes in WAVE format.

References

[Alg07] Jarallah S. Alghamdi. Measuring software coupling. In *Proceedings of the 6th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, pages 6–12, Stevens Point, Wisconsin, USA, 2007. World Scientific and Engineering Academy and Society (WSEAS).

[APH$^+$10] Panu Akerman, Arto Puikkonen, Pertti Huuskonen, Antti Virolainen, and Jonna Häkkilä. Sketching with strangers: in the wild study of ad hoc social communication by drawing. In *Proceedings of the 12th ACM international conference on Ubiquitous computing*, Ubicomp '10, pages 193–202, New York, NY, USA, 2010. ACM.

[AW10] Lisa Anthony and Jacob O. Wobbrock. A lightweight multistroke recognizer for user interface prototypes. In *Proceedings of Graphics Interface 2010*, GI '10, pages 245–252, Toronto, Ontario, Canada, 2010. Canadian Information Processing Society.

[Ber] Matt Berther. The specification pattern: A primer. `http://mattberther.com/2005/03/25/the-specification-pattern-a-primer`, accessed March 2011.

[BG07] Sandro Boccuzzo and Harald Gall. Cocoviz: Towards cognitive software visualizations. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 72 –79, 2007.

[Bux10] Bill Buxton. A touching story: A personal perspective on the history of touch interfaces past and future. *Society for Information Display (SID) Symposium Digest of Technical Papers*, 41:444–448, 2010.

[Bux11] Bill Buxton. Multi-touch systems that i have known and loved. Microsoft Research, February 2011.

[Che10] Chaomei Chen. Information visualization. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(4):387–403, 2010.

[EF97] Eric Evans and Martin Fowler. Specifications. *Proceedings of PLoP 97 Conference*, 1997.

[FC01] Stephen Fallows and Balasubramanyan Chandramohan. Multiple approaches to assessment: Reflections on use of tutor, peer and self-assessment. *Teaching in Higher Education*, 6:229 – 246, 2001.

[Few04] Stephen Few. *Show me the numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2004.

[FHD09] Mathias Frisch, Jens Heydekorn, and Raimund Dachselt. Investigating multi-touch and pen gestures for diagram editing on interactive surfaces. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, ITS '09, pages 149–156, New York, NY, USA, 2009. ACM.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[Gar10] Vahid Garousi. Applying peer reviews in software engineering education: An experiment and lessons learned. *Education, IEEE Transactions on*, 53(2):182 –193, May 2010.

[Ghe10] Giacomo Ghezzi. Sofas: software analysis services. volume 2. ACM, May 2010.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005.

[HIB+07] Steve Hodges, Shahram Izadi, Alex Butler, Alban Rrustemi, and Bill Buxton. Thinsight: versatile multi-touch sensing for thin form-factor displays. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 259–268, New York, NY, USA, 2007. ACM.

[HK] Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*.

[IBR+03] Shahram Izadi, Harry Brignull, Tom Rodden, Yvonne Rogers, and Mia Underwood. Dynamo: a public interactive surface supporting the cooperative sharing and exchange of media. In *Proceedings of the 16th annual ACM symposium on User interface software and technology*, UIST '03, pages 159–168, New York, NY, USA, 2003. ACM.

[IEE08] Ieee standard for software reviews and audits. *IEEE STD 1028-2008*, pages 1 –52, 15 2008.

[Kar04] Levent Burak Kara. An image-based trainable symbol recognizer for sketch-based interfaces. In *in AAAI Fall Symposium Series 2004: Making Pen-Based Interaction Intelligent and Natural*, pages 99–105. AAAI Press, 2004.

[KDPG09] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *Proceedings of the 2009 16th Working Conference on Reverse Engineering*, WCRE '09, pages 75–84, Washington, DC, USA, 2009. IEEE Computer Society.

[Ker01] Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2001.

[LHR88] Karl Lieberherr, Ian Holland, and Arthur Riel. Object-oriented programming: an objective sense of style. *SIGPLAN Not.*, 23:323–334, January 1988.

[Liba] Microsoft TechNet Library. Physical features of a microsoft surface unit. `http://technet.microsoft.com/en-us/library/ee692114(Surface.10).aspx`, accessed April 2011.

[Libb] MSDN Library. Tagged objects. `http://msdn.microsoft.com/en-us/library/ee804823%28v=Surface.10%29.aspx`, accessed April 2011.

[Lik32] Rensis Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 22(140):1–55, 1932.

[LLL08] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. Comparing software metrics tools. In *ISSTA*, pages 131–142. ACM, 2008.

[LM10] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice. Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-oriented Systems.* Springer Verlag, 2010.

[Mar] Kevin Marshall. Beginning multi-touch on windows 7 & basic gesture recognition. `http://blogs.claritycon.com/kevinmarshall/`, accessed February 2011.

[Mar01] Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, TOOLS '01, Washington, DC, USA, 2001. IEEE Computer Society.

[Mar02] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, October 2002.

[Meh82] Nimish Mehta. A flexible machine interface. Master's thesis, Department of Electrical Engineering, University of Toronto, 1982.

[MGDM10] Naouel Moha, Yann G. Guéhéneuc, Laurence Duchien, and Anne F. Le Meur. Decor: A method for the specification and detection of code and design smells. *Software Engineering, IEEE Transactions on*, 36(1):20 –36, 2010.

[Mic] Microsoft. Microsoft surface case studies. `http://www.microsoft.com/surface/casestudies.aspx`, accessed January 2011.

[Mic07] Microsoft. The history of microsoft surface. `http://www.microsoft.com/presspass/presskits/surfacecomputing/docs/SurfaceHistoryBG.doc`, accessed January 2011, May 2007.

[MT04] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30:126–139, 2004.

[NDe] NDepend. Metrics definition. `http://www.ndepend.com/Metrics.aspx\#MetricsOnMethods`, accessed February 2011.

[New07] Laptop News. Microsoft surface diagram: How it all works. `http://www.cheaplaptops.org.uk/20070601/`, accessed March 2011, June 2007.

[Nos98] John T. Nosek. The case for collaborative programming. *Commun. ACM*, 41:105–108, March 1998.

[OCBZ09] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. In *Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on*, pages 390 –400, 2009.

[OCS10] Steffen Olbrich, Daniela S. Cruzes, and Dag I.K. Sjoberg. Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1 –10, 2010.

[PGFL05] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, pages 67–75, New York, NY, USA, 2005. ACM.

[Pin05] Martin Pinzger. *ArchView - Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, May 2005.

[PKS$^+$08] Peter Peltonen, Esko Kurvinen, Antti Salovaara, Giulio Jacucci, Tommi Ilmonen, John Evans, Antti Oulasvirta, and Petri Saarikko. It's mine, don't touch!: interactions at a large multi-touch display in a city centre. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1285–1294, New York, NY, USA, 2008. ACM.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipse in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[PW85] David L. Parnas and David M. Weiss. Active design reviews: principles and practices. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 132–136, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[Rela] Microsoft News Press Release. Microsoft and samsung unveil the next generation of surface. `http://www.microsoft.com/presspass/press/2011/jan11/01-06mssurfacesamsungpr.mspx`, accessed January 2011.

[Relb] Microsoft Press Release. Look what's surfacing at microsoft. `http://www.microsoft.com/presspass/features/2007/may07/05-29surface.mspx`, accessed January 2011.

[Relc] Microsoft Press Release. Microsoft launches new product category: Surface computing comes to life in restaurants, hotels, retail locations and casino resorts. `http://www.microsoft.com/presspass/press/2007/may07/05-29mssurfacepr.mspx`, accessed January 2011.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1st edition, April 1996.

[RO09] Neil Roodyn and Maridee O'Day. *Developing for Microsoft Surface*. nsquared, 2009.

[RSG99] Linda Rosenberg, Ruth Stapko, and Albert Gallo. Object-oriented metrics for reliability. In *IEEE International Symposium on Software Metrics. Engineering Institute*. Addison Wesley, 1999.

[Sai10] Shiori Saito. History of touch interfaces. Creative Environment for Emerging Electronic Culture (CE3C), Alberta College of Art and Design, June 2010.

[SDBP98] John Stasko, John Domingue, Marc H. Brown, and Blaine A. Price. *Software Visualization: Programming as a Multimedia Experience*. MIT Press, Cambridge, MA, 1998.

[Shn96] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, Washington, DC, USA, 1996. IEEE Computer Society.

[SMC99] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantive. Structured design. *IBM Systems Journal*, 38(2.3):231 –256, 1999.

[Sys] 3M Touch Systems. What is multi-touch? `http://solutions.3m.com/wps/portal/3M/en_US/TouchTopics/Home/Terminology/WhatIsMultitouch/`, accessed April 2011.

[TDDN00] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A meta-model for language-independent refactoring. In *Principles of Software Evolution, 2000. Proceedings. International Symposium on*, pages 154 –164, 2000.

[Tri05] Adrian Trifu. Diagnosing design problems in object oriented systems. In *In Proceedings of 12th Working Conference on Reverse Engineering (WCRE*, pages 7–11. Society Press, 2005.

[Tuf86] Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, Cheshire, CT, USA, 1986.

[vEM02] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, pages 97 – 106, 2002.

[vM77] Georg von Mayr. *Die Gesetzmäßigkeit im Gesellschaftsleben*. 1877.

[War00] Colin Ware. *Information visualization: perception for design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[WKCJ00] Laurie Williams, Robert R. Kessler, Ward Cunningham, and Ron Jeffries. Strengthening the case for pair programming. *IEEE Softw.*, 17:19–25, July 2000.

[WM08] Xin Wang and F. Maurer. Tabletop agileplanner: A tabletop-based project planning tool for agile software development teams. In *Horizontal Interactive Human Computer Systems, 2008. TABLETOP 2008. 3rd IEEE International Workshop on*, pages 121 –128, 2008.

[WWL07] Jacob O. Wobbrock, Andrew D. Wilson, and Yang Li. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 159–168, New York, NY, USA, 2007. ACM.

[WYCL08] Yanqing Wang, LI Yijun, Michael Collins, and Peijie Liu. Process improvement of peer code review and behavior analysis of its participants. *SIGCSE Bull.*, 40:107–111, March 2008.