Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

A Semi-automatic Approach to Code Smells Detection

Tiago Alexandre Simões Pessoa

Dissertação para obtencão do Grau de Mestre em Engenharia Informática

Orientadores: Prof. Doutor Fernando Brito e Abreu
Prof. Doutor Miguel Pessoa Monteiro

Setembro de 2011

(Page intentionally left blank)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado em Engenharia Informática

2º Semestre, 2010/2011

A Semi-automatic Approach to Code Smells Detection

Tiago Alexandre Simões Pessoa

Orientadores: Prof. Doutor Fernando Brito e Abreu
Prof. Doutor Miguel Pessoa Monteiro

Setembro de 2011

(Page intentionally left blank)

Acknowledgements

Fernando Brito e Abreu
Miguel Pessoa Monteiro
Sérgio Bryton
Mãe
Pai
Tanya
Sushi

(Page intentionally left blank)

Resumo

A erradicação de cheiros no código é frequentemente apontada como uma forma de melhorar a legibilidade, extensibilidade e desenho do software. Contudo, a detecção de cheiros no código continua a ser uma actividade consumidora de muito tempo e propensa a erros, parcialmente devido à inerente subjectividade do processo de detecção correntemente usado. Para mitigar este problema de subjectividade, esta dissertação apresenta uma ferramenta que automatiza uma técnica para a detecção e avaliação de cheiros no código fonte em Java, desenvolvida como um plugin Eclipse. A técnica é baseada num modelo de Regressão Logística Binária que usa métricas de complexidade como variáveis independentes e que é calibrado pelo conhecimento de peritos. É fornecida uma visão geral da técnica, a ferramenta é descrita e a sua utilização é validada por um caso de estudo de exemplo.

Palavras-Chave: Engenharia de Software Automatizada, Refabricações, Cheiros no Código, Avaliação Empírica, Métricas

(Page intentionally left blank)

Abstract

Eradication of code smells is often pointed out as a way to improve readability, extensibility and design in existing software. However, code smell detection remains time consuming and error-prone, partly due to the inherent subjectivity of the detection processes presently available. In view of mitigating the subjectivity problem, this dissertation presents a tool that automates a technique for the detection and assessment of code smells in Java source code, developed as an Eclipse plugin. The technique is based upon a Binary Logistic Regression model that uses complexity metrics as independent variables and is calibrated by expert's knowledge. An overview of the technique is provided, the tool is described and validated by an example case study.

Keywords: Automated Software Engineering, Refactoring, Code Smells, Empirical Evaluation, Metrics

(Page intentionally left blank)

Table of Contents

1 Introduction 1

1.1 Motivation 2

1.2 Context 3

1.3 Solution 3

1.4 Expected Contributions 4

1.5 Document Structure 4

2 Quality 5

2.1 Quality Models 6

2.2 Software Metrics 6

2.3 Empirical Evaluation 6

2.4 ISO 9126/IEC 7

3 Refactoring 9

3.1 Refactoring: Meaning and Context 10

3.2 Refactoring Process 11

3.3 Code Smells 13

3.4 Refactorings Format 17

3.5 Illustrative Example of a Refactoring Application 18

3.6 Refactoring Tools 19

4 Automated Code Smells Detection 20

4.1 BLR Model 21

4.2 Automation 23

4.3 Models of Use 25

5 Smellchecker 27

5.1 Eclipse 28

5.2 *Metrics Plugin* 31

5.3 Smellchecker Overview 33
5.4 Smellchecker Architecture and Implementation 36
5.5 Smellchecker Usage 43
6 Case study 54
6.1 Objectives 55
Problem Statement 55
Design Planning 56
6.2 Source Code Projects 56
6.3 Data collection and descriptive statistics 57
6.4 BLR Estimation 61
6.5 Conclusions 68
7 Related Work 69
7.1 Papers and Articles 70
7.2 Open Source Tools 75
8 Conclusions and Future Work 77
8.1 Closing Remarks 78
8.2 Threats to Validity 78
8.3 Future Work 79
9 Bibliography 81

List of Figures

FIG. 1: ISO 9126 QUALITY MODEL 8
FIG. 2: THE REFACTORING PROCESS 12
FIG. 3: EXTRACT METHOD EXAMPLE (ADAPTED FROM[33]) 18
FIG. 4: CODE SMELLS DETECTION PROCESS 23
FIG. 5: LOCAL USAGE 25
FIG. 6: **REMOTE USAGE** 26
FIG. 7 ECLIPSE SOFTWARE DEVELOPMENT KIT 28
FIG. 8: *AST* WORKFLOW [ADAPTED FROM HTTP://WWW.ECLIPSE.ORG] 30
FIG. 9: *METRICS* VIEW OF A METHOD METRICS 31
FIG. 10: SUMMARIZED METRICS AT PACKAGE LEVEL 32
FIG. 11: SMELLCHECKER'S COMPONENT DIAGRAM 37
FIG. 12: SMELLCHECKER PERSPECTIVE 44
FIG. 13: 'SMELLCHECKER: CODE METRICS' VIEW 47
FIG. 14: SMELLCHECKER PROPERTIES PAGE 47
FIG. 15: SMELLCHECKER PREFERENCES PAGE 48
FIG. 16: WARNING INVALID THRESHOLD SETTING 49
FIG. 17: SMELLCHECKER TOOLBAR 49
FIG. 18: SMELLCHECKER MENU ACTIONS 50
FIG. 19: WARNING NO PROJECT ELEMENT SELECTED 50
FIG. 20: INFORMATION OF THE TOTAL OF ANNOTATIONS FOUND IN THE CODE 51
FIG. 21: WARNING DATABASE NOT POPULATED 51
FIG. 22: ERROR R IS OFFLINE 52
FIG. 23: ERROR R COULD NOT COMPUTE BLR COEFFICIENTS 52
FIG. 24: SUCCESS BLR MODEL CALIBRATED 52
FIG. 25: JOPT *MLOC* BOXPLOT 58
FIG. 26: APACHE *MLOC* BOXPLOT 58

(PAGE INTENTIONALLY LEFT BLANK)

LIST OF TABLES

TABLE 1: MÄNTYLÄ BLOATERS 14
TABLE 2: MÄNTYLÄ ABUSERS 15
TABLE 3: MÄNTYLÄ PREVENTERS 15
TABLE 4: MÄNTYLÄ DISPENSABLES 15
TABLE 5: MÄNTYLÄ COUPLERS 16
TABLE 6: KERIEVSKY CODE SMELLS 16
TABLE 7: SAMPLE EXTRACT FOR CALIBRATING A *LONG METHOD* CODE SMELL ESTIMATION MODEL 22
TABLE 8: ***METRICS PLUGIN* 1.3.8 METRICS** 33
TABLE 9: *JOPT* AND *APACHE* DESCRIPTIVE STATISTICS 57
TABLE 10: JOPT *MLOC* STATISTICS 58
TABLE 11: *LONG METHOD* TAGGED INSTANCES 60
TABLE 12: SPEARMAN RHO CORRELATION TEST FOR JOPT 60
TABLE 13: SPEARMAN RHO CORRELATION TEST FOR APACHE 61
TABLE 14: BLR PREDICTION JOPT (THRESHOLD=0,8) 62
TABLE 15: JOPT BLR ESTIMATIONS ON JOPT 62
TABLE 16: BLR PREDICTION APACHE (THRESHOLD=0,8) 63
TABLE 17: APACHE BLR ESTIMATIONS ON APACHE 64
TABLE 18: JOPT BLR ESTIMATIONS ON APACHE 65
TABLE 19: APACHE BLR ESTIMATIONS ON JOPT 65
TABLE 20: SPEARMAN CORRELATION TEST ON AGGREGATED DATA OF THE TWO PROJECTS 66
TABLE 21: AGGREGATED BLR ESTIMATIONS ON JOPT 67
TABLE 22: AGGREGATED BLR ESTIMATIONS ON APACHE 68
TABLE 23: SMELLS DETECTION COMPARISON PART 1 73
TABLE 24: SMELLS DETECTION COMPARISON PART 2 74
TABLE 25: SMELLS DETECTION COMPARISON PART 3 74

(PAGE INTENTIONALLY LEFT BLANK)

Code Listings

LISTING 1: JAVA METHOD EXAMPLE 31
LISTING 2: PLUGIN.XML ECLIPSE'S POPUPMENUS CONTRIBUTION 40
LISTING 3: SMELL *ANNOTATION TYPE* 41
LISTING 4: SMELL *ENUM TYPE* 41
LISTING 5: *LONG METHOD* ANNOTATION 45
LISTING 6 - *LONG METHOD* AND *DUPLICATED CODE* ANNOTATION 46

(Page intentionally left blank)

1 Introduction

Contents

1.1 Motivation 2

1.2 Context 3

1.3 Solution 3

1.4 Expected Contributions 4

1.5 Document Structure 4

1.1 **Motivation**

As advocated by the agile XP methodology [1], refactoring techniques are sought to reduce costs associated with software life cycle at both the construction phase [2] and the production phase [2] by supporting iterative and incremental activities and also by improving software extensibility, understandability and reusability [3]. Taking into account that software maintenance activities are the most costly in the software life cycle [4], [5],[6], tangible benefits are expected from regularly performing refactoring. Empirical evidence showing the dire consequences of code infested with smells, seems to concur [7].

Even with an approach based on guidelines offered by Beck [1] and Fowler [3], the need of informed human assistance is still felt, to decide where refactoring is worth applying [8]. It is here that the concept of code smells makes a contribution [3]. Nevertheless, we have found, in the context of post-graduate courses, that the manual detection of code smells is an excessively time-consuming activity (therefore costly) and is error-prone, as it depends on the developer's degree of experience and intuition.

Empirical studies on the effectiveness of code smells detection techniques are still scarce, but there is some evidence that their eradication is not being achieved to a satisfactory degree, often because developers are not aware of their presence [9]. This is due to the lack of adequate tool support, which requires sound techniques for code smells diagnosis. The subjective nature of code smells definition hinders that soundness [3, 10].

Currently used code smells detection techniques come in two flavours. The first concerns qualitative detection using (inevitably biased) expert-based heuristics. The latter uses thresholds on software metrics obtained from the source code under analysis and seems more appealing for supporting automation due to its repeatability. However, it has two important preconditions for effective use. First, the same set of metrics cannot be used to detect all smells of a catalog such as the one in [3] since code smells are very distinct in nature. Second, even with a customized set of metrics chosen by an expert for detecting a particular smell, the resulting model must be calibrated, i.e., its internal values must be determined to reduce false positives and false negatives. That entails an empirical validation based on existing classification data. Mantyla et al. [11] confirm the difficulty of assessing code smells by using metric sets and the hard task of defining a detection model.

Even worse, studies of refactoring essentials such as smells detection and quality effects upon refactoring are still scarce, with little empirical evidence on both ends [9].
The main point is that even the most mature activity of the refactoring process (e.g. refactoring mechanics) is not fully automatic. Noting that the remaining activities of the process are considerably more debatable, much research is necessary for more systematic refactoring usage [8].

1.2 **Context**

The design and development process has a lot to gain by using assessment techniques to help effectively identify code smells, as well as helping to know when refactoring can be beneficial. Code smell identification can be used through intuition, according to the programmer or designer's degree of experience, but it is desirable to better support it through more objective and precise means (using metrics).
It is of paramount importance the usage of the right software metrics, since these represent the most effective way to supply empirical evidence, contributing to our understanding of the different dimensions of the software quality concerns. But the refinement of metrics to use for a particular code smell can only be obtained with experienced expert's knowledge and it is even suggested it might be impossible for some cases [3, 12].

1.3 **Solution**

This work contributes to the field of code smells detection by providing an automated process, supported by a tool (an Eclipse plugin), capable of code smell assessment in Java source code in an objective and automatic way. In contrast with existing proposals that rely purely on the opinion of a single expert, we propose a statistical based detection algorithm that will go through progressive calibration based upon a developers' community. The detection algorithm, based on *Binary Logistic Regression*, was initially calibrated by using a moderately large set of pre-classified methods (by human experts) and validated for the *Long Method* code smell, as depicted in Bryton et al. [10]. The larger the set, the better is expected to be the detection. This approach relies on the community of users to perform continuous recalibration of the code smells detection models (one per each smell).

A prototype version of the Smellchecker tool is presented: an Eclipse plugin for detecting code smells in Java code. This prototype allows smell tagging, visualization and detection. The assessment of this automatic process validity will be made using the *Long Method* code smell. On a second phase the process will be pushed further to test the same analysis process with other code smells to see if it is extendable.

1.4 **Expected Contributions**

The expected contributions of this work are the following:

- Automate an objective and semi-automatic technique for code smells detection based on *Binary Logistic Regression*;
- Further validate the soundness of that process for the detection of th *Long Method* code smell;
- Validate the process with other code smells detection;
- Improve metrics for code smells assessment;
- Extend the Eclipse IDE with a code smell detector tool.

1.5 **Document Structure**

This document has a chapter on a brief notion of Software Quality, followed by a detailing chapter about Refactoring. The description of our automated solution follows. Then the Smellchecker plugin is explained. A case study is presented next. Concluding the document with conclusions and future work and the bibliographical references.

2 Quality

Contents

2.1 Quality Models 6

2.2 Software Metrics 6

2.3 Empirical Evaluation 6

2.4 ISO 9126/IEC 7

Software development methodologies are used to enhance the quality of software products and to reduce its production costs [13]. Software quality assessment is a process in software development. Depending of the software development methodology in use, different aspects of the source code can be evaluated.

2.1 **Quality Models**

Quality models convey an overview of the set of characteristics that are indicative of product quality and the way of measuring them. These models are usually constructed in a tree-like structure with the main quality factors divided into a set of quality sub-factors or criteria that are easier to understand, quantify and measure. Actual metrics are proposed for the criteria [14].

2.2 **Software Metrics**

In Software Engineering metrics are used to collect data of entities regarding: the process of software production, its products and resources required [14]. Software metrics can measure internal or external attributes. Internal attributes of a product, process or resource can be measured solely on its own, separately from its behavior. External attributes, on the other hand, are measured taking into account how the product, process or resource relates to its environment. Its behavior is the focus.

Examples of internal attributes are the size of a software module in terms of lines of code; the complexity in terms of how many decision points exist in the code; or the dependencies among modules. This kind of attributes can be statically measured.

The number of failures experienced by the user, or the time it takes to retrieve an information from a database, are examples of external attributes and can only be measured at runtime.

2.3 **Empirical Evaluation**

Empirical evaluation is a key activity to enable us to obtain evidence and learn about the quality of our software artifacts. Empirical evaluation is a deep concern to researchers and practitioners in the area of software development.

Systematic assessment of software development techniques is imperative through all the software lifecycle phases, from requirements engineering to implementation and maintenance. For example, estimation models and measures of software internal

attributes are assessment techniques that can assist software developers, managers, customers, and users to characterize and improve the quality of code and products. As a consequence, assessment is a central issue to enable the effective transfer of new construction techniques to the mainstream of software development and to gain industrial attention for the new techniques. In [9] Bennett et al. point out to the need of a code smell assessment strategy and means to apply it. The work in this dissertation contributes to that end, since it aims at improving the code smell detection process by means of a quantitative based decision process, based on complexity metrics collected from source code.

2.4 **ISO 9126/IEC**

ISO/IEC 9126 [42] is an international standard for the evaluation of software. It builds on previous work done by McCall [15] and Boehm [16].

It defines a quality model which includes a set of primary characteristics under which the software should be evaluated. These characteristics are further refined in sub-characteristics comprising the totality of the internal and external quality model [43]. According to the standard, source code quality can be characterized by the following main characteristics: functionality, reliability, usability, efficiency, maintainability and portability.

ISO 9126 also addresses guidelines for the measurement of the characteristics of the quality model by using external metrics [45], internal metrics [44], and quality in use metrics [46].

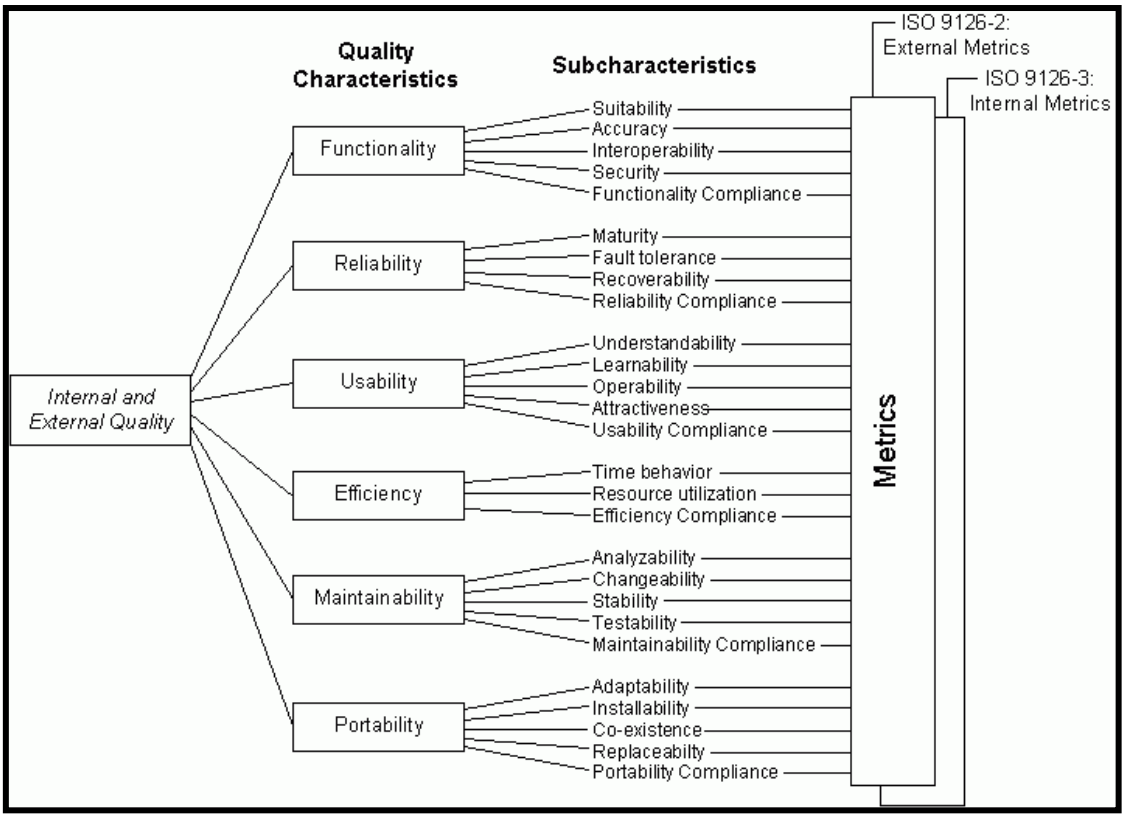


Fig. 1: ISO 9126 Quality Model

3 Refactoring

Contents

3.1 Refactoring: Meaning and Context 10

3.2 Refactoring Process 11

3.3 Code Smells 13

3.4 Refactorings Format 17

3.5 Illustrative Example of a Refactoring Application 18

3.6 Refactoring Tools 19

3.1 **Refactoring: Meaning and Context**

As the evolution of a software system causes continuous readjustments to its code base, its internal structure becomes more complex and the initial design may become progressively cluttered [3]. In that perspective Martin Fowler formalized a systematic approach to cleaning up, maintaining and expanding the design in a controlled way he defined refactoring [3].

The term refactoring was first introduced by Opdyke, meaning the process of applying code transformations to improve code's structure, style and design without altering its external behaviour [17]. Refactoring also stands for the sequence of steps in which a behaviour preserving transformation can be made possible.

For enforcing refactorings behavior preserving nature, Opdyke [17] introduced the notion of refactoring pre-conditions: invariants that must be true before applying a refactoring, with Roberts [18] extending the concept to refactorings post-conditions: invariants that remain true after refactoring application.

Since manual refactor is error-prone refactoring operates in small changes to guarantee preservation of behaviour. To this end every alteration should be carefully tested before being validated, hence the importance of good tests coverage and the need of a systematic approach for their use. Common tests associated with refactoring are unit tests. These tests concern single modules. In the context of refactoring, unit tests serve as regression tests.

Regression testing is a software testing practice that seeks to uncover bugs in existing functionality that may occur after functional or structural enhancements. Ensuring that code modifications do not invalidate previously accepted tests [19].

Fowler advises systematic use of refactoring whenever it facilitates the introduction of a new functionality, when bad design arises, and at code reviews [3]. Fowler and Kent Beck further evolved the concept with application guidelines and introduced the code smell notion which is the description of a symptom that might indicate a potential problem of poor structure or style in source code aimed for refactoring [3]. Eradication of code smells is known to improve code readability, improved implementation of Object Oriented concepts, improved code maintainability, changeability and improved extensibility [3].

Refactoring origins are linked to Object Oriented Development and an initial catalogue of refactorings for source code transformation can be found in Fowler's seminal work

on refactoring [3][1]. Kerievsy [20], fusing concepts of Gang Of Four's Design Patterns [21], extended Fowler's catalogue with specific code smells and refactorings to evolve the code to make use of design patterns.

Refactoring also found use by evolutionists occupied with a paradigm shift, Laddad [22] suggested refactorings to migrate Objected-Oriented systems into Aspect Oriented ones. Opening way to Monteiro and Fernandes initial catalogue [23] of refactorings and smells proper to evolve Object-Oriented source code to its Aspect aware counterpart.

Refactoring is also advocated by Beck as one of the founding design practices in the Extreme Programming (XP) agile methodology [1].

More recently, architectural refactoring focused in higher level design emerged as further prove to refactorings vitality [24]. In the reverse engineering field, experimental studies suggested that refactorings may help identify how and why a system changes over time [25].

3.2 **Refactoring Process**

To prepare the refactoring process for an automate reasoning we adapted Mens and Tourwé's refactoring process activities [8] to five main activities[2]:

1. Detect refactoring opportunities (*Code Smells Detection*);
2. Determine appropriate refactorings concerning each one of those opportunities;
3. Select refactoring(s) to apply;
4. Assess the effect of the refactoring in software or process quality characteristics;
5. Repeat the previous steps until no more opportunities to improve are detected.

[1] A revised and actualized source for this refactoring catalogue can be found in http://www.refactoring.com/catalog/index.html [6 September 2011].

[2] Note that every refactor is a behavior preserving operation by definition, so the activity of guaranteeing that the system remains functional unchanged is implied on the above steps. Also, the purpose of applying the refactoring process is to improve code's quality so the checks to guaranteeing that the quality was in fact improved are in the model.

This process is presented in Fig. 2. Note that a list of refactoring opportunities is produced as output from the first phase. The second phase associates a list of appropriate refactorings to each entry of the previous list. In phase three, a refactoring or a set of refactorings are selected from that list and applied. Next the quality is assessed based on precise criteria and a quantitative result is produced. The process iterates itself until no more refactoring opportunities arise.

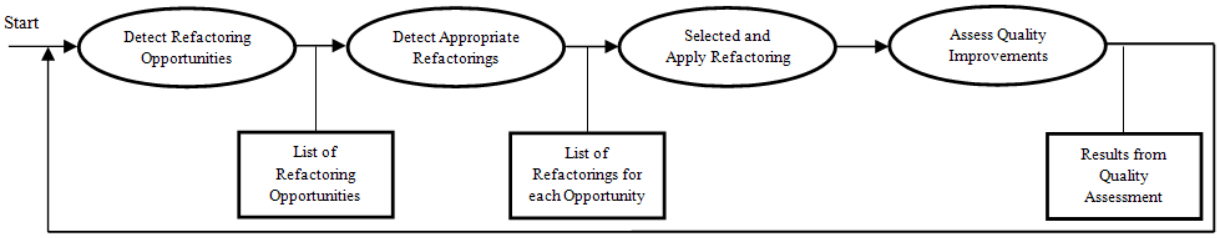


Fig. 2: The refactoring process

The process described is generic enough to be applied to different software artefacts - mainly design diagrams and source code.

The process can be analyzed at different levels of abstraction. At a higher level, refactoring opportunities are described as generalized code smells. Selected refactorings to remove these kind of smells are applied in a generalized way to every intervenient artefact on the process (e.g. at this level of abstraction artefact covers classes, methods or variables depending of the aim of the refactoring operation).

For example, considering the refactoring Rename Method aimed to remove a code smell[3] we could define as Long Name (i.e. symptom of an artefact name that may be too long). In this context various occurrences of the referred code smell may be presented throughout the code. The refactoring action must rename each one of them, resulting in the application of several instances of Rename Method to several instances of Long Name code smell.

At a lower level of abstraction more detail is specified and every particular instance of a code smell is identified. Regaining the previous example, the aim is not to remove the Long Name code smell from every artefact but instead remove a particular instance of that smell pertaining to a single artefact.

[3] In the context of chapter 4, code smell is used to identify a symptom of a design problem that may be present in any software artefact, source code or other.

For the purpose of this work a code smell is always an identifier for a refactoring opportunity. Refactoring is always a solution to a problem or at least a perceived problem. Refactoring opportunities are from here on referred interchangeably with the term code smells. The appropriate refactorings suggested to remove such smells can be derived from catalogues. Various authors identify code smells and propose appropriate refactorings to deal with them. The process of selecting which refactoring to apply from the list of different possibilities is the most problematic phase of the process. It is not yet clear how to accomplish that objective although works in the area have tackled the problem [26].

The assessment phase is dependent of the evaluation criteria used to measure the quality objectives for the system. In any case it should be formal and quantitatively based. The criteria used to measure the quality of the system under refactoring should also guide phase 3 of the refactoring process, aiding in choosing the best possible refactoring solution.

3.3 **Code Smells**

A code smell is the description of a symptom that might indicate a potential problem of poor structure in source code [3]. From the point of view of a programmer code smells are heuristics to indicate when and what to refactor.

One example of a bad smell identified by Fowler is *Duplicated Code*. The simpler refactoring for eliminating it is Fowler's Extract Method, where the *Duplicated Code* goes in a method of his own and it is invoked where is needed [3]. The procedures for dealing with such code smell are summarized by Fowler [3] in the following way:

- Identify where occurrences of the *Duplicated Code* are.
- If *Duplicated Code* is in different methods of the same class apply Extract Method and invoke the new created method from the places where the replicated code originally appeared.
- If *Duplicated Code* appears in sibling subclasses, apply Extract Method followed by Pull Up Method to parent class.
- If code is duplicated in unrelated classes, then the code belongs to only one of them and should be invoked by the others, or it may belong to neither of

the classes and should belong to a third class, either a pre-existing class or a entirely new one.

This process of *Duplicated Code* smell identification, provided by Fowler and Beck, simple amounts to: when you have identical expressions, either in the same class or some other class, probability suggests the *Duplicated Code* smell is in evidence and you should apply the respective refactoring.

In their seminal work Fowler and Beck announced 22 original bad code smells [3]. Coupled with guidelines distilling the context where they might constituted a problem they become symptoms for bad structuring of the source code. Refactorings suggestions were then presented to mitigate or completely resolve the stink. Although refactoring mechanics were meticulously explained, code smells suffered from the vague aesthetic its fancy name tried to hide. Expressed in natural language, the qualitative heuristics served as ambiguous, confusion-prone torches set to blaze the minds afire of practitioners, developers and common people alike as seen in the previous example.

In his book [12], Bill Wake revolved around resolving code smell problems. Providing a set of practical examples and the first glimpse of order, Fowler's proposed code smells were unequivocally divided in Smells Within Classes and Smells Between Classes. At the same year of 2003, as part of his Master Thesis [27], Mäntylä fathered in May a taxonomy to further expose smell's relationships, providing a better understanding of the concepts evolved to five groups.

Table 1: Mäntylä Bloaters

The Bloaters
Long Method *Large Class* Primitive Obsession Long Parameter List DataClumps
Represents something that has grown so large that it cannot be effectively handled. Primitive Obsession and Data Clumps are actually more of a symptom that causes bloats. When a Primitive Obsession exists, there are no small classes for small entities. Thus, the functionality is added to some other class

Table 2: Mäntylä Abusers

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

Tables Table 1, Table 2, Table 3, Table 4 and Table 5 detail Mäntylä code smells taxonomy. Respectively, from top to base: group name, smells in the group and reasoning behind the grouping. Note that all Fowler's original smells are represented except Comments and Incomplete Library Class that aren't included in any grouping. A new code smell named *Dead Code* is introduced by Mäntylä and it represents code fragments left behind. Pieces that were once used but currently not.

Table 3: Mäntylä Preventers

The Change Preventers
Divergent Change *Shotgun Surgery* *Parallel Inheritance Hierarchies*
Are smells that hinder changing or further developing the software. These smells violate the rule suggested by Fowler and Beck which says that classes and possible changes should have a one-to-one relationship. For example, changes to the database only affect one class, while changes to calculation formulas only affect the other class

Table 4: Mäntylä Dispensables

The Dispensables
Lazy Class *Data Class* *Duplicate Code* *Dead Code* *Speculative Generality*
The common thing for the Dispensable smells is that they all represent something unnecessary that should be removed from the source code

Table 5: Mäntylä Couplers

The Couplers
Feature Envy *Inappropriate Intimacy* *Message Chains* *Middle Man*
This group has four coupling-related smells. This group has 3 smells that represent high coupling. Middle Man smell on the other hand represent a problem that might be created when trying to avoid high coupling with delegation. Middle Man is a class that is doing too much simple delegation instead of really contributing to the application

Kerievsky [20] introduced the concept of refactoring source code to patterns. Updating Fowler's original catalogue with new code smells that possible indicate the introduction of patterns as a possible refactoring solution. Kerievsky also extended some of Fowlers smells to take in account that a possible pattern could be a solution. In Table 6 are all the code smells Kerievsky worked.

Table 6: Kerievsky Code Smells

Kerievsky Code smells	
Duplicated Code	*Indecent Exposure*
Long Method	***Solution Sprawl***
Conditional Complexity	*Alternative Classes with Different Interfaces,*
Primitive Obsession	***Lazy Class***
Large Class	*Switch Statements*
Combinatorial Explosion	***Oddball Solution***

Literature has various evidences of new code smells beeing suggested. In most cases they are merely reashes of Fowlers smells adapted to a particular domain: Van Emden and Moonen produced a new set of code smells qualitative heuristics specific to deal with test code [28], and suggesting not conformance to particular coding rules to be identified as code smells so as specific concrete language constructs (i.e. Typecast and Instanceof in Java) [29]. Another example is Dudziak and Wloka [30] with the definition of two new smells: Shared Collection (a field of a collection is modified via

getter and setter methods when the access to a collection field should be encapsulated) and Unnecessary Openness (a class has public and package-visible features that aren´t use outside of its scope).

A domain example specific to a paradigm jump is the work of Monteiro e Fernandes [23] where the concept of code smells is extended to support Aspect Oriented constructs.

An indicator for the presence of code smells in the source code could come from a higher perspective. Code smells as detailed by Fowler were suggested [31] to indicate the presence of anti-patterns [32]. Proposing that code smells could function as symptoms for the presence of design smells and the presence of anti-patterns the possibility of encountering Fowler's code smells at the source code level.

3.4 **Refactorings Format**

Fowler described the code transformations known as refactorings using the following format: name of the refactoring, summary that includes typical situations where the refactoring is needed and what it does, motivation that indicates why the refactoring is needed, mechanics that include a concise series of steps on how to apply the refactoring and examples that show a code example of the refactoring usage [3].

Two different refactorings are mentioned on the above solution for resolving the bad smell of *Duplicated Code*: Extract Method and Pull Up Method. Fowler provided a detailed description on the mechanics to perform Extract Method [3]:

- Select the code fragment to be extracted.
- Create a new method with a suggestive name indicating what it's responsible for.
- Copy the extracted code fragment from the source to the newly created method body.
- Inspect the extracted fragment for variables that are local in scope to the source method and make them parameters to the method or temp variables if they are only use within the extracted code.
- If any of those local variables are modified by the extracted code and needed outside of it they may be passed as return values.

- In the source method, replace the extracted fragment with a call to the new created method.
- Test the new solution.

The motivation for the usage of this refactoring is driven by the possibility of aggregating code that will be simpler to understand on its own method. It also increases the potential for its reuse and makes the higher-level methods more readable, providing a simple and descriptive naming convention for the new method is followed.

Another refactoring often used is Move Method and it is characterized by moving a method from one class to another. This type of action is indicated when a method uses, or will be using, more features of another class than the features of the class where it actually is.

3.5 **Illustrative Example of a Refactoring Application**

Symptom of the code smell *Duplicated Code* is observable in Figure 1 since the first three instructions from the methods 'void hopOverLong()' and 'void showOff()' are the same (left area code).

In Fig. 3, the result of applying the Extract Method refactoring to the duplicated instructions is displayed in the right code square. A method 'void popWheelie()' was created and the *Duplicated Code* moved to this new method. Calls to the new method replace the old *Duplicated Code* fragments.

```
void hopOverLog(){
     placePedalsAt(HORIZONTAL);
     rotatePedals(100);//pedal hard!
     liftHandlebars();
     rotatePedals(0);//stop pedaling
     liftRearWheel();
}

void showOff(){
     placePedalsAt(HORIZONTAL);
     rotatePedals(100);//pedal hard!
     liftHandlebars();
     rotatePedals(50);//ease up a bit
     changeExpression(Faces.GRIN);
}
```

```
void hopOverLog(){
     popWheelie();
     rotatePedals(0);//stop pedaling
     liftRearWheel();
}

void showOff(){
     popWheelie();
     rotatePedals(50);//ease up a bit
     changeExpression(Faces.GRIN);
}

void popWheelie() {
     placePedalsAt(HORIZONTAL);
     rotatePedals(100);//pedal hard!
     liftHandlebars();
}
```

Fig. 3: Extract Method Example (adapted from[33])

3.6 **Refactoring Tools**

Initial refactoring tools serve the main purpose of guarantying safe transformations to the code so the programmer don't have to retest the program at every single refactoring step [3].

The origins of systematic refactoring as a practice are linked to Smalltalk [3]. So it's no surprise the first proper tool for that end erupted in that niche circle. Known as the Refactoring Browser, the tool permitted a series of automatic refactorings interactively, in a safe and fast manner [33]. The tool gained wide acceptance once its functionalities were integrated in conjunction with the Integrated Development Environment (IDE) for Smalltalk the Smalltalk Browser [3].

Refactoring tools now cover all activities of the refactoring process. From smell detection tools to prototypes developed to discover the best refactoring when more than one can be of use [8]. Currently well known IDEs provide some refactoring support (e.g. Eclipse, NetBeans, IntelliJ, Visual Studio). However, this support is limited to semi-automatically applying a refactoring [8], with the developer still responsible for selecting the area of code he wishes to refactor and assist with additional information depending on the particular refactor the developer whishes to apply. IntelliJ is graced with extra functionalities when compared with other IDEs since it is capable of software inspection capabilities such as a code duplicates detection and visualization mechanism, and a *Dead Code* detection system.

Automated tools base their decisions on specific metrics oriented to detect structural bad designs or smells. Some tools use inside information (via metrics) to inspect for code smells evidence, leaving for the user the responsibility to decide which refactor to apply. Additionally more powerful tools try to indicate which refactor is the most indicated to eliminate the detect code smell [34].

Simmond and Mens [35] identified one fully automatic tool, named Guru, capable of restructuring inheritance hierarchies and refactoring methods of SELF programs [36].

A category of tools of software visualization exists to support developers identification of code smells [37-38].

4 Automated Code Smells Detection

Contents

4.1 BLR Model 21

4.2 Automation 23

4.3 Models of Use 25

4.1 **BLR Model**

This work main objective is the contribution of an automatic process capable of code smells assessment. The process here proposed is based on a technique first introduced in 2010 by Bryton et al.[10]. The authors suggested and demonstrated the validity of a process adjusted for code smells detection that is objective, deterministic and possible of automation. The process was built around the idea of utilizing the power of statistical techniques to obtain a mathematical model able to detect *Long Method* instances upon source code analysis.

The approach relied on a statistical regression technique, where the independent variables that explained the model could be drawn automatically, thus providing an opening for automation. The dependant variable was in turn gained by directly accessing expert's knowledge.

Before covering any more details on the process an explanation of the Binary Logistic Regression Model ensues.

Binary Logistic Regression

Binary logistic regression (BLR) is used for estimating the probability of occurrence of an event (here, the existence of a code smell) by fitting data to a logistic curve. It is a generalized linear model where the dependent (aka outcome) variable has two possible values (code smell present or absent) and an arbitrary set of numeric independent (aka explanatory) variables can be used (a set of code complexity metrics). The general equation of the logistic function used to estimate the percentage of probability of a particular code smell is the following:

$$f(z) = \frac{1}{1+e^{-z}} \quad \wedge \quad z = \beta_0 + \beta_1 \times x_1 + \beta_2 \times x_2 + \ldots + \beta_n \times x_n .$$

Where z is called the logit, x_k are the regressors or explanatory variables (code complexity metrics collected from the source code) and β_k are the regression coefficients calculated during the calibration process. The usefulness of the BLR model is that allows the ranking its results by probability.

To perform BLR calibration with a statistical tool such as *SPSS* or *R*, a sample with values for all variables (explanatory and outcome) is needed. Table 7 presents an extract

of such a sample, corresponding to four methods on the *org.apache.commons.cli* package from *Apache Commons CLI 1.2*. The collected metrics are ***MLOC*** (method lines of code), ***NBD*** (nested block depth), ***VG*** (cyclomatic complexity), and ***PAR*** (number of parameters). These are the explanatory variables in the BLR model. ***Long Method*** is the dependent variable: an expert indication of the presence of the *Long Method* code smell on the particular method.

After calibration and validation of the regression coefficients, the instantiated model is used to predict the possible presence of a particular code smell.

Table 7: Sample extract for calibrating a *Long Method* code smell estimation model

Application	ApacheCommonsCLI1.2			
Package	org.apache.commons.cli			
Class	GnuParser	Parser	HelpFormatter	PosixParser
Method	Flatten	parse	renderOptions	burstToken
MLOC	69	67	59	46
NBD	5	5	4	4
VG	11	14	10	6
PAR	3	4	5	2
Long Method	0	1	1	0

Process Details

To use the BLR model, first we have to calibrate it to fit the data to a logistic curve that mimics real occurrences. So a sample collecting all variables for the model is the first step of the process.sda

The explanatory variables to be used are code complexity metrics and can be obtained automatically by means of software applications. The choice of the adequate metrics to select for each code smell estimation model based on BLR can be performed by using the Wald or the Likelihood-Ratio tests.

The outcome variable is added to the model by domain experts (code smells *connoisseurs*). The quality of these assessments and the quantity of data provided will contribute for the model accuracy.

Next is the proper calibration of the BLR model using mathematical calculus to derive the regression coefficients for the model. This can be done by a statistical engine (SPSS or R for instance). Final step is testing the model for goodness-of-fit, and the application of its estimations to predict code smell instances in selected code bases.

The described logic is the platform on which the process proposed in this work builds up. Developments required for its automation and the model of use expected to guarantee results and usability of the concept are unveiled in the following sections.

4.2 **Automation**

Fig. 4 outlines how the BLR model approach to code smells detection can be sequenced to evolve it to automatic processing. Note the necessity of input by domain experts at the start of the cycle. Although it is a limitation for automation purposes, its necessity can be reduced and favorable conditions can be offered to ease this action, not dismissing the fact that the power of this process and expected quality improvements to the model rely on human effort.

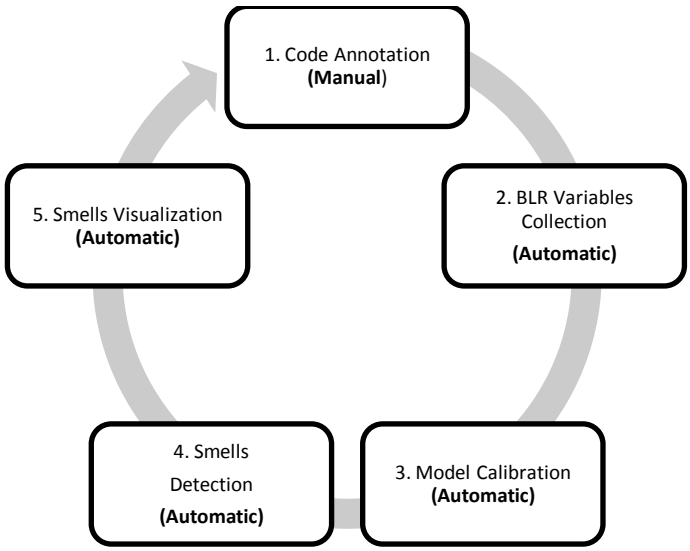


Fig. 4: Code smells detection process

Since one of the goals is to provide tool support of the activities in Fig. 4, we start by grounding this approach to more concrete terms, narrowing its scope the following way:

Code Annotation. In the first iteration, experts must tag the code sample for the presence of code smells in methods, classes or interfaces. So to yield an adequate

sample for the initial calibration of the models. In subsequent iterations those developers will only tag false positives (developer disagrees with a detected smell) and false negatives (developer identifies a non-detected code smell). These cases are expected to decrease over time as more data results in more finely calibrated models that produce more precise results (and more in sync with the user tastes). The finer the results less is the necessity for the users to calibrate the model. It is expected that the quantity of calibrations decrease over time, with the necessity for recalibration efforts only being called for when the user starts perceiving the model as inaccurate.

BLR Variables Collection. Automatic process that requires a parser-enabled tool that computes metrics on the target source code (the one annotated by domain experts). More metrics means more predictable variables for the BLR model, and better chances at estimating valid code smells predictions models. Because not all code smells are created equal, so is the need for different metrics to evaluate each of them. Information on code smell presence are taken from the annotations.

Models Calibration. Calibration of the BLR models by calculating and validating the regression coefficients. It is an automatic process performed by a statistical processor. Note that there will be one model for each code smell. Each model may have different explanatory variables (metrics) and it is up to the end user to give the final call about their validity, providing feedback to better attune it to his needs.

Smells Detection. Application of the calibrated BLR models to selected source code elements. This estimates the probability of presence of the corresponding code smell in the selected artifact.

Smells Visualization. Identification of the source code artifacts where code smells are estimated to be present. Developers can set the threshold probability (e.g. see only the code smells above 90% probability) for each code smell. Detected code smells will vary depending on the selected probability threshold. Increasing the probability too much will cause more false negatives, while decreasing it in excess will cause more false positives. It will be up to the developer to fine tune the threshold to get the adequate level of advice (let us call it “sensitivity”) regarding the presence of code smells. It will also be up to the developer to decide on the adequacy of applying a given refactoring to remove a detected code smell.

The described process can have two different usage patterns. The first concerns a single user with a single machine. The second, a remote usage for more than one user. Next section exemplifies them.

4.3 **Models of Use**

Local Usage

In this case the process (including the calibration) is completely local (Fig. 5). The user is responsible for tagging an initial source code base to calibrate the models. Then, the user can apply the models to detect the occurrence of code smells in all code bases of his choice. It is also possible to refine the calibration of the model by providing additional code smell tagging information.

The usefulness of this option is one of practical value: tuning the models, through progressive calibrations, to personal user preferences, thus matching the models to the user notions of where a code smell might be present.

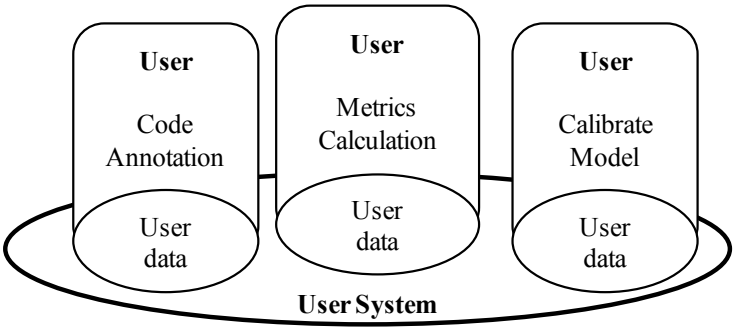


Fig. 5: Local usage

Remote Usage

The process has a remote central server responsible for storing, on its own data base, the code smells tagging and metrics values provided by several users (Fig. 6). With the calibration and validation of the BLR model being performed on the server, users can remotely query the server for the most recent model parameters. This model is calculated from the aggregated data provided by all users, augmenting the statistical significance of the BLR estimates and thus providing a more accurate detection of the code smells.

One of the simpler updates the user can make is to provide feedback on false positives and false negatives detected, thus contributing for the models' progressive enhancement.

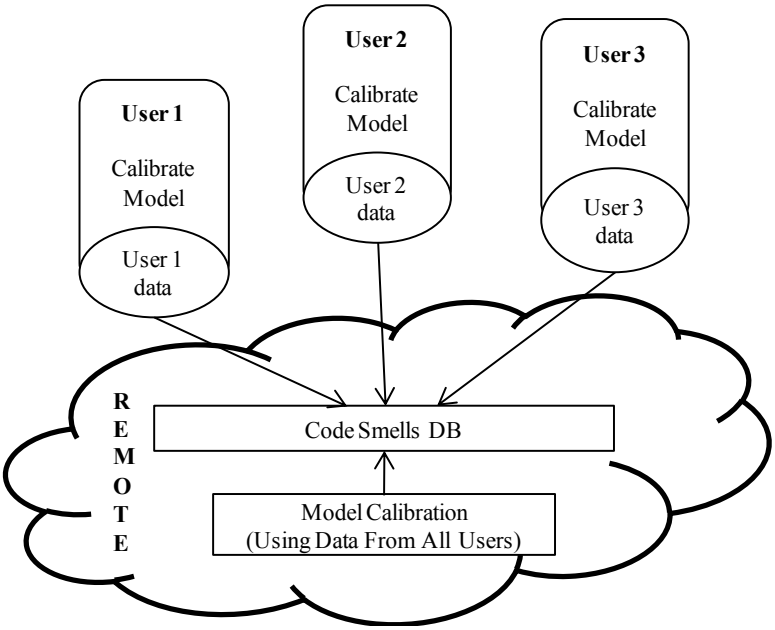


Fig. 6: **Remote usage**

We will further define the boundaries and scope of our detection process by providing a concrete tool to exemplify its usage. The tool, called Smellchecker, will be described in the next section.

5 Smellchecker

Contents

5.1 Eclipse 28

5.2 *Metrics Plugin* 31

5.3 Smellchecker Overview 33

5.4 Smellchecker Architecture and Implementation 36

5.5 Smellchecker Usage 43

This chapter describes the development and implementation of the code smells detection tool Smellchecker. Developed as an Eclipse plugin it was designed to bring code smells detection capability to Eclipse's Java Development Tools (JDT).

This chapter gives details of Smellchecker's architecture and implementation and examples of use. First it starts with a description of the Eclipse Software Development Environment (IDE), its plugin architecture and the concepts behind JDT's source code manipulation with *Java Model* and the *Abstract Syntax Tree*. Because Smellchecker's extensively use *Metrics Plugin* 1.3.8 functionalities a section detailing it precedes a Smellchecker's plugin overview. Detailed architecture and implementation specifications are given next. Ending this chapter is an exploratory example of Smellchecker's usage.

5.1 **Eclipse**

Eclipse is an open platform. It was designed to be extensible. At the core is the Eclipse Software Development Kit (SDK), with which we can build various tools. These products or tools can further be extended by other tools. For example, a simple text editor can be extended to create a XML editor. Eclipse's extensibility is achieved by creating these products/tools in form of plugins. Fig**. 7** shows the principle components within Eclipse´s SDK.

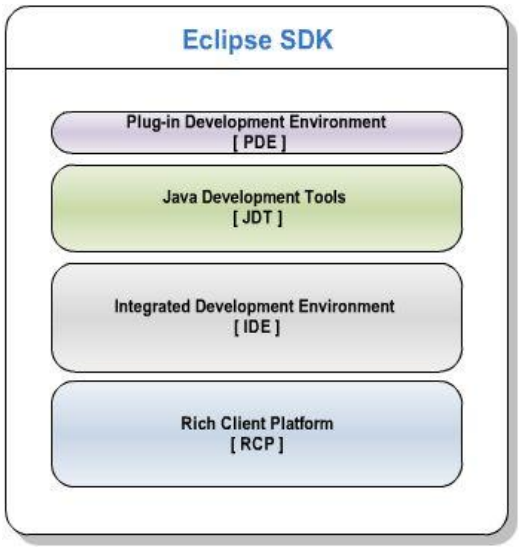


Fig. 7 Eclipse Software Development Kit

Rich Client Platform (RCP) provides the architecture and framework to build any rich client application. Integrated Development Environment (IDE) is a tools platform and a rich client application in itself. It is possible to build various kinds of tools by using the Eclipse IDE. Plugin Development Environment (PDE) provides all tools necessary to develop plugins and RCP applications. Java Development Tools (JDT) is a complete Java IDE that provides APIs to access and manipulate Java source code. It also permits the creation of new projects and handles modifications to existing ones, providing the tools to build and launch Java programs.

Plugins

Eclipse is built upon the OSGI framework (Equinox) . The OSGi framework provides a dynamic modular architecture in which bundles can be deployed (Eclipse uses the term plugins). Eclipse plugins are the same as OSGi bundles and are used to extend the Eclipse framework. Eclipse isn't a single Java program, but a small program which provides the functionality of typical loader called plugin loader.

A plugin is a Java program which extends the functionality of Eclipse in some way. Each Eclipse plugin can either consume services provided by other plugin or can extend its functionality to be consumed by other plugins. These plugin are dynamically loaded by Eclipse at run time on an on-demand basis.

When a plugin wants to allow other plugins to extend or customize portions of its functionality, it will declare an extension point. The extension point declares a contract (typically a combination of XML markup and Java interfaces), that extensions must conform to. Plugins that want to connect to that extension point must implement that contract in their extension. The key attribute is that the plug-in being extended knows nothing about the plug-in that is connecting to it beyond the scope of that extension point contract. This allows plug-ins built by different individuals or companies to interact seamlessly, without their knowing much about one another besides the extension point contract.

A plugin consists of a *bundle manifest file*: MANIFEST.MF that provides important details about the plug-in, such as its name, ID, and version number. The manifest tells also what Java code it supplies and what other plug-ins it requires. A plugin may provide code, documentation, resources bundles, or data to be used for other plugin.

A plugin can also provide a *plugin manifest file*: plugin.xml. It describes how it extends other plugins, or what capabilities it exposes to be extended by others (extensions and extension points).

Java Model and *Abstract Syntax Tree*

Java's *JDT* allows accessing Java source code in two different ways: either by using the *Java Model* or by using the Abstract Syntax Tree (*AST*). Where *Java Model* is a light-weight and fault tolerant representation of the Java project. It does not contain as many information as the *AST* (e.g. it does not contain the main body of a method) but is fast to re-created in case of changes. Eclipse's outline view uses the Java model for its representation, this way the information in it can quickly be updated.

The *AST* is a detailed tree representation of Java source code. The *AST* defines API to modify, create, read and delete source code. Each element in the Java source file is represented as a subclass of *ASTNode*. Each specific *AST* node provides specific information about the object it represents. For example you have *MethodDeclaration* (for methods), *VariableDeclarationFragment* (for variable declarations) and *SimpleName* (for any string which is not a Java keyword). Fig. 8 shows the overall *AST* workflow and how it relates to file source code.

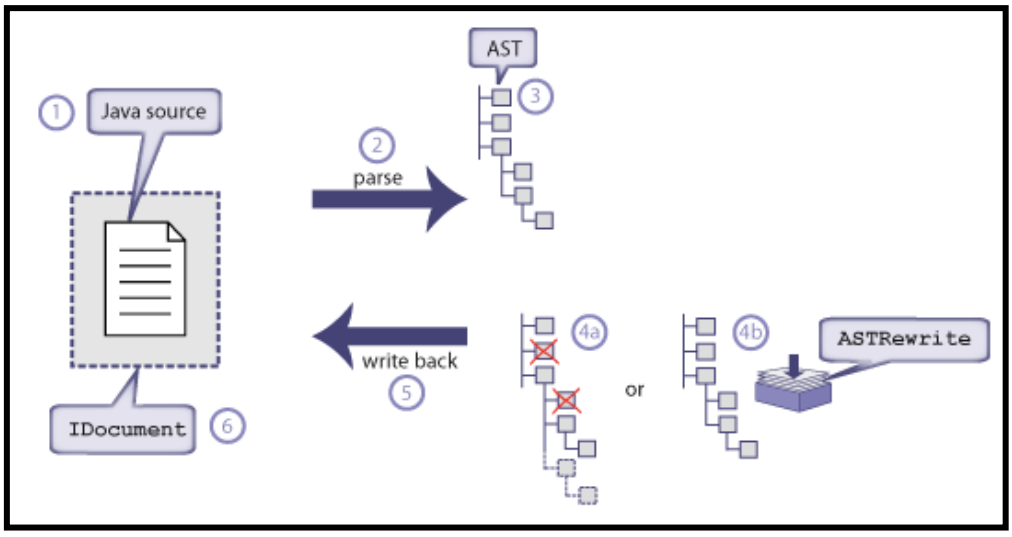


Fig. 8: *AST* workflow [adapted from http://www.eclipse.org]

5.2 *Metrics Plugin*

Java Metrics 1.3.8 provides metrics calculation and a dependency analyzer plugin for the Eclipse platform. Our main interest is on the metrics provided. Yet, we decided to keep all *Metrics Plugin* functionalities because they provide for a good analysis. Even if Smellchecker's cannot understand the dependency analyzer visioning, still Smellchecker's users may find it useful to check dependencies between modules.

```
private void arrangeOptions( Collection<String> unarranged ) {
    if ( unarranged.size() == 1 ) {
        options.addAll( unarranged );
        return;
    }

    List<String> shortOptions = new ArrayList<String>();
    List<String> longOptions = new ArrayList<String>();

    for ( String each : unarranged ) {
        if ( each.length() == 1 )
            shortOptions.add( each );
        else
            longOptions.add( each );
    }

    sort( shortOptions );
    sort( longOptions );

    options.addAll( shortOptions );
    options.addAll( longOptions );
}
```

Listing 1: Java method example

Metrics resources provide a good compliment to code smells tagging operation. Users can check metrics of a particular software element while annotating the code, this may provide useful to make more informed decisions. For example, when wondering if the code on Listing 1 should be annotated for the *Long Method* presence, the user may check for metrics values to assist him by looking at the *Metrics* view on Fig. 9.

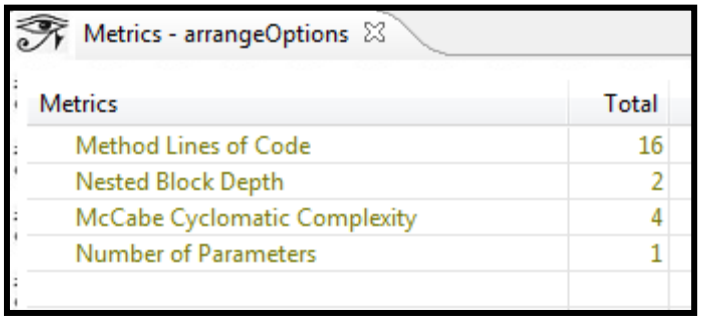
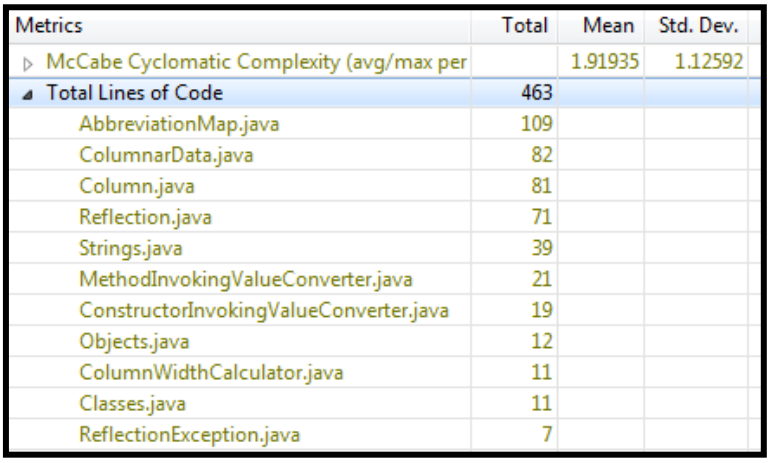


Fig. 9: *Metrics* view of a method metrics

Metrics provides summarized information on metrics calculated at the level of the components: package and system. This measures are shown with average and standard deviation calculation. This is important because the aggregated info can be drilled down within the view to decompose it to its basic elements. For instance, it means that the *Method Lines Of Code* (*MLOC*) metric is summarized (sum aggregation) at the package level and with a click on the metric name within the view, information of *MLOC* of all classes within the package will show. Information we can further drill down in the same view by choosing to see the methods metrics.

An example is seen in Fig. 10, where the metric *MLOC* at package level is drilled down to the compounding classes information. This can facilitate code inspection for code smells, allowing the user to check metrics organized by their value. Spotting on the example given in Fig. 10 the classes with more *MLOC* within a package/system component for further analysis.



Metrics	Total	Mean	Std. Dev.
▷ McCabe Cyclomatic Complexity (avg/max per		1.91935	1.12592
◢ Total Lines of Code	463		
AbbreviationMap.java	109		
ColumnarData.java	82		
Column.java	81		
Reflection.java	71		
Strings.java	39		
MethodInvokingValueConverter.java	21		
ConstructorInvokingValueConverter.java	19		
Objects.java	12		
ColumnWidthCalculator.java	11		
Classes.java	11		
ReflectionException.java	7		

Fig. 10: Summarized metrics at package level

Metrics provide other views and visualization capacities, more information on the *Metrics Plugin* may be found online[4].

Most important for Smellchecker is the capacity of calculus of various metrics types for Java source code elements.

[4] Metrics Plugin: http://metrics2.sourceforge.net/

Table 8 displays all the metrics available on *Metrics Plugin*, and this is the ones Smellchecker will use. Details of the metrics names, with the kind of element of a Java project they can be measured on, are in the table.

Table 8: ***Metrics Plugin* 1.3.8 metrics**

Metric	Acronym	Type	Method	Package/System
Number of Methods	NOM	**X**		**X**
Number of Fields	NOF	**X**		**X**
Total Lines Of Code	TLOC	**X**		**X**
Method Lines Of Code	*MLOC*	**X**	**X**	**X**
Number of Parameters	PAR		**X**	
Specialization Index	SIX	**X**		
McCabe Cyclomatic Complexity	VG	**X**	**X**	
Weighted Methods per Class	WMC	**X**	**X**	
Lack of Cohesion of Methods	LCOM	**X**		
Afferent Coupling	Ca			**X**
Instability	I			**X**
Abstractness	A			**X**
Normalized Distance from Main Sequence	Dn			**X**

5.3 **Smellchecker Overview**

Java was the language chose for the tool implementation, and it is also upon Java source code that the tool will detect code smells instances. That is because Java is a modern, well known, all purpose programming language, with many open source available software solutions we could use for aiding in implementing the Smellchecker tool.

Eclipse framework was selected as the target platform to support Smellchecker's development due to its advanced Java support, available refactoring features, along with its plugin development facility. Eclipse is also a stylish and appropriate choice for our tool deployment (as a plugin) since its architecture by components supports integration of virtually any component within its architecture. And yet, despite its advanced support for Java source code refactoring, as part of its standard JDT toolkit, code smells detection is by and large completely lacking (2005's *Code Nose Plugin*[39] is the known exception, although a version of it seems to not be available online for testing or using purposes).

Smellchecker was developed as an Eclipse plugin using Eclipse's Plugin Development Environment (PDE). The Smellchecker prototype architecture uses Java 1.6 and it was built upon Eclipse platform 3.5.

In order to concentrate our efforts on the specialization aspects of the code smells detection process a mechanism that provided code metrics calculations was necessary. Although there are various tools that perform metrics calculations on Java source code, few of them provided the capability of exporting its calculations in a way that could be used for automation. So we focused on open source projects that provided free access to their source. We tried *CyVis*[5] 0.9.0, *Dependenct Finder*[6] 1.2.1, *State of Flow Eclipse Metrics*[7] 3.14, and *Eclipse Metrics*[8] 1.3.8.

The combination of quality and quantity of metrics, combined with its nature as an Eclipse plugin, made *Eclipse Metrics Plugin* 1.3.8 the natural and most appropriated choice for incorporation within our tool. And comparing with State of *Flow Eclipse Metrics 3.14*, *Metrics 1.3.8* was the best documented version of the two.

Smellchecker's extends the source code of the *Eclipse Metrics 1.3.8* plugin for the purpose of supporting the functionalities required for the code smells detection activity. It leaves *Metrics* components almost unchanged, only making a minor modification to its activator class. All other contributions are on Smellchecker's own packages and the only resource used at source level from *Metrics* is a static class that returns the metrics calculations for a Java element passed as input.

There were two main reasons why the choice was made for directly extending *Metrics Plugin* source code. First of them is that the plugin does not implement an extension point that provides access to elements metrics. It only provides extensions to add new metrics and to add new export type files. The other is that it would not be desirable to burden the user with the necessity of exporting the metrics calculations from *Metrics* and feed them to Smellchecker. Also, the computational burden would be elevated by the necessity of parsing an entire file every time metrics were needed. Even if we started by reading *Metrics* entire project metrics calculations only once in the beginning of the process, every time any change occurred in the project's source code would mean the repetition of the process (and a heavy one). A better solution is to make the necessary

[5] http://cyvis.sourceforge.net/
[6] http://depfind.sourceforge.net/
[7] http://eclipse-metrics.sourceforge.net/
[8] http://metrics2.sourceforge.net/

modifications to the *Metrics Plugin* source code (that is why the project's source code is made available) and make the process automatic.

The cleaner and transparent way of giving Smellchecker's plugin direct access to *Metrics'* metrics calculations, would be by adding to *Metrics* an extension point to permit direct metrics access, and then making Smellchecker implement an extension for it. This was not our solution however. Ours was to directly change *Metrics* and add the new behavior to the source code itself. This decision comes from the fact that in the future, new metrics must be added to Smellchecker's. And as it is, *Metrics* extension point that permits adding new metrics, makes restrictions on the type of calculations it does, namely that they must be numeric. So, *Metrics* core code itself must be updated to achieve higher flexibility on the metrics it processes. Also, to deal with *Metrics* source code was to have a good head start to know how it operates and how to modify it.

The metrics supported by Smellchecker are the ones *Metrics* provides. Was behind the scope of this work to extend them, so the BLR explanatory variables we can use are confined to Metrics measurements. Which constraints the number and efficiency of code smells that can be detected. For instance, we could test to relate *Lines of Code* (LOC) with the *Duplicated Code* smell. In fact one could argue that *Long Method*s have a high probability of having *Duplicated Code* in them. That may be true. But it is also true that even a method with one line of code could be duplicated. I happens all the time when different classes implement the same basic functionality because they are unaware of other classes implementations. So for *Duplicated Code* more efficient methods already exist [40] than trying to support their detection with Smellchecker's current available metrics. That is not saying that the BLR approach Smellchecker uses is not powerful enough for some smells, but metrics should be adjustable to a specific code smell, and for the current moment we only can use in Smellchecker's a restricted subset of them.

Smellchecker's is fully integrated on the Eclipse IDE. Source code annotation will be permitted by accessing the appropriate option of code smell tagging on the context menu for both classes and methods entities. The tagging will be evident on the source code in the form of Java annotations. Java annotations were the choice for marking the code because they are native to the Java language. Although they are not guaranteed to always have support (some java compilers may delete annotations information), Eclipse's JDT makes sense of them and they are the only way to express outside information on Java source code using the standard Java language. The other choice at

our disposable was by using Eclipse's Markers, but that way it would be confined to the Eclipse framework and the annotation information would have to be saved outside the Java files to guarantee persistence. Java annotations are visible in the source code and the files can be parsed by another tool to get the annotation information.

In terms of statistical processing Smellchecker*'s* first and only choice was *R*[9]. *R*'s powerful statistical engine was even more appealing because of its open source nature. BLR coefficients model calculation activity will be dynamic and to the user is required have the statistical *R* tool installed. Following sections will detail the components development and integration and it will be clear what is expected from R and how the communication between Smellchecker and *R* will proceed.

Interaction with Smellchecker's users is made trough Eclipse's Workbench facilities and main Java text editor.

Details of Smellchecker's architecture and implementation solutions follow in the next section.

5.4 **Smellchecker Architecture and Implementation**

The development of the proposed code smells detection solution can be agruppated by functionalities (traceable to the activities described in Fig. 4) that support:

- Source Code Annotation (Java annotations)
- BLR Variables Collection (Eclipse *Metrics Plugin*)
- BLR Calibration (R communication)
- Code Smells Detection (source code parsing)
- Code Smells Visualization (Eclipse JDT)

An overview of all the major components comprising Smellchecker is provided in Fig. 11.

[9] R: http://www.r-project.org/

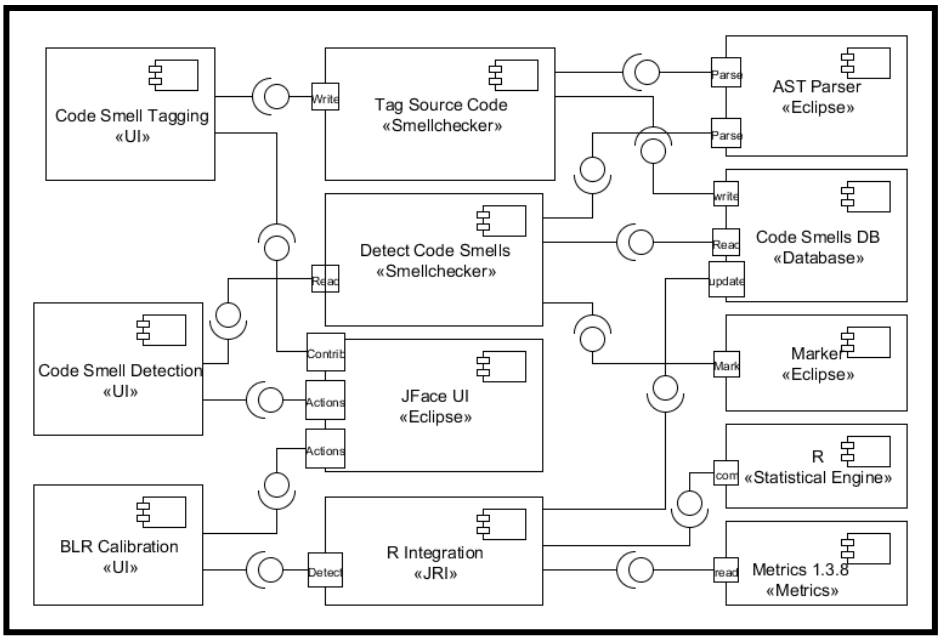


Fig. 11: Smellchecker's component diagram

User's interactions with the application are made through Eclipse's workbench elements and they are associated with three main user interaction components: *Code Smells Tagging «UI»*, BLR *Calibration «UI»* and *Code Smells Detection «UI»*. This are all associated with the *JFace UI «Eclipse»* component that is responsible for the actual UI interaction.

To each of this components corresponds a action phase of the detection process that requires user interaction. *Code Smells Tagging «UI»* deals with the Source Code Annotation, BLR *Calibration «UI»* calibrates the BLR model, and *Code Smells Detection «UI»* is associated with the Code Smells Detection and Visualization functionalities previous stated. In order to perform each action required by users, internal components are responsible for the dynamics of the process.

The *Code Smells Tagging «UI»* interacts with the *Tag Source Code* «Smellchecker» component to write annotations to the code. *Code Smells Tagging «UI»* is comprised of a set of Eclipse's user interface environment (UI) workbench popup menus actions necessary to annotated the code. It also receives the appropriated action events raised by the user and calls the appropriate function reasoning on the *Tag Source Code* «Smellchecker» component that is responsible for updating the source code. Being this last component responsible for writing or deleting annotations to the code.

BLR *Calibration «UI»* has a set of menu choices responsible for receiving user calls for retrieving the variables necessary to the BLR model and its calibration. When the action is to retrieve the BLR variables call to the component *Parse Code Variables* «Smellchecker» are made. When it is for the proper BLR model calibration it interacts with the component *RIntegration «JRI»* that is responsible for communicating with *R* application (*R «Statistical Engine»* for statistical processing and the retrieval of the calibrated coefficients. *Parse Code Variables* «Smellchecker» is responsible to parse the code and get the information from the Java code smells Annotations and the metrics calculations from the *Metrics 1.3.8 «Metrics»* component.

The *Code Smell Detection «UI»* is called when the user gives order to detect the code smells in the code with the BLR calibrated model. It deals with component *Detect Code Smells* «Smellchecker» responsible for applying the BLR estimation to the code elements in search for smells confirmation and then annotate the code when applied.

Common to all «Smellchecker» components is the *Persistence«DB»* component. Where the global information is stored and retrieved.

Now we will detail each of the components, separating them in sections regarding their actions: UI construction, Source Code Annotation, Metrics Integration, BLR calibration, Code Smells Detection, and Code Smells Visualization.

Code structure

Smellchecker*'s* source code are organized in seven different packages:

```
package com.tp.refactoring.smellchecker.persistence;
package com.tp.refactoring.smellchecker.codesmells;
package com.tp.refactoring.smellchecker.regression;
package com.tp.refactoring.smellchecker.rinterface;
package com.tp.refactoring.smellchecker.smellsupdater;
package com.tp.refactoring.smellchecker.ui;
package com.tp.refactoring.smellchecker.ui.preferences;
```

Persistence package has the Smellchecker*FileManager*() *Class* that mediates the creation of files and reads and writes in order to preserve metrics information. While most files are created dynamically during the plugin operation, two files keep persistent data regarding all variables calculated to the moment for classes or methods. This package also comprises a *Class* (*Initializer*()) responsible for keeping all symbolic data information during run time, for instance the code smells identifiers, metrics names and

other constants that Smellchecker classes need to utilize and that it can obtain only from the *Initializer*() *Class.*

C*ode smells package* has the classes that represent the annotated elements in the code, as well as the metrics values for classes and methods. The *regression package* has the classes that comprise the BLR model, its coefficients and calculation. *Package rinterface* communicates with the *R* engine and it is responsible for feeding it with the appropriated commands and retrieve the BLR model coefficients. The *smellsupdater* is the set of classes that transverse the code and retrieves metrics and source code annotations and call the appropriated function in *persistence* to record the variables. The *ui package* has all the elements that comprise the views of the system and the elements for their updates. Package *preferences* is capable of setting the plugin preferences and property pages.

UI construction

Graphic elements on the Eclipse are constructed when the workbench is first created (on Eclipse loading). When Eclipse's reads Smellchecker's *plugin manifest file* (*plugin.xml*) it knows that this plugin contributes to the UI and that its contributions to the workbench rely on a entry to the menu bar ('Smellchecker' menu), a toolbar with actions specific to Smellchecker, popup menus that are element context sensitive, and a set of views for visualization purposes. When Eclipse first loads only the graphical display of the contributions is added to the workbench. The entries taking the form of a proxy object that are only initialized upon user action on them.

Extensions to the popUpMenus are elements of the *Code Smells Tagging «UI»* component. One of Smellchecker's contribution to Eclipse's *popupMenu* is detailed in Listing 2.

```
<extension point="org.eclipse.ui.popupMenus">
      <objectContribution
            objectClass="org.eclipse.jdt.core.IMethod"
            id="...method">
      <menu
            label="Tag Smell"
            path="...menu"
      (...)
<action label="Long Method"
      menubarPath="..."
      class="...CodeSmellsTag"
      id="LongMethod">
</action>
```

Listing 2: Plugin.xml Eclipse's popupMenus contribution

This extension point is element sensitive and contributes only to objects of the type *IMethod*, meaning that wherever there is an *IMethod* object in Eclipse's UI, the entry '*Tag Smell*' will be present and an example of its elements is the action with the label '*Long Method*' that creates the Smellchecker*'s* Class "*CodeSmellsTag*"(from the package *codesmells*) upon selection. Entries for other code smells are added here, each of the code smells being described not by the object they create (all create element of the same "*CodeSmellsTag*" Class) but by their *id* in Smellchecker's internal processing and by their *label* in the UI.

Other contributions to Eclipse's workbench that are extension points to graphical elements are actions ("org.eclipse.ui.actionSets") and views ("org.eclipse.ui.views"). The actions represent menu and toolbar entries (the same actions in both) of three actions. They are the '*Retrieve Variables'* representing the component *Parse Code Variables* «Smellchecker» and activating the Smellchecker's Class 'GetCodeMetrics' from the package *smellsupdater*. Other action is '*RIntegration'* regarding the component *RIntegration «JRI»* (Class RIntegration), And the action '*SmellsDetector*' (Class '*SmellsDetector'*) represents the component *Detect Code Smells* «Smellchecker».

Source Code Annotation

Eclipse's SWT/JFace UI Framework provides user interface resources that allow the code smells annotations assistance. That is the case of the previous described contributions to Eclipse's *popupMenus*. They make tagging actions contributions to methods and types on the Eclipse workbench via a popup menu.

Each action is the representation of a code smell. The Class called (when a smell is selected) is "*CodeSmellsTag*", and within the Class the different smells are identified by the *id* of the action that created the Class (each time a code smell is selected, an object of the class it references is created). The class now checks the element (*IMethod* or *IType* representing methods and types on the *Java JDT Model*) for the presence of a Java Annotation for the specified smell. The *JDT´s Java Model* is in most practical respects "read only". *JDT's Java Model* do not present any functionality to change the code. So the changes are made directly to the Java source file with the *IBuffer* interface using the representation of the java class: the *compilation unit*. Changes with *IBuffer* are similar to *StringBuffer*, with the added difference that changing an *IBuffer* associated with a *compilation unit* propels the resulting changes throughout the Java Model. So, if the annotation for the particular code smell is already in the code it will be erased by deleting it using the *IBuffer*. If the code smell is not there it is added with *IBuffer*.
The annotation operation may be performed manually by annotating the code directly in the Eclipse text editor. Java annotations used for the code smells tagging are defined as in Listing 3 and Listing 4.

```
public @interface Smells {
      Smell[] type();
      String author();
}
```

Listing 3: Smell *annotation type*

```
public enum Smell {
      LongMethod, LongParameterList,
      LargeClass, LazyClass,DuplicatedCode
}
```

Listing 4: Smell *enum type*

The annotations information stays only in the java source file. When is time to access to them it must be through the JDT's Java Model.

BLR Variables Collection

When the user selects '*Retrieve Variables'* action either from the Smellchecker toolbar or from the Smellchecker's menu the action will create the Class 'GetCodeMetrics'. This class is responsible for athwart the project elements, getting the metrics for each

class and each type, as well as their annotation information regarding code smells. The function *Dispatcher.getAbstractMetricSource*(*JavaElement*) from the Eclipse Metrics project returns the metrics for the element. Annotation information is accessed through the JDT's Java Model.

Metrics calculation is accomplished by the Eclipse Metrics 1.3.8 plugin, and since we are extending its source code we have direct access to its functions. Metrics calculations are done in build time and persisted with JDBM[10] (is a transactional persistence engine for JavaJDBM).

All variables data are collected to four files. A set of four files for each project the user tries to collect variables for the BLR model from.

The Class Smellchecker*FileManager*(), in the Smellchecker *persistense* package, manages the files. One file gathers all metrics collected for code smells methods (including the smell presence variable). The other gathers the metrics collected for all classes and its annotations. The two other files are formatted versions of the other ones that can be load to R for the BLR calculation.

Models calibration

Calibration and validation of the BLR models is performed by the R statistical engine. Interaction between the plugin and R is made with *JRI*, a Java/R Interface that allows running an instance of R as a process that responds to command line type commands and outputs back to Java the data resulting from its computations.

R is dynamically loaded with the Java *JRI* jar Class *Rengine*(). Calls to its function *eval*(*String*) permit to feed R with commands like its command line interface from the standalone version.

The function that calculates the coefficients for the BLR model is from the form:

"logit<- glm(LongMethod~ MLOC + NBD + VG + PAR, family=binomial (link="logit"), na.action=na.pass)"

Where *LongMethod* is the dependant variable and *MLOC*, *NBD*, *VG* and *PAR* the independent variables. The call coefficients(logit) gets the coeficients from the model in a String format that needs to be parsed.

[10] http://jdbm.sourceforge.net/

There is only one calculation at each time for a particular code smell selected by the user.

Smells Detection

The code will be passed through for each method or class. Depending of what smell is selected the BLR estimation will be applied to the corresponding element. It is here that the value of the estimation is tested against the threshold value selected by the user. If the value of the estimation is higher or equal to the threshold a code annotation (similar to the ones previously described) is inserted in the code using the principles already described. If the node is considered to have a code smell, then a JDT Marker is used to mark it as problematic and it will rise a warning the same way as Eclipse's compilation warnings appear.

Smells Visualization

Since nodes identified by the model have been marked, they will appear in the error log view and they will have the same properties as compiler errors. So a jump to the smelly section of the code can be performed upon a click. Also the code annotated from the previous action.

5.5 **Smellchecker Usage**

Smellchecker makes several contributions to Eclipse's workbench Java integrated development environment. Relying on Eclipse's Java editor and Java Development Tools (JDT), added support is presented for source code tagging, processing and visualization of Code Smells related information.

Fig. 12 shows the user perspective with main components for Smellchecker operation. Smellchecker plugin consists of a specialized toolbar. A 'Smellchecker' menu on Eclipse's menu bar providing the same actions presented in the toolbar. A set of views. A collection of actions contributions appearing in the context of Java element types. And properties and preference pages to assist configurations.

The following sections will explain all components in the context of their usage.

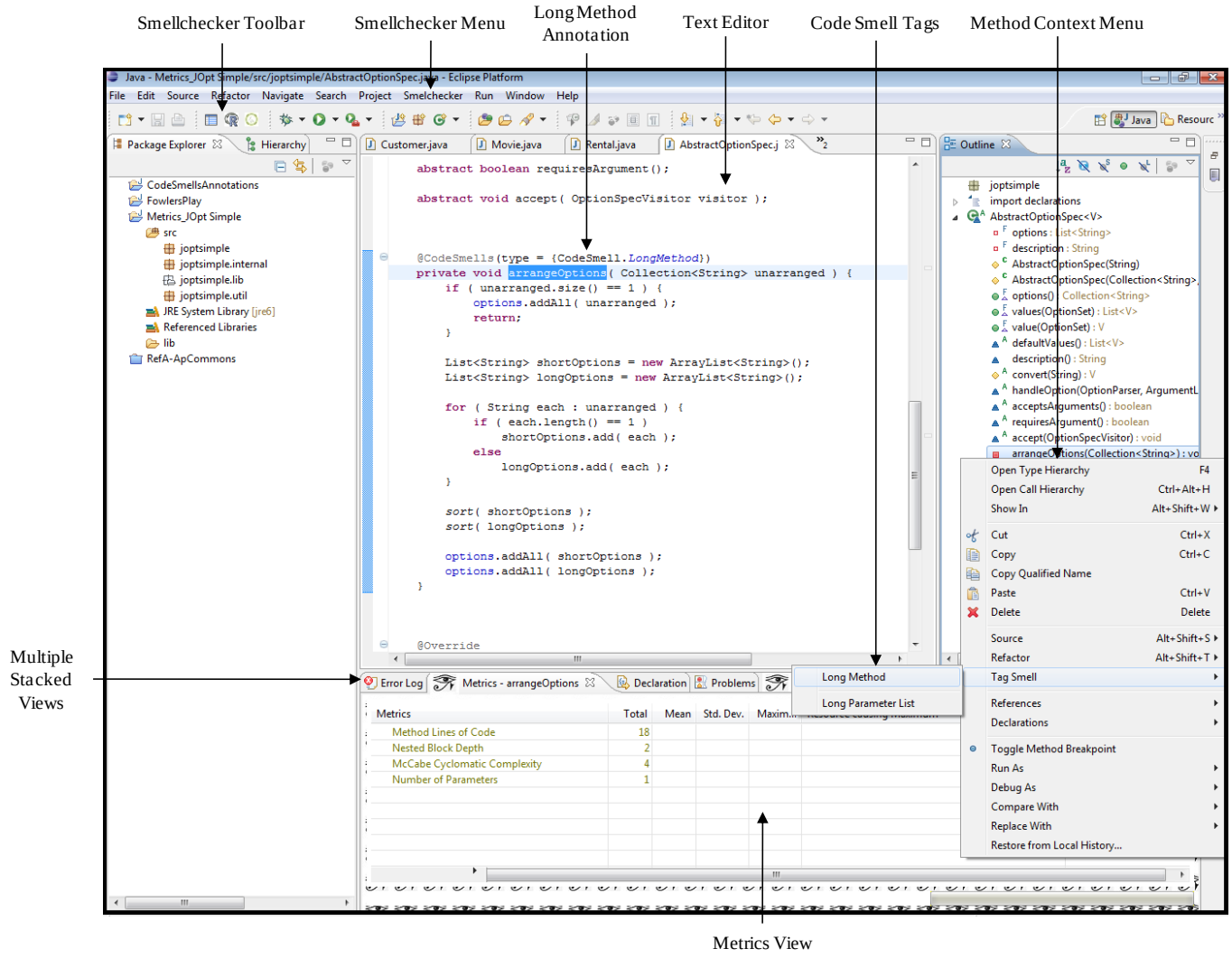


Fig. 12: Smellchecker Perspective

Annotating

Fig. 12 shows the user perspective with main components for Smellchecker operation. One of its components is a collection of actions contributions appearing in the context of Java element types. The 'Tag Smell' context menu is presented for both classes (*IType* Java Elements to be exact) and methods declarations. Within this menu the code smells names are representative of the type of annotation to be made for the selected element.

The context menu *'Tag Smell'* is sensitive to the kind of code smell to be associated with the class or method elements, presenting two different sets adequate to each nature. Methods can be tagged for the presence of: *Long Method*, *Large Parameter List*, *Duplicated Code*, *Dead Code*, *Speculative Generality*, *Temporary Field*, and the rest of Fowler's catalogue of code smells [3]. Classes can be annotated for: *Large Class*, *Lazy Class*, *Data Class*, *Duplicated Code*, *Speculative Generality*, as well as the rest of Fowler's catalogue of Class code smells.

Upon action selection a Java *CodeSmells* annotation will be seen in the editor. In the case of Fig. 12 the code smell tagging citation assumes the form:

```
@Smells(type = {Smell.LongMethod}, author="TP")
```

Listing 5: *Long Method* annotation

The annotation refers to the method *arrangeOptions(Collection<String>)*and denotes the presence of the *Long Method* code smell has perceived by the user. Further selection of the same action will delete the code smell *Long Method* annotation from source code. In short, if the method or class does not have an annotation for the perceived Code Smell the corresponding annotation will be inserted, otherwise it will be deleted. Has in all updates to the source code via Eclipse's Text Editor, the changes will only be committed to file with Eclipse's save command.

The annotation operation may be performed manually within the Text Editor by Smellchecker users, obliged they respect the Code Smells annotation syntax.

Classes and methods can be inflicted by more than one Code Smell at a time. For instance, in the previous example, *arrangeOptions(Collection<String>)* - annotated for the *Long Method* code smell - can be extended to include the *Duplicated Code* smell. Selection of this new smell symptom from the method's 'Tag Smell' context menu will update the annotation in`@Smells(type = {Smell.LongMethod}, author="TP")`

Listing 5 to the one in

Listing 6 :

```
@Smells(type = {Smell.LongMethod, Smell.DuplicatedCode }, author="TP")
```

Listing 6 - *Long Method* and *Duplicated Code* annotation

Further selection of the method's context menu *Duplicated Code* action will delete the smell annotation.

. When the indication of no code smell is signaled to the element (e.g. selecting *Long Method* again) the *CodeSmells* annotation is removed altogether.

There is no restrictions to the number of code smells an element can have. Annotations tagging via the 'Tag Smell' menu actions will account for the non occurrence of duplicated entries and misspelling of code smells.

Note that manual insertions are less restrict and duplicated entries or the misspelling of a method code smell for one of a class are not warned by the Java compiler. If a code smell indication do not conform to its right usage it will be ignored by the parser later.

Smellchecker users just need to be accountable for annotation of code smells presence within the code. All other elements will be perceived has free of code smell symptoms. For a most accurate calibration of the Binary Logistic Regression Model (BLR), is imperative that all elements not annotated, are assured by the user as representatives of code free of smell symptoms. If the user is unsure if a particular method or class is indicative of a particular code smell, then the element can be tagged to skip the parsing process so that its information is not fed to the BLR coefficients model calibration.

A view - context sensitive to the element selected – presents metrics calculations for all methods and classes. Its information can be used as guidance during the code smells annotation process aiding in an informed decision. This view (as all particular to the Smellchecker plugin), if not visualized in Eclipse's workbench, is accessed via Eclipse's Menu bar option *'Window'*, then option *'Show View'*, option *'Other…'*, Smellchecker separator and *'*Smellchecker*: Code Metrics'* selection.

Presented in Fig. 12, the Smellchecker*: Code Metrics* view can be seen in more detail in Fig. 13.

The example showing code metrics for the *arrangeOptions(Collection<String)*method. Metrics of *Method Lines of Code*, *Nested Block Depth*, *Cyclomatic Complexity* and *Number of Parameters* are present.

Metrics in this view are the explanatory variables for the BLR model when predicting a code smell at the method level. The view is context aware so a class, package, or method must be selected.

If a class was selected instead of a method, a different set of metrics would be presented by the Smellchecker*: Code Metrics* view.

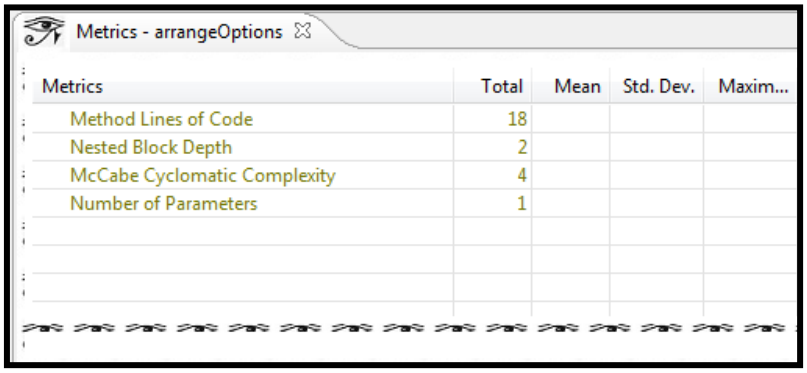


Fig. 13: 'Smellchecker: Code Metrics' view

Setting Properties and Preferences

For the metrics process collection to occur an option must be enabled in the Java project Properties page. In the 'Smellchecker' separator of the Java project Properties page, the *'Enable Metrics'* option must be checked, permitting project metrics calculation during build cycle (Fig. 14).

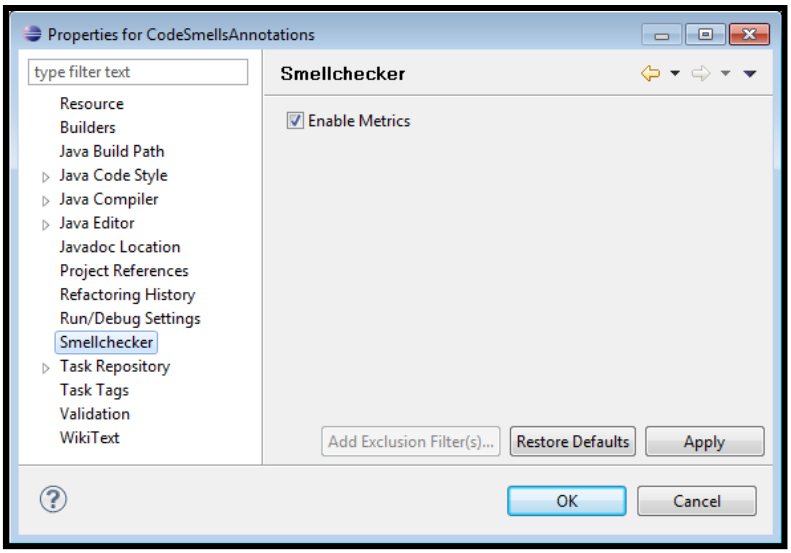


Fig. 14: Smellchecker properties page

Although the metrics calculation must be enabled for each desired project, Smellchecker's preferences are transversal to open projects and to Eclipse's work sessions. Fig. 15 shows Smellchecker's Preferences page:

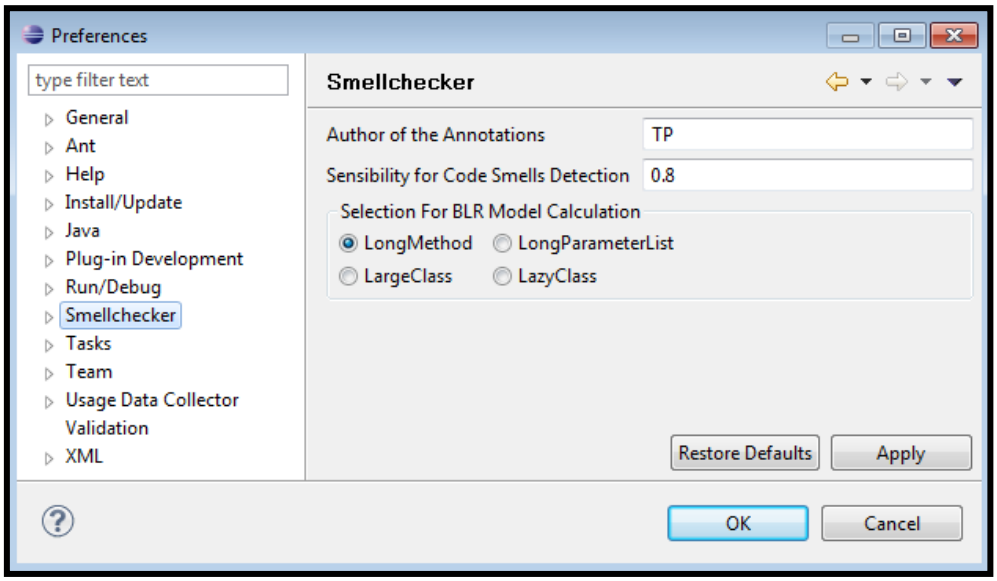


Fig. 15: Smellchecker preferences page

Smellchecker's users may tag for the occurrence of various code smells types and instances throughout the code. Yet, when is time for the BLR model regression coefficients calculation, the model accepts only one dependent variable at a time. In this page the user is responsible for choosing which of the supported code smells will serve

as that variable. A default value is provided in the form of *Long Method* selection. Value that users can modify at any time.

Smellchecker' users can set the sensibility threshold for the BLR model estimations. Only estimations above or equal the threshold are considered indicative of a code smell presence. The threshold can be any Real number between 0,00 and 1,00.

If the user fails to provide a threshold indication or provides a value not conforming to specifications, a warning will be presented (Fig. 16) and the default threshold value will be set to the Smellchecker default of 0.8.

In the last of the Smellchecker's preferences to be set, a field can also be edited to identify the author of the annotations. If not specified the Operating System login (if available) will serve that purpose.

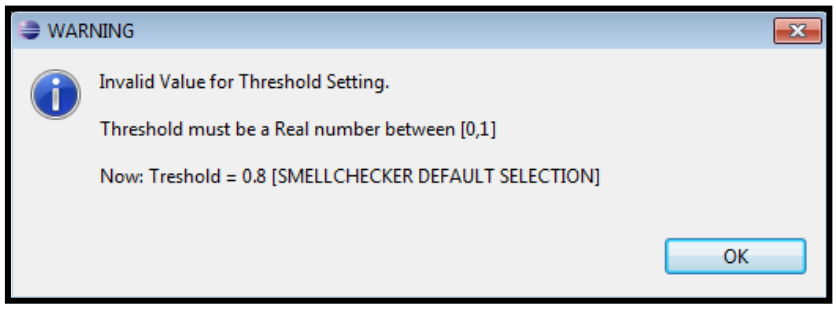


Fig. 16: Warning Invalid threshold setting

Collecting Data

When code smells tagging is done and metrics are enabled for the project, the assisted part of the code smells detection process is complete. Smellchecker can now retrieve all the necessary information for the BLR model calibration, calculation, and application.

To this task a Smellchecker toolbar provides the actions necessary to proceed with the code smells detecting process (Fig. 17). Description of the actions as provided by the entries in the 'Smellchecker' menu are in Fig. 18.

Fig. 17: Smellchecker toolbar

First action after the code smells tagging process is retrieving source code information on metrics values for all methods and classes, as well as the information of the code smells annotations.

Action '*Retrieve Variables'* is responsible for collecting to two different files the information of code elements (methods and classes), metrics values, and indications of code smells presence as tagged. One file gathers information for classes metrics and corresponding code smells, while the other does the same for method metrics and its code smells.

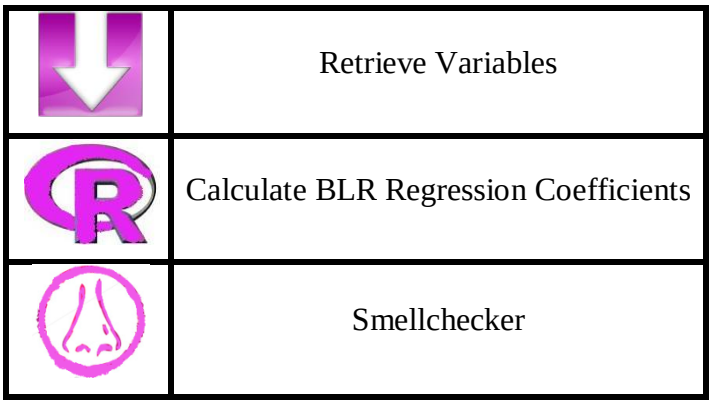


Fig. 18: Smellchecker menu actions

One necessary step by Smellchecker's users, is the selection of an element of the project in Eclipse's workbench prior the '*Retrieve Variables'* call, so the plugin knows in what project should it retrieve information from. Failure in compliance is pointed out by an warning advice, vide Fig. 19.

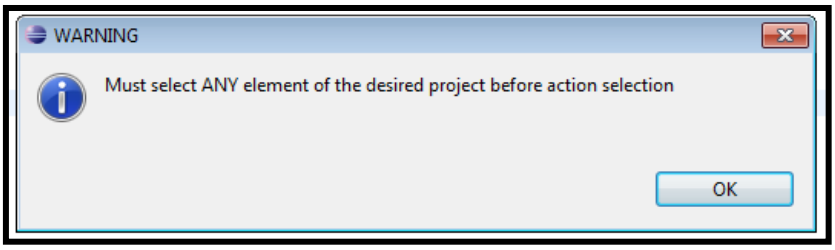


Fig. 19: Warning No project element selected

Although Eclipse permits various Java projects to be open simultaneously, and Smellchecker's tagging facilities the option to annotate any of them. Still, when the

retrieval of data process begins, it does so, for only one project, the one of the element selected.

Retrieval of data runs with the internal parser ignoring malformed values, failure in metrics retrieval or other inconsistencies that may occur for an element or other. These kind of elements will be skipped and not added to the variables table.

When the parser finishes processing the source code elements, it informs the user of the total tagged code smells instances (Fig. 20).

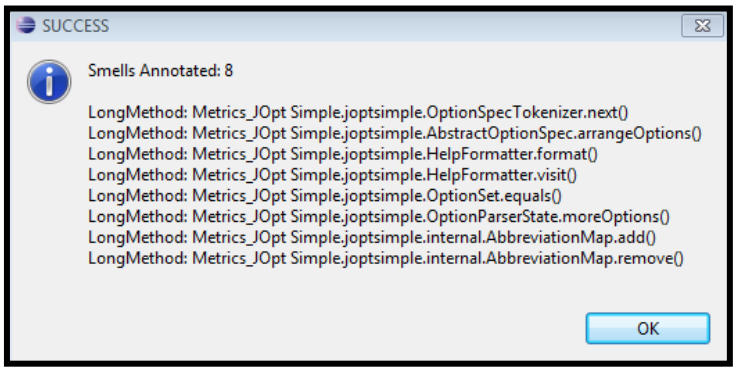


Fig. 20: Information of the total of annotations found in the code

Calibrating the BLR model

Next, is required by Smellchecker's users the order to calibrate the BLR model. The data provided and collected in previous steps will now be used for calculating the regression coefficients necessary to the model.

It is now, that the information provided by the Smellchecker '*Preferences'* page comes in handy. Remembering Fig. 15, and the indication of what code smell should be used as the dependant variable (it can be modified at any time).

Metrics values collected (restricted to method or classes metrics) will function as the predictable variables, and the indication of presence of a particular code smell (gathered from the *CodeSmells* annotations tags) as the outcome variable.

Action '*Calculate* BLR *Regression Coefficients'* is responsible for taking the correct data variables from the files produced in the previous process. And feed them to R statistical engine. Retrieving then the coefficients of the BLR model.

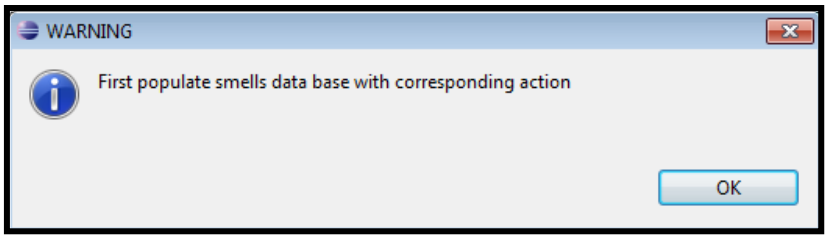


Fig. 21: Warning Database not populated

The outcome of the process is passed to the user using information messages. Fig. 21 means that no proper data base of metrics values and code smells tagging information exists (‘*Retrieve Variables’* action must run).

Fig. 22: Error R is offline

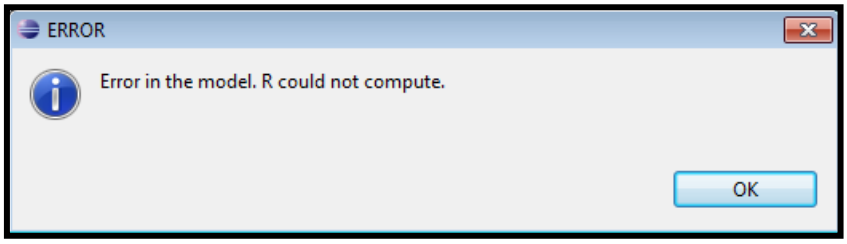


Fig. 23: Error R could not compute BLR coefficients

Fig. 22 and Fig. 23 are error signals. The former informs that R statistical engine is offline while the latter informs that R could not estimate the coefficients for the BLR model. Either because of R internals, or because bad input data. Fig. 24 has the formula for success. Indication that the BLR regression coefficients are computed and the model proper to use.

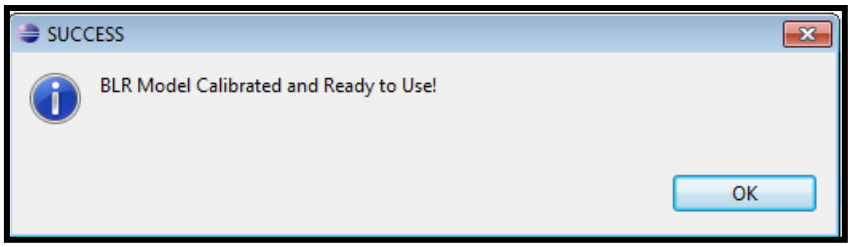


Fig. 24: Success BLR model calibrated

Smellchecking

Smellchecker's users can now analyze the performance of the estimated BLR model. Upon selection of the 'Smellchecker' action the calibrated BLR model will be applied to all elements of the source code. Depending if it was calibrated for a method or a class code smell, so will be the kind of source code elements processed.

The calibrated BLR model will give an estimate value of the probability for a code smell presence. Users sensibility threshold gives the element of decision on whether or not a an element should be marked as 'smelly'. Estimations above or equal to the threshold will be marked as code smells problematic.

Smellchecker will annotate the code in a similar way to the users annotations. Java *CodeSmells* annotations will be automatically inserted when needed.

An estimation of the validity of the process will be given to the user. It is a summary consisting of the total of detections made by the BLR estimation model. The number of false positives detected and the number of false negatives will be shown to the use.

6 Case study

Contents

6.1 Objectives 55

6.2 Source Code Projects 56

6.3 Data collection and descriptive statistics 57

6.4 BLR Estimation 61

6.5 Conclusions 68

Two open source Java systems were used to validate the BLR process presented in this work. The case study presented in this section also serves the purpose of collecting information data to draw an analysis and discussion for the validity, extensibility and usefulness of the Smellchecker's approach. First in the chapter we present the context of the problem and what objectives our solution will explore.

6.1 **Objectives**

Problem Statement

The eradication of source code smells is known to improve the design, readability and extensibility of software. Code smells indications for when to apply specific refactorings have been devised, yet they are not objective and when they are, they are not consensual. For example there is no threshold or collection of metrics settings that indicate when a method should be considered long. From experience and from literature we know that many faults are associated with methods that are too long. But what is long? Is long related to a specific domain? Maybe with a project? Or with a developer taste? And there is some way to derive some metrics or setting of the already existing ones that can indicate where is the long problem?

Through empirical experimentation with the proposed BLR we will try to shed some light on this questions defining objectives for the study presented in this chapter.

Objectives Definition

The main objective of this dissertation is reducing the subjectivity in code smells detection. Such an ambitious and generic goal is refined in several research objectives that facilitate its assessment achievement.

Research Objectives (RO):

- **RO 1:** Determine if the proposed BLR automate process is valid for *Long Method* detections;
- **RO 2:** Determine if the quality of the detection is crosscutting to different projects;
- **RO 3:** Determine if aggregated data from various projects sources makes for a more balanced and precise BLR model.

Design Planning

Through the study of two software projects hypothesis will be formulated about the presence of *Long Method* instances. This hypothesis will provide the grounding for the BLR experimentation detection mechanism. *Long method* is the code smell that at first instance may be better suited for the complexity metrics Smellchecker has. Long method is traditionally linked to *Lines Of Code* and other complexity measurements related to size [27, 39].

Obtained results will provide the ground for refinement of the defined objectives (RO1, RO2 and RO3). And different data sets in the form of annotated projects by a domain expert will provide the basis for empirical validation and validity of the process.
Next section describes the projects this study uses as sample.

6.2 Source Code Projects

Source code data is necessary for calibration and verifiability of validation of the results provided by Smellchecker. The criteria for selecting the Software projects used in this case study was that they be open source (providing availability), written in Java (for Smellchecker's parser to work), relatively small (because of the necessity of manual identification of the code smells), from the same application domain (for studying BLR generalization over a project), and in number not inferior to two for intertwining BLR model estimations. *JOpt Simple* and *Apache Commons CLI* projects are examples of the established criteria. Info on the projects follows.

JOpt Simple

JOpt Simple[11] is a Java library for parsing command line options. It attempts to honor the command line option syntaxes of *POSIX getopt()* and *GNU getopt_long()* in the interest of striving for simplicity. It also aims to make option parser configuration and retrieval of options and their arguments simple and expressive.

Version used:

- *JOpt Simple 3.2* (08-Dec-2009)

[11] http://jopt-simple.sourceforge.net/

Apache Commons CLI

The Apache Commons CLI[12] library provides an API for parsing command line options passed to programs. It's also able to print help messages detailing the options available for a command line tool.

Version used:

- *Apache Commons CLI 1.2*

6.3 Data collection and descriptive statistics

With the projects selected and the BLR metrics defined by Smellchecker's we can summarize the following information for our study:

Population: *Java Software Projects* (Object Oriented Projects**)**
Sample: *Apache CommonsCLI 1.2* and *JOpt Simple 3.2* (Java Applications)
Variables: *MLOC, NBD, VG, PAR* (Software Metrics)
Dependent Variables: *Long Method Present*

With the variables for our BLR model already decided (with *Long Method Present* being defined as true or false for a given method) we start analyzing the projects for clues on what measurements to make that could validate the defined research objectives.

Size metrics for the two selected projects are presented in Table 9. *Apache Commons CLI (Apache)* Total Lines Of Code (TLOC) doubles the value obtained from *JOpt Simple (JOpt)*, and it has less classes and fewer methods. One hypothesis could be raised for the probability of *Long Method* instances in *Apache* to be higher that of *JOpt*.

Table 9: *JOpt* and *Apache* descriptive statistics

	JOptSimple	***ApacheCommonsCli***
Number Of Classes	40	23
Number Of Methods	217	184
Number of Interfaces	3	1
Number of Packages	3	1
Total Lines of Code	1603	3345

[12] Link: http://commons.apache.org/cli/

Before any BLR estimation to occur let's analyze some descriptive statistics for both projects to see if our hypotheses still hold.

Table 10: JOpt *MLOC* statistics

	Min.	**1st Qu.**	**Median**	**Mean**	**3rd Qu.**	**Max.**
JOpt	0	1	2	3.586	5	20
Apache	0	5	6	11	13	77

Table 10 shows *MLOC* descriptions for both projects. Differences exist in their distribution. Fig. 25 and Fig. 26 show both projects' *MLOC* boxplot side by side.

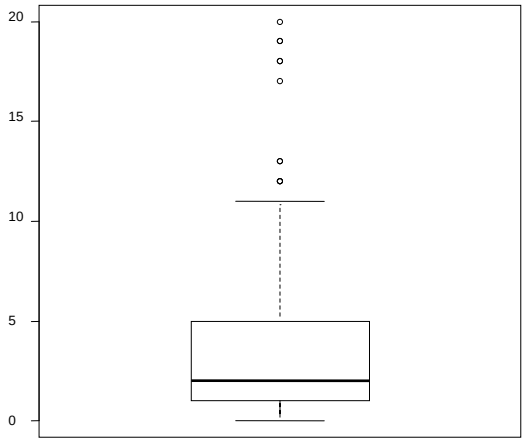


Fig. 25: JOpt *MLOC* boxplot

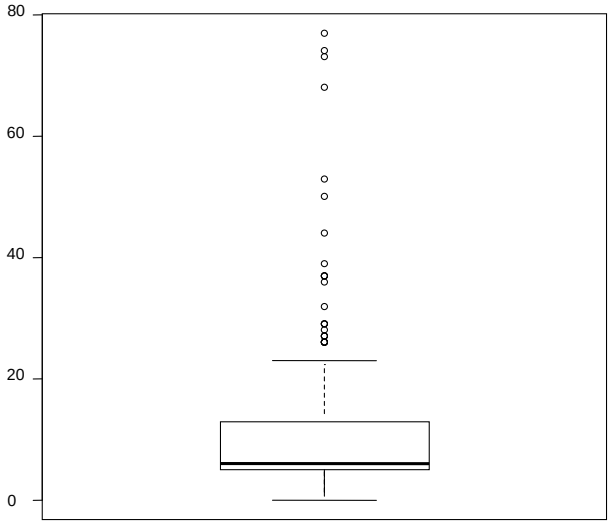


Fig. 26: Apache *MLOC* boxplot

Apache's boxplot shows more outliers and more extreme values while JOpt's boxplot indicate that 75% of the values are within the 5 MLOC reach. MLOC is not the only predictor for the Long Method code smell, nevertheless methods with as few as 5 lines of code or less rarely are problematic. Everything indicates that in fact the presence of the Long Method code smell is higher for Apache commons.

Next step of the process is annotating the code of the two projects for occurrences of the Long Method code smell.

Concerning the *Long Method* code smell, Fowler[3] defined signals of its presence with the following:

- The longer the method the more difficult it is to understand;
- Semantic distance occurs when there is a difference between what the method does and how it does it;
- Comments are indications that procedures are not clear;
- Conditionals and loops instances are good signs for extraction.

McConnell[41], explained his insights on *Bad Routines* and *Routine Size* with:

- Accessor routines should be short;
- Depth of nesting, number of variables, and other complexity related considerations should dictate the length of the routine rather than imposing length restrictions;
- Complex algorithms can grow up to 100 – 200 Lines Of Code (LOC);
- Routines longer than 200 LOC decrease understandability;
- If an a method or procedure is not invoked for a single purpose is an example of a *Bad Routine*;
- The upper limit for an understandable number of parameters of a routine is 7.

Since I do not have acquaintance with the projects, my analysis for *Long Method* presences will be from the point of view of someone who first has to understand the software before making changes to it. So in light of the previous definitions presented I follow the guidelines that express more clarity, classifying methods for Long Method presences with the following heuristics:

- if it is not understandable what it does;
- if it is long and tackles different concerns;
- if its name is not a clear indication of what it does;
- if it does more than indicated by the name;
- if conditionals and loops could be expressed by a method call;
- and if it has comments to clarify what the code does.

Results from the tagging process are expressed in Table 11. Results are a confirmation of the expectations formed during the data analysis. More than the double of *Long Method* instances were identified in the Apache Project comparing to JOpt.

Table 11: *Long Method* tagged instances

	Number of *Long Method* identifications
JOpt	9 (of 217 methods)
Apache	24 (of 184 methods)

Now, it is important to compare correlations between the variables, specially how they behave when compared to the presence of *Long Method* instances.

Table 12 and Table 13 represent data from the Spearman Rho test for JOpt and Apache collected variables. The correlation values for JOpt variables expressed in

Table 12, indicate a low level of correlation between metrics and the *Long Method* instances prediction. Indication they may not result in a good estimation when applied to the BLR model. The results also express a reasonable correlation between *MLO*, *NBD*, and *VG*. With *PAR* metric being the least related not only to each other, but also to the *Long Method* instances.

Table 12: Spearman Rho correlation test for JOpt

	MLOC	NBD	VG	PAR	LongMethod
MLOC	1.0000000	0.5713811	0.6931416	0.3166164	0.3203925
NBD	0.5713811	1.0000000	0.5246364	0.1539146	0.2751765
VG	0.6931416	0.5246364	1.0000000	0.2893859	0.3787257
PAR	0.3166164	0.1539146	0.2893859	1.0000000	0.1286591
LongMethod	0.3203925	0.2751765	0.3787257	0.1286591	1.0000000

This results may be explained by the fact that, since the main factor of identification of *Long Method* instances was my understandability of the code, similar methods in terms of the metrics readings were tagged differently based on my comprehension of the nature of the computation.

For the Apache project correlation results are much better (Table 13) but still not as high as expected to be a good fit.

Table 13: Spearman Rho correlation test for Apache

	MLOC	NBD	VG	PAR	LongMethod
MLOC	1.0000000	0.7409062	0.7215259	0.2483759	0.4899487
NBD	0.7409062	1.0000000	0.9093956	0.2199401	0.5797428
VG	0.7215259	0.9093956	1.0000000	0.2669665	0.6090507
PAR	0.2483759	0.2199401	0.2669665	1.0000000	0.2589788
LongMethod	0.4899487	0.5797428	0.6090507	0.2589788	1.0000000

Smellchecker's do not test the validity of the model through statistical analysis tough. The coefficients for the BLR model are calculated without automatic measuring correlation, checking for multicollinearity in the independent variables, or the goodness-of-fit analysis for the variables. Instead, it calculates the BLR coefficient models and are up to the users the responsibility of recalibrating the model if they see fit. Next section shows the results of applying the BLR estimation with the given data.

Now, in order to accomplish our first research objective:

RO 1: Determine if the proposed BLR automate process is valid for *Long Method* detections.

The BLR model must be calibrated and evaluated for each project and the measurement of the quality of its estimation will be the percentage of correct instances the model can predict.

6.4 **BLR Estimation**

With the data input provided, Smellchecker's has calibrated two BLR models for the *Long Method* estimation. JOpt calibrated model is the following:

$$f(z) = \frac{1}{1 + e^{-z}} \quad \wedge \quad z = -13.1 + 0.41 \times MLOC + -1.32 \times NBD + 2{,}61 \times NBD + -0{,}56 \times PAR$$

Applying the model to the same project, results in 6 detections of *Long Method* instances with the threshold value of 0,8. All 6 detections coincide with previous annotated smells, while 3 are false negatives, and there is no false positives.

Table 14 indicates the results of the BLR prediction in terms of the annotated smells.

Table 14: BLR prediction JOpt (Threshold=0,8)

	Smellchecker BLG Estimation: *Long Method*		
***Long Method* Annotated**	**0**	**1**	Correct %
0	208	0	100%
1	3	6	33%
		Overall %	98,62%

Although the overall percentage was good, the detections concerning actual instances of *Long Method*s is slightly above average. The Spearman analysis done previous was an indication that this was to be expected. The very low number of annotations of *Long Method*s may explain why the model does not give a better estimation. In order to gain a better idea of what the BLR model may be expressing we need to vary its sensibility. Table 15 summarizes the results.

Table 15: JOpt BLR estimations on JOpt

JOpt BLR estimations on JOpt					
Threshold Sensibility	False Positives	False Negatives	Negative Correct (%)	Positive Correct (%)	Overall Correct (%)
0,9	0	6	100%	33,3%	97,2%
0,8	0	3	100%	66,7%	98,6%
0,7	0	3	100%	66,7%	98,6%
0,6	0	3	100%	66,7%	98,6%
0,5	2	3	99,0%	66,7	97,7
0,4	3	2	98,6%	77,8	97,7
0,3	4	2	98,1%	77,8%	97,2
0,2	5	2	97,6%	77,8%	96,8
0,1	7	0	96,6%	100%	96,8%

We can see that although the number of overall correctness is high, the percentage in terms of the total estimation of the cases where existed the Long Method (positives corrected estimation) is lower. But overall the BLR model calculated do not refute our first research objective (RO1).

We will now repeat the same process for Apache Commons data. Here is the result of the calibration effort:

$$f(z) = \frac{1}{1 + e^{-z}} \quad \wedge \quad z = -7.60 + 0.06 \times MLOC + 1.04 \times NBD + 0{,}46 \times NBD + 0{,}25 \times PAR$$

When applied to Apache Commons source code the results are the ones shown in

Table 16.

Table 16: BLR prediction Apache (Threshold=0,8)

	Smellchecker BLG Estimation: *Long Method*		
***Long Method* Annotated**	**0**	**1**	Correct %
0	157	3	98,13%
1	12	12	50%
		Overall %	91,85%

Note that the probability of detection was lower in every account if we compare it with the JOpt model. Estimations for other sensitivities are presented in

Table 17.

Table 17: Apache BLR estimations on Apache

Apache BLR estimations on Apache					
Threshold Sensibility	False Positives	False Negatives	Negative Corrected %	Positive Corrected	Overall Correct %
0,9	1	15	99,49%	37,5%	92,63%
0,8	3	12	98,45%	50,00%	93,01%
0,7	3	11	98,45%	54,17%	93,55%
0,6	4	9	97,93%	62,50%	94,01%
0,5	4	8	97,93%	66,67%	94,47%
0,4	6	6	96,90%	75,00%	94,47%
0,3	6	4	96,90%	83,33%	95,39%
0,2	8	2	95,85%	91,67%	95,40%
0,1	16	0	91,71%	92,63%	92,63%

Not that this estimations again confirm our RO1 that it is possible to predict Long Method instances with the calibrated BLR model. What we can see is that the quality of the estimation of the *Long Method* depends upon the sensibility of the threshold. If the sensibility is lower more Long Methods are detected (less false negatives) but more are the rate of false positives.

So now that we have the BLR model from two different projects, we will move to RO2:

- **RO 2:** Determine if the quality of the detection is crosscutting to different projects.

To see if RO2 is valid, the BLR model calibrated from one application will be used to estimate *Long Methods* on the other, and then the results will be analyzed to see if the model is still valid or its validity depends entirely of the input data it receive and is not possible to detach it from it. Table 18 depicts the results of applying the JOpt BLR model to Apache and Table 19 the inverse. Apache BLR model estimating values on JOpt Source.

Table 18: JOpt BLR estimations on Apache

JOpt BLR estimations on Apache					
Threshold Sensibility	False Positives	False Negatives	Negative Corrected %	Positive Corrected	Overall Correct %
0,9	9	3	95,34%	87,50%	94,47%
0,8	9	2	95,34%	91,67%	94,93%
0,7	11	1	94,30%	95,83%	94,47%
0,6	14	1	92,75%	95,83%	93,09%
0,5	15	1	92,23%	95,83%	92,63%
0,4	15	1	92,23%	95,83%	92,63%
0,3	16	1	91,71%	95,83%	92,17%
0,2	16	1	91,71%	95,83%	92,17%
0,1	19	0	90,16%	100%	91,24%

Table 19: Apache BLR estimations on JOpt

Apache BLR estimations on JOpt					
Threshold Sensibility	False Positives	False Negatives	Negative Corrected %	Positive Corrected	Overall Correct %
0,9	0	9	100,00%	0,00%	95,85%
0,8	0	9	100,00%	0,00%	95,85%
0,7	0	9	100,00%	0,00%	95,85%
0,6	0	9	100,00%	0,00%	95,85%
0,5	0	9	100,00%	0,00%	95,85%

0,4	0	9	100,00%	0,00%	95,85%
0,3	0	9	100,00%	0,00%	95,85%
0,2	1	8	99,52%	11,11%	95,85%
0,1	2	5	99,04%	44,44%	96,78%
0,05	6	3	97,125	66,67%	95,85%

Although the estimations of Apache BLR did not bring good results on the JOpt project (it is not strange if we remember that even for the own project the values were a bit off),

The JOpt BLR model performed even better in Apache that the native apache BLR model on itself. This is a great evidence that proves our RO2. We see that not all models can be translated well to other projects (Apache BLR case) but we proved that some can even exceed the expectations and improve the estimations.

This leads to our last research objective:

- **RO 3:** Determine if aggregated data from various projects sources makes for a more balanced and precise BLR model.

The data from the two projects will be combined into one BLR mode estimation and we will analyze if it feats the two project simultaneously well. Table 20 shows the Spearman test for the aggregated data. It shows a medium relation, but as we learned from the Apache project this do not mean better estimations.

Table 20: Spearman correlation test on aggregated data of the two projects

	MLOC	NBD	VG	PAR	LongMethod
MLOC	1.0000000	0.6629293	0.6045190	0.2156950	0.4059708
NBD	0.6629293	1.0000000	0.7275962	0.1768284	0.4729347
VG	0.6045190	0.7275962	1.0000000	0.2755200	0.5094618
PAR	0.2156950	0.1768284	0.2755200	1.0000000	0.2036277
LongMethod	0.4059708	0.4729347	0.5094618	0.2036277	1.0000000

Here is the BLR model for the aggregated data from the two projects (Apache and JOpt)

$$f(z) = \frac{1}{1 + e^{-z}} \quad \wedge \quad z = -6.69 + 0.065 \times MLOC + 0.44 \times NBD + 0{,}65 \times VG + 0{,}27 \times PAR$$

Table 21 shows the results table of applying the new BLR model to JOpt project. Only for high tolerance discrepancies between the estimation and the real value does the model stars to detecting the present *Long Method* instances. Let's compare now the results with the model´s estimation to the Apache project in Table 22.

Table 21: Aggregated BLR estimations on JOpt

Aggregated BLR estimations on JOpt					
Threshold Sensibility	False Positives	False Negatives	Negative Corrected %	Positive Corrected	Overall Correct %
0,9	0	9	100%	0,0%	95,9%
0,8	0	9	100%	0,0%	95,9%
0,7	0	9	100%	0,0%	95,9
0,6	0	9	100%	0,0%	95,8%
0,5	0	8	100%	11,1%	96,3%
0,4	0	7	100%	22,2%	96,8%
0,3	0	6	100%	33,3%	97,2%
0,2	2	5	44,4%	99,0%	96,8%
0,1	2	5	44,4%	99,0%	96,8%

Table 22: Aggregated BLR estimations on Apache

Aggregated BLR estimation on Apache					
Threshold Sensibility	False Positives	False Negatives	Negative Corrected %	Positive Corrected	Overall Correct %
0,9	3	13	98,1%	45,8%	91,3%
0,8	4	10	97,5%	58,3%	92,4%
0,7	4	8	97,5%	66,6%	93,5%
0,6	5	7	96,8	70,8	93,5
0,5	6	6	96,25%	75,0%	93,5%
0,4	6	5	96,3	79,2	94,0
0,3	7	1	95,6	95,8	95,7
0,2	11	1	93,1	95,8	93,5
0,1	20	0	87,5	100	89,1

We can see that the new BLR model is still useful, but it is hard to say that it better and that improves significantly the model. From this example more conclusions we cannot tell, despite the fact that it is still applicable. But prove of any improvements is not clear.

6.5 **Conclusions**

We proved that the BLR model approach is a sound technique to evaluate code smells. Evidence of that was achieved through an example concerning the Long Method code smell. We proved soundly our fist two research objectives. The BLR method can predict

Long Method instances and that the model can be used with efficiency in different source projects. For the third objective, it is proved that the model do not lose validity, but his efficiency is to question and only this example is not enough to make any substantial claim.

7 Related Work

Contents

7.1 Papers and Articles 70

7.2 Open Source Tools..... 75

7.1 **Papers and Articles**

The focus of this work is on source code smells detection. Architectural code smells and refactorings [24], so as the ones applied to higher abstraction diagrams such as UML, are not covered. The same goes for *Duplicated Code* detection mechanisms, which is an active research area at this moment more than capable of sustaining a work study by its own. However, sparingly and not exhaustive, references to *Duplicated Code* detection mechanisms and tools may occur, particularly in the context of more ample code smell detection suites.

Code smells identification in source code can be approached from two distinct angles. The first concerns detection methods of pure qualitative nature, making use of biased heuristics that pertain to the expert's opinion that voices his choice. The second presents a more formal approach using software metrics calculations but relying in subjective thresholds, lacking strong empirical evidence or any sign of empirical proof at all.

A account of the advances on code smell detection and characterization follows in a brief survey.

Fowler and Kent Beck introduced the concept of code smells and produced an original catalogue of twenty-two smells providing heuristics of qualitative nature for their detection [3]. Kerievsky extended Fowler's work with new code smells heuristics in the vein of his predecessor, but aimed at introducing design patterns through refactorings [20].

Simon *et al*. [42] proposed an approach to code smells detection based on a generic similarity measure of code entities (cohesion), and developed a prototype - extension of the metrics tool Crocodile - to visualize those distances in space, which then could serve as indication for applying four of Fowler's refactorings. With proximity in space indicating the relative affinity between entities, the developer aimed with knowledge of the code's design could then make an informed choice on which entities to reallocate so that the principle of cohesion denoted visual could be expressed more directly by the design.

Making use of static analysis to gather simple code metrics over the program abstract syntax tree Dudziak and Wloka implemented the prototype add-in J/Art for NetBeans IDE [30]. Providing developers with ad-hoc support for detecting code structural weaknesses (smells) and for choosing which refactor to perform. This tool identifies 12

of the 22 code smells proposed by fowler in a very straightforward way and adds two new smells: Shared Collection and Unnecessary Openness. The detection process is simplified by assuming subjective thresholds in order to test smells presence.

Van Emden and Moonen produced a new set of code smells qualitative heuristics specific to deal with test code for which they created a set of new refactorings [28]. They ventured further and continued expanding the concept of code smells to domain specific interests. Namely, not conformance to coding rules and concrete Java constructs (i.e. Typecast and Instanceof), could be understood as code smells as well [29]. Also proposed was a set of design considerations for code smell detection tools based on static analysis, upon which they developed a prototype tool jCOSMO responsible for collecting primitive smell aspects (aspects visible in source code entities) through code parsing [29]. The primitive smell aspects collected where the presence of Java constructs Instanceof and Typecast.

Tourwé and Mens first significant contribution to the field, come in the form of logic meta programming [43] applied to code smells identification and subsequent refactorings proposal [34]. With direct access to object-oriented source code entities (Java or Smalltalk) by a metalevel interface representational mapping, the authors, through the power of declarative meta language SOUL, with elegantly concise logic rules were able to detect presence of code smells Obsolete Parameter and Inappropriate Interfaces. The same principle was further extended so to dynamically update refactoring proposals. Same authors followed work with a comprehensive survey on refactoring [8] where a generic refactoring process involving quality assessments after each refactoring step was defined.

The technique of critical pair-analysis was used by Mens, Taentzar et al [26] to gain insight on refactorings implicit dependencies. By representing refactorings as graph transformations suggestion of an order for refactoring usage was demonstrated.

Marinescu defined in more proper quantitative terms, metrics with which to derive evidences of code smells presence [44].

Apart of creating a code smells taxonomy, Mäntylä [27] evaluated each one of Fowler's code smells (and his own *Dead Code* smell detailed in the Code smells chapter) according to a measure he introduced named Measurability. This measure was subjectively rated, between 0 (impossible to measure) and 5 (easy and correct to predict), according to his expert knowledge of the willingness of measurability of each smell. For example, he rated *Long Method* as a 5 on Measurability and proposed a

polynomial metric that combined Number of Lines of Code (NLOC), Cyclomatic Complexity and Halstead metrics to do the measurement.

Mäntylä and Lassenius' first Finnish empirical findings showed evidence of conflicting smell evaluations calls when judged by experts [11]. Second exposition [45] confirmed first findings and corroborate the difficulty of setting code metrics to predict even the simplest code smells (i.e. *Large Class* as an example).

Recently, Moha et al. [31] introduced new functionalities to the code smells method DÉCOR that embodies and defines the specification and detection of code and design smells, that when instantiated, is capable of detecting design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and 15 underlying low level code smells.

A taxonomy for comparing between different code smells detection methodologies (or code smells detection approaches) and tools was created to help position this work detection process within available research. A guideline followed was that each methodology was analyzed from the smell detection capabilities point of view, not caring for the processes underlining those detection. Because for a smell detection point of view is irrelevant how the methodology is implemented, just its results and functionalities are of interest.

An explanation of the not trivial identifiers follows:

- Assessment: Type of assessment followed by the methodology. It can be of two values: Quantitative or Qualitative, depending if it the heuristic in use can be quantified or not, respectively.
- Smells: Code smells supported by the heuristics.
- Tool: Name of the tool that instantiates the methodology if it exists.
- Language: It is tool oriented in the sense that relates to code languages the tool support.
- Functionalities: Types of functionalities supported: (Code smells) Detection, mechanisms of Visualization (of the code smell after detection), Proposal of appropriate refactorings, automatic Refactorings application, and code smells Extensible (if it permits extending the list of code smells detected).
- Heuristic: automatic if it supports a fully automatic detection system under some condition. Manual, if manual calibration of some sort is expected every single time the detection process runs.

- Automation: as in process Automation. Values: automatic if the process do not need any kind of users or developers input. Semi-automatic if needs some sort of minimal user or developer input.

Tables Table 24, Table 25 and Table 25 compare some of the methodologies that were studied with ours.

Table 23: Smells Detection Comparison Part 1

Methodologies	**Fowler**	**Simon et al.**	**Emden & Moonen**	**Smellchecker**
Year	1999	2001	2003	2011
Assessment	Qualitative	Quantitative	Quantitative	Quantitative
Smells	Fowlers 22 Smells	Feature Envy, Inappropriate Intimacy, *Large Class*, *Lazy Class*	Instanceof, Typecast (Language specific Java)	Undifferentiated[13]
Tool	-	Crocodile (Enhanced)	jCOSMO	Smellchecker
Language		Integrated in a CASE tool	Java	Java
Functionalities	-	Detection, Visualization	Detection, Visualization, Extensible	Detection, Visualization
Heuristic		Automatic	Automatic	Automatic
Process Automation	-	Semi-automatic	Semi-automatic	Semi-automatic

The idea of automating code smells detection by using metrics and tools is not new as seen by the previous survey. The technique that this work proposes to automate [10] is in contrast with all other works because of its implementation of a dynamic statistical process that relies on expert's knowledge that can be applied, theoretically[14], to any smell.

[13] In theory all code smells that current metrics and a polynomial aproximation can derive

[14] Empirical studies are necessary to validate that assumption. For now only *Long Method* Smell is validated

Table 24: Smells Detection Comparison Part 2

Methodologies	Dudziak &Wloka	Tourwé & Mens	Smellchecker
Year	2002	2003	2011
Assessment	Quantitative	Quantitative	Quantitative
Smells	12 Fowler Smells[15], Shared Collection, Unnecessary Openness	Obsolete Parameter, Inappropriate Interfaces	Undifferentiated[16]
Tool	J/Art (NetBeans Add-in)	Refactoring Brower (extended with SOUL)	Smellchecker
Language	Java	SOUL over Smalltalk/Java	Java
Functionalities	Detection, Visualization, proposal	Detection, Visualization, Proposal, Extensible	Detection, Visualization
Heuristic	Automatic	Automatic	Automatic
Automation	Semi-automatic	Semi-automatic	Semi-automatic

Table 25: Smells Detection Comparison Part 3

Methodologies	CodeNose	Smellchecker
Year	2005	2011
Assessment	Quantitative	Quantitative
Smells	Half of Fowlers Smells	Undifferentiated[17]
Tool	Eclipse Plugin	Smellchecker
Language	Java	Java
Functionalities	Detection, Visualization, proposal	Detection, Visualization
Heuristic	Automatic	Automatic
Automation	Semi-automatic	Semi-automatic

Classifications for the different methodologies seem similar, without any factor that stands out and making this work's approach obvious better than all rest. This was

[15] Fowler smells supported: Duplicated Code, *Long Method*, Large Class, Long Parameter List, Feature Envy, Lazy Class, Speculative Generality, Temporary Field, Inappropriate Intimacy, Data Class, Refused Bequest, Comments.

[16] In theory all code smells that current metrics and a polynomial aproximation can derive
[17] In theory all code smells that current metrics and a polynomial aproximation can derive

intended and to be expected. And serves to stress the real meaning behind this work's methodology. This is not, at least primary, a methodology for end user usage, what we have here is the unique proposal of automate the methodology described by Bryton et al. [10] so that code smells metrics can be derived, refined and continuous improved. As seen by the survey, most code smells detection approaches use some kind of metrics to identify where on the source code the smell could be. And the process that it is used by the researchers is always the same: "Let's look at the Code Smell intuitive definition", "Let's look at code metrics that I think could express this Smell", and "This is it". But the it they refer is not always a hit, more of a miss more. So to gather some sort of understanding, of the relation between metrics and code smells, one must step out of the box and let the associative process of relating metrics and smells be guided externally, so to relations and affinities we have not suspected yet can be made known and then, the intuitive human process begins again choosing what of this relations should be consciously put inside the box.

7.2 **Open Source Tools**

A few open-source tools exist for detecting code smells in Java code. Most of them use static analysis, that is, they do not require executing the program, such as the one presented in this paper.

PMD (http://pmd.sourceforge.net/**).** This widely used tool uses static analysis techniques to scan Java source code and look for potential problems like *possible bugs* (empty try/catch/finally/switch statements), *Dead Code* (unused local variables, parameters and private methods), *suboptimal code* (wasteful String/StringBuffer usage), *overcomplicated expressions* (unnecessary *if* statements, *for* loops that could be *while* loops) and *duplicate code* (copied/pasted code means copied/pasted bugs). PMD is integrated with *JDeveloper, Eclipse, JEdit, JBuilder, BlueJ, CodeGuide, NetBeans, IntelliJ IDEA, TextPad, Maven, Ant, Gel, JCreator,* and *Emacs*.

FindBugs (http://findbugs.sourceforge.net**).** This tool is also widely used and integrated with *Eclipse*, using the static analysis capabilities of Apache's Byte Code Engineering Library (BCEL) to inspect Java bytecode for occurrences of bug patterns. The latter are code idioms that are often errors. Bug patterns arise for a variety of reasons such as: difficult language features, misunderstood API methods, misunderstood invariants when

code is modified during maintenance, or garden variety mistakes as typos or use of the wrong boolean operator. Their authors report that its analysis is sometimes imprecise since many false positives (up to 50% of identified bugs) can be risen.

SISSy (http://sissy.fzi.de**).** According to its authors, the Structural Investigation of Software Systems tool can detect some well-known code smells and the violation of over 50 typical OO design principles, heuristics and patterns, such as bottleneck classes, god classes, *Data Class*es or cyclical dependencies between classes or packages. SISSy can analyze systems written in Java, C++ or Delphi but, as far as we could ascertain, is not integrated with any IDE.

Smelly (http://smelly.sourceforge.net**).** Is an Eclipse plugin that, according to its authors, is able to detect the following code smells in Java code: *Data Class*, *God Class*, *God Method*, *High Comment Density*, ***Long Parameter List*** and *Switch*. Only the one in bold matches the original name in the original code smells catalog [3].

Code Bad Smell Detector (http://cbsdetector.sourceforge.net/**).** This tool claims to detect five of Fowler et al. [3] code smells: *Data Clumps*, *Switch Statements*, *Speculative Generality*, *Message Chains*, and *Middle Man*, from Java source code. It has no recent downloads and appears to be associated with an ongoing PhD work. It is also not integrated with an IDE.

8 Conclusions and Future Work

Contents

8.1 Closing Remarks 78

8.2 Threats to Validity 78

8.3 Future Work 79

8.1 **Closing Remarks**

The idea of automating code smells detection by using metrics and tools is not new. However, the detection technique used in the Smellchecker tool is in contrast with all other known proposals due to the usage of a dynamic statistical process that relies on expert's knowledge that can be applied, theoretically[18], to any smell.

The main contributions of this dissertation are:

- A process for supporting the prediction of code smells occurances, based upon a logistic regression model, that allows progressive calibration either by a single user or by a user community;
- The proposal of an open source software architecture to support the aforementioned process for pratitioners using the Eclipse plugin architecture;
- A prototype based on the aforesaid architecture, for standalone Java developers using the Eclipse Environment;
- Validation of the process for the *Long Method* code smell.

8.2 **Threats to Validity**

Recalling Mäntylä's taxonomy, we expect that some code smells categories like the *Bloaters* or the *Couplers* will be much easier to detect than the others, using the proposed BLR based approach, simply because existing metric sets cover aspects such as code complexity and coupling between software components, that will act as good predictors. In other words, we cannot claim that it will be possible to find appropriate BLR based models capable to detect with a considerable degree of confidence, all the 22 Code Smells described by Fowler. Other detection techniques will probably be more suited in those cases. For example, *Duplicated Code* is an active area of research with sophisticated mechanisms already derived to identify code clones [46], which do not rely on source code metrics.

We have only collected evidence of the adequacy of the *Long Method* predictor presented herein for some software systems / components. Further evidence must be

[18] Further empirical studies are required to validate this assumption.

collected to calibrate our regression model before we can claim some sort of generality (applicability beyond our samples).
Metrics selected for the Smellchecker tool derive from their availability in a prior existing Eclipse Plugin and only represent a small subset of the available collection of metrics in literature [47]. These concern classic complexity measurements that are not suited to detect all code smells.

8.3 **Future Work**

The developed tool and its underlying process, rises different working opening opportinities. Concerning the *Long Method* code smell:

- Continuous refinement of the *Long Method* prediction model with extended calibration data,
- Validation of the regressors with different data sets,
- Refinement of the *Long Method* code smell detection model by adding and validating new predictors (more metrics),
- Studies of the sensibility of the model to bad data inputs,
- Experimentation with software from different domains to check if the mathematical prediction model of the *Long Method* code smell can be transferred across projects,
- Openings for more evolved regression techniques.

To further validate the regression process herein presented, an effort must be made to:

- Replicate the process for other code smells, deriving new prediction models for the remaining smells,
- Which implies adding, and the subsequent testing for correlation and validation, of new sets of metrics appropriated for each smell.
- To minimize the sensibility of the model to one experts opinion and over fitting issues Smellchecker's architecture must be distributed, so that a large set of participants can mitigate the natural bias that exists within small sets.

And to close the circle - further down the line - the mandatory integration with the proper refctoring techniques and tools that mitigate the code smells problem.

- According to [3, 12], a given code smell can be mitigated/removed by applying one out of a set of refactoring transformations. Since *Eclipse* supports several of those transformations, we envision that upon code smells identification, adequate refactorings could be suggested to remove the smell.

In the future, we will look at ways of computing the expected quality improvements attained by applying each of the refactoring alternatives. Hopefully that will allow to provide some advice for the developer.

9 Bibliography

1. Beck, K. and C. Andres, *Extreme programming explained: embrace change*. 2004: Addison-Wesley Professional.
2. Ambler, S., J. Nalbone, and M. Vizdos, *Enterprise unified process, the: extending the rational unified process*. 2005: Prentice Hall Press Upper Saddle River, NJ, USA.
3. Fowler, M. and K. Beck, *Refactoring: improving the design of existing code*. 1999: Addison-Wesley Professional.
4. Lientz, B. and E. Swanson, *Software maintenance management: a study of the maintenance of computer application software in 487 data processing organizations*. Vol. 4. 1980: Addison-Wesley Reading MA.
5. Guimaraes, T., *Managing application program maintenance expenditures.* Communications of the ACM, 1983. **26**(10): p. 739-746.
6. Bennett, K. and V. Rajlich. *Software maintenance and evolution: a roadmap*. 2000: ACM.
7. Khomh, F., M. Di Penta, and Y. Guéhéneuc. *An exploratory study of the impact of code smells on software change-proneness*. 2009: IEEE.
8. Mens, T. and T. Tourwé, *A survey of software refactoring.* Software Engineering, IEEE Transactions on, 2005. **30**(2): p. 126-139.
9. Counsell, S., et al., *Is a strategy for code smell assessment long overdue?*, in *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*. 2010, ACM: Cape Town, South Africa. p. 32-38.
10. Bryton, S., F. e Abreu, and M. Monteiro. *Reducing Subjectivity in Code Smells Detection: Experimenting with the Long Method*. 2010: IEEE.
11. Mantyla, M., J. Vanhanen, and C. Lassenius. *Bad smells-humans as code critics*. 2004: IEEE.
12. Wake, W., *Refactoring workbook*. 2003: Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.
13. Harter, D., M. Krishnan, and S. Slaughter, *Effects of process maturity on quality, cycle time, and effort in software product development.* Management Science, 2000. **46**(4): p. 451-466.
14. Fenton, N. and S. Pfleeger, *Software metrics: a rigorous and practical approach*. 1998: PWS Publishing Co. Boston, MA, USA.
15. McCall, J., et al., *Factors in software quality*. 1977: General Electric, National Technical Information Service.
16. Boehm, B., et al., *Characteristics of software quality*. 1978.
17. Opdyke, W., *Refactoring object-oriented frameworks*. 1992, Citeseer.
18. Roberts, D. and R. Johnson, *Practical analysis for refactoring*. 1999: University of Illinois at Urbana-Champaign.
19. Rothermel, G. and M. Harrold, *A safe, efficient regression test selection technique.* ACM Transactions on Software Engineering and Methodology (TOSEM), 1997. **6**(2): p. 173-210.
20. Kerievsky, J., *Refactoring to patterns*. 2005: Pearson Education.

21. Gamma, E., et al., *Design patterns: elements of reusable object-oriented software*. Vol. 206. 1995: Addison-wesley Reading, MA.
22. Laddad, R., *AspectJ in action*. Vol. 512. 2003: Manning.
23. Monteiro, M. and J. Fernandes. *Towards a catalog of aspect-oriented refactorings*. 2005: ACM.
24. Garcia, J., et al., *Toward a catalogue of architectural bad smells.* Architectures for Adaptive Software Systems, 2009: p. 146-162.
25. Demeyer, S., S. Ducasse, and O. Nierstrasz. *Finding refactorings via change metrics*. 2000: ACM.
26. Mens, T., G. Taentzer, and O. Runge, *Analysing refactoring dependencies using graph transformation.* Software and Systems Modeling, 2007. **6**(3): p. 269-285.
27. Mäntylä, M., *Bad smells in software–a taxonomy and an empirical study.* 2003.
28. Deursen, A., et al., *Refactoring test code.* 2001.
29. Van Emden, E. and L. Moonen. *Java quality assurance by detecting code smells*. 2003: IEEE.
30. Dudziak, T. and J. Wloka, *Tool-supported discovery and refactoring of structural weaknesses in code*. 2002, February.
31. Moha, N., et al., *DECOR: A method for the specification and detection of code and design smells.* Software Engineering, IEEE Transactions on, 2010. **36**(1): p. 20-36.
32. Brown, W. and I. Books24x7, *AntiPatterns: refactoring software, architectures, and projects in crisis*. Vol. 20. 1998: Wiley Chichester.
33. Roberts, D., J. Brant, and R. Johnson, *A refactoring tool for Smalltalk.* Theory and Practice of Object systems, 1997. **3**(4): p. 253-263.
34. Tourwé, T. and T. Mens. *Identifying refactoring opportunities using logic meta programming*. 2003.
35. Simmonds, J. and T. Mens, *A comparison of software refactoring tools.* Reporte Técnico. URL: ftp://prog. vub. ac. be/tech_report/2002/vub-prog-tr-02-15. pdf, 2002.
36. Moore, I. *Automatic inheritance hierarchy restructuring and method refactoring*. 1996: ACM.
37. Murphy-Hill, E., *Improving refactoring with alternate program views.* Research Proficiency Exam, Portland State University, Portland, OR, 2006.
38. Dhambri, K., H. Sahraoui, and P. Poulin. *Visual detection of design anomalies*. 2008: IEEE.
39. Slinger, S., *Code smell detection in Eclipse. Master's thesis.* Delft University of Technology, 2005.
40. Koschke, R., *Survey of research on software clones.* Duplication, Redundancy, and Similarity in Software, 2006.
41. McConnell, S., *Code complete: a practical handbook of software construction.* 2004.
42. Simon, F., F. Steinbrückner, and C. Lewerentz. *Metrics based refactoring*. 2001: Published by the IEEE Computer Society.
43. Wuyts, R., *A logic meta-programming approach to support the co-evolution of object-oriented design and implementation.* PhD, Vrije yiversity of Brussel, 2001.
44. Marinescu, R., *Measurement and quality in object-oriented design.* 2005.
45. Mäntylä, M. and C. Lassenius, *Subjective evaluation of software evolvability using code smells: An empirical study.* Empirical Software Engineering, 2006. **11**(3): p. 395-431.

46. Kim, M., et al., *An empirical study of code clone genealogies.* ACM SIGSOFT Software Engineering Notes, 2005. **30**(5): p. 187-196.
47. Henderson-Sellers, B., *Object-oriented metrics: measures of complexity*. 1996: Prentice Hall.