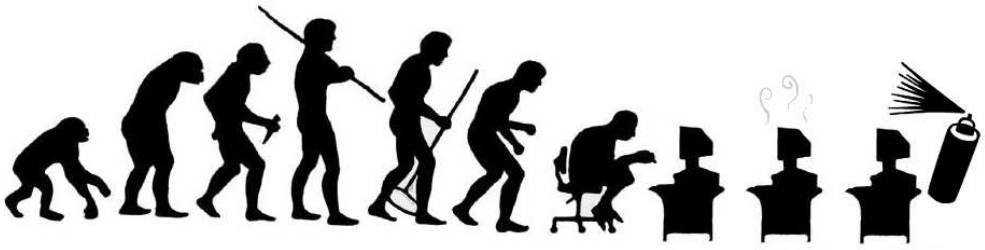
Evaluating the Lifespan of Code Smells in a Software System using Software Repository Mining

Master's Thesis — June 28, 2011



Ralph Peters

Evaluating the Lifespan of Code Smells in a Software System using Software Repository Mining

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ralph Peters
born in Dordrecht, the Netherlands

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

Cover picture: The evolution of code smells.

Evaluating the Lifespan of Code Smells in a Software System using Software Repository Mining

Author: Ralph Peters
Student id: 1524771
Email: ralphpeters85@gmail.com

Abstract

An anti-pattern is a commonly occurring solution that will always have negative consequences, when applied to a recurring problem. Code smells are considered to be symptoms of anti-patterns and occur at source code level. The lifespan of code smells in a software system can be determined by mining the software repository on which the system is stored. This provides insight into the behaviour of software developers with regard to resolving code smells and anti-patterns. This thesis presents a custom built application that computes the lifespans of certain types of code smells in a software repository. As a case study, this tool is applied on seven open source systems in order to answer research questions concerning the lifespan of code smells and the refactoring behaviour of developers. The results of this study reveal that engineers are aware of code smells, but not very concerned with their impact, given the low refactoring activity. Finally, several suggestions are given to further develop the application and to extend the work done in this thesis.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. L. Hollink, Faculty EEMCS, TU Delft

Preface

From the very first time I set foot on the holy grounds of information and communication technology, I knew - albeit subconsciously - that I wanted to pursue a career in this field.

Throughout the years I learned that being a software engineer is all but trivial. You can see it daily: those tiny programming obstacles that turn out to take more time than planned or even invalidate a part of your initial design, the massive communities in which colleagues around the world seek solutions to their problems and the mild but pressing reactions of your customers who forgive you for misinterpreting their requirements.

Spawned from the areas of mathematics and electrical engineering, computer science has grown into a mature scientific discipline with its own subdivisions and continues to grow beyond our imagination. I, for one, feel proud and privileged to be part of that development. A development that is also recognised by my parents, who always allowed me to find my own path - both inside and outside the walls of the faculty - and supported me all the way. A written commendation is not enough to describe the value of this positive attitude.

As a graduate student at the Delft University of Technology I was constantly surrounded and inspired by capable lecturers and likeminded and motivated peers, who valued the idea of a good education as much as I did. My professional and personal development would not only have been boring without them, but also less fruitful. If there is one thing I learned from them is that you don't hope for anything, but make it happen! Moreover, one tends to forget those people operating in the shadows, but always in a professional and helpful manner. The contributions of teaching assistants, system administrators and facility staff members are only noticed when they themselves are not seen for a while, at which point it becomes clear how essential their work is. To these people, I need to express my sincere thanks for all their assistance.

After many years of learning, the crown of my academic life was ready to be created. This thesis is the end result of that strenuous challenge. The expertise and flexible guidance of Andy Zaidman as my daily supervisor throughout the duration of this graduation project ensured its successful outcome. Thank you, Andy! The application built for this project integrates external tools, developed at other universities. A big box of gratitude goes to Nikolaos Tsantalis and his team for their work on JDeodorant and for letting me use it. The same applies to Yann-Gaël Guéhéneuc and his willingness to immediately fix bugs in Ptidej on my request.

So now the time has come for me to say goodbye to my life as a full-time student. I have always believed in the virtues of hard work, clear communication and mutual respect for contradicting perspectives to obtain the best results as an engineer. Honesty compels me to say - virtues are still ideals after all - that I have succeeded in upholding that attitude most of the time, but also not at times.

In conclusion, it has all been a mind-blowing and mind-boggling experience. I wish strength, wisdom and courage to everybody who made this venture possible and to the ones who still need to embark on that journey.

Ralph Peters
Delft, the Netherlands
June 28, 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Evolution can be defined as the natural process of change in all life forms over successive generations [23]. It is based on the belief that all organisms evolve by means of natural selection, mutation and genetic drift [19]. They reproduce to make offspring, which have slightly different genetic structures than their parents. Moreover, spontaneous mutation can introduce random genetic traits to an individual. Depending on the external circumstances, these genetic differences may increase the chances of the descendants to survive and reproduce. This means that more members of the next generations will have the advantageous variations, which will eventually result in permanent changes within the entire population of organisms. Over time, populations branch off to develop into new species as they become geographically separated or genetically isolated from the original population.

Software evolution can be loosely defined as the study and management of the process of repeatedly making changes to software over time for various reasons [32]. Change, whether accidental or intentional, is inevitable in a software system. The successful evolution of software is becoming increasingly critical, given the growing dependence on software at all levels of society and economy [34] [40]. Software applications that were developed using new programming paradigms, such as *Object-Oriented Programming* (OOP), will eventually render most monolithic programs obsolete. However, these old legacy systems still exist in industry. Given the fact that technology and user requirements advance every day, the question arises how long it takes before a particular software system starts to decay and show negative results in its behaviour. The discipline of software evolution tries to provide theoretical knowledge and a set of best practices in order to understand the causes and consequences of this deterioration and take action to counter the negative effects.

Strictly speaking, “software evolution” is an ill-chosen name. The word “evolution” implies that subsets of software systems branch off and become independent applications, which are analogous in biological evolution to populations and new species, respectively [43]. Naturally, this phenomenon occurs on occasion in software development, but it is not within the scope of this graduation project. Given its definition, *software ageing* would be a better alternative. According to Bombardieri et al. [15], software ageing can be defined as a process that progressively reduces the capability of a software product to satisfy stated or implied user requirements and makes software changes more expensive and error-prone. After all, a software system usually starts small and expands under many influences, before

it eventually starts to grow old and shows a greatly increased complexity and a significant performance reduction. Nevertheless, the majority of the software engineering community considers software evolution to be every change to a system after its initial development, which includes software ageing. Therefore, the term software evolution will be used in this thesis rather than software ageing.

Parnas [40] identified two main aspects that lead the software evolution process:

1. *lack of movement*: unless software is frequently updated according to contemporary standards, its users will become dissatisfied over time and they will change to a new product as soon as the benefits outweigh the costs of switching over and retraining;

2. *ignorant surgery*: changes to software are made by people who do not understand the original design concept. This usually causes the structure of the software system to degrade, because the modifications often invalidate the initial design. Software that has been repeatedly changed and maintained in this manner is understood by less people over time and becomes very expensive to update. Changes take longer and are more likely to introduce new bugs.

The subject of this graduation project finds its origin in the second aspect. There are many types of changes that introduce inconsistencies into the behaviour or the source code of a software program. Examples include unforeseen exception cases, conflicting naming conventions and *anti-patterns*. *Code smells* are identified as symptoms of anti-patterns [36]. This graduation project deals with the lifespan of certain types of code smells, which can be revealed by *mining software repositories*. In the following sections, the terminology, context, goals and structure of this thesis and the research questions are outlined.

1.1 Terminology

1.1.1 About Anti-patterns

The term *anti-pattern* is defined by Brown et al. [16] as a commonly occurring solution that will always generate negative consequences when it is applied to a recurring problem. Nevertheless, people use anti-patterns because they appear to be the right approach. As such, they are closely related to *design patterns*, which are reusable solutions that do not cause counterproductive results when applied to recurring problems [24].

An anti-pattern always has two key features:

1. *Repeated patterns* of actions, processes or structures can be found that initially appear to be useful, but ultimately produce more bad consequences than beneficial results;

2. A *refactored* solution is available that is clearly documented, repeatable and proven in practice.

In software development, anti-patterns occur in the source code of a software system. Examples of anti-patterns in software design and OOP are circular dependencies between software modules, using design patterns in the wrong place and making objects so powerful that they are very difficult to implement.

1.1.2 About Code Smells

How can anti-patterns be found in a software program? Browsing source code is never an easy task for developers, regardless of their knowledge of the system. One would expect something out of the ordinary with regard to the adopted programming standards. In other words, there must be parts of the code that "smell" bad. Hence, these indicators have been given the name *code smells* by Beck and Fowler [22].

Code smells are related to anti-patterns in the same sense that sneezing is related to a cold. In general, if there are several symptoms, then a deeper problem is usually the cause. Thus, when a developer wants to refactor an anti-pattern, she only has to find certain code smells and start looking for the issue that caused it. In order to solve the problem, it is not the code smells that must be refactored, but the anomaly that causes them.

Examples of code smells are large methods, classes with a poor sense of information hiding and code inheritance that is hardly used in practice. Software entities like classes, methods and variables that suffer from a code smell are also known as *code smell instances* [14]. Code duplication can be considered as a special case of code smell, since it is one of the most pervasive smells that can be found in any software system [22]. This led to extensive research, solely dedicated to duplicated code [21] [27] [30]. This type of code smell is outside the scope of this thesis.

1.1.3 About Software Repositories and MSR

In most cases, the source code of a software system is stored and maintained in a *software repository*, which takes the form of a public of private server. In a software development process that requires working in teams, project members tend to choose for a *version control system* (VCS), such as *Subversion* (SVN) [7]. A VCS allows them to store multiple versions of the software product they are developing and work on new releases without concurrency problems.

The main concept is as follows. Whenever developer A wants to work on a part of the software system, he *checks out* the source code from the VCS. That is, he downloads a copy of the source code onto his local workstation. This copy is also called the *working copy*. After developer A has made his changes to the source code, he needs to check it back in into the repository, which is known as *committing*. This causes the VCS to create a new *revision*, which is the state of the contents of the repository at some point in time. The latest revision is also called the *head* revision. After developer A committed his work, the head revision can be checked out by developer B, which contains the changes made by developer A. However, developer B can also choose to check out an earlier revision for historical reasons or compare its contents with those of another revision. Naturally, a project team has to adopt certain conventions to be able to work effectively with a VCS.

Next to holding the source code of a software application, a VCS implicitly stores data about its contents and the commits. These *metadata* exist for the entire duration of the project and mainly provide administrative information, such as the filenames of the contents that have been changed between two versions, how and when they were changed and the developer that made the changes. Research has been devoted to devise methods to extract

this information and uncover evolutionary relationships, similar to the field of data mining [26]. Hence, this approach has been given the name *Mining Software Repositories* (MSR). This technique has been operationalised to allow software engineers to obtain the metadata of a VCS. An example is *SVNKit* [8]: a Java library that provides functions, which access a VCS and return metadata to the user. SVNKit has been used for that purpose in this graduation project.

1.2 Problem Statement

Like humans, software grows old and will start to show more irregularities at some point in time. The disproportionally growing presence of anti-patterns and code smells into a software system is a common characteristic of the ageing process and is usually the result of changing requirements, pressing deadlines and original designers leaving the project over time [13]. However, there are a number of practices available to slow down or reverse the ageing process, like redocumentation and refactoring [40].

Studying the evolution of a software project usually entails looking into the past and comparing it to a future situation. This can be achieved by mining a software repository, which can lead to the discovery of *logical coupling*. Ambros et al. [18] describe this kind of coupling as implicit and evolutionary dependencies between the artifacts of a system which, although potentially not structurally related, evolve together and are therefore linked to each other from an evolutionary point of view. As such, logical coupling might not be immediately clear from the source code, but has a meaning nevertheless.

The lifespan of code smells in a software repository is something that can be deduced, based on the results of mining software repositories. A code smell infection may occur up to a particular revision, after which its underlying cause is refactored by a software engineer and thus no longer be present in the next version. The priority that is attached to a refactoring effort depends on a number of factors, such as the nature of the code smell, the developer's insight and deadline pressures. This leads to the following research question:

RQ1 *Are some types of code smells refactored more and quicker than other smell types?*

An implicit sub-question is: *How long (i.e. in terms of time and number of revisions) do certain code smells live inside a software system?* To answer these questions, a trend among several distinct software repositories needs to be found between the instances of a particular code smell and their lifespans. For example, if multiple software entities infected by code smell X exist in more than fifty revisions and instances infected by code smell Y exist in less than twenty revisions of a VCS, then this may be evidence of the general attitude of developers towards the severity of both smells. The strategy to take here is to compute the average lifespans of multiple code smells in various software systems and compare them with each other.

A related issue is the point in a system's life cycle at which code smells are refactored. The corresponding research question is as follows:

RQ2 *Are code smells refactored more at an early or a later stage of a system's life cycle?*

Again, a pattern must be discovered among different repositories. The only difference is that the occurrence of a code smell is considered at a particular point in the development life cycle. For example, if there are instances of the same code smell present in the earlier versions as well as in the later versions of a system, it is interesting to know how many revisions the instances survive. The same strategy for RQ1 also applies here, except it only considers a subset of revisions. The average lifespans of several code smells within the early and later revisions in a software system must be determined and compared with each other. This must then be repeated for other software projects. The nature of the life cycle of all investigated systems must be taken into account, because not every project team starts using a VCS before the system reaches a certain state of maturity.

Of course, not every software developer has the same knowledge and experience. What one engineer sees as a bad coding practice may be perceived by the other as a necessity. However, it is interesting to find out if one developer solves a certain type of smell quicker than the other, which leads to the next research question:

RQ3 *Do some developers refactor more code smells than others and to what extent?*

To give a fair answer to this question, it is useful to determine the common commit behaviour of developers. Do they commit changes per task or file? Are the changes small or do they comprise the entire refactored solution? In general, whenever an infected instance stops being a code smell in a particular revision, the name of the developer who made the commit has to be retrieved. This approach must be taken for all instances in multiple software repositories and will result in a list of developers, along with the number of code smell instances they resolved. Of course, their project involvement and the rationale behind the smell removal must also be taken into account.

Finally, there are a number of reasons for refactoring code smells. The corresponding research question is:

RQ4 *What refactoring rationales for code smells can be identified?*

For example, a code smell needs to be refactored in order to add or test functionality. Also, a dedicated refactoring phase in the development life cycle may be introduced. This is where the logical coupling plays a role. For instance, if several code smells cease to exist within a low number of consecutive versions, then this might hint towards a planned refactoring stage. Next to this, if a change commit activity occurs almost immediately after the removal of a code smell, then this may indicate a refactoring activity to accommodate new functionality. The focus here lies on code smells that are intentionally refactored and not accidentally.

1.3 Thesis Project

1.3.1 Project Goals

The goals of this graduation project are as follows. A Java application named *SACSEA* is developed, which is then applied to seven software projects in order to answer the research questions. This tool is built using the Eclipse IDE. Its main feature is that it searches for code smell instances within a user-specified range of revisions of a software repository and computes their lifespans. The output will be a visual report containing the lifespans of the infected instances and the corresponding revisions. Also, metadata regarding the committed changes will be returned. To answer the research questions, experiments are devised and performed. These will consist of applying SACSEA on the source code of seven distinct Java systems, processing the results and drawing conclusions from them.

1.3.2 Relevance for the Faculty

The contribution of this research is to provide insight into the perspective and awareness of software developers on the severity of different code smells. The outcomes of this project can be used as a stepping stone for further work in the field of software evolution.

1.4 Thesis Structure

The outline of this thesis is as follows. The next chapter presents background information on the code smells and techniques involved in this graduation project. As mentioned before, the approach for obtaining an answer to the research questions includes determining the lifespan of code smell instances in different software systems. This is done using a custom built tool, which is described in full detail in chapter 3. Upon completion of this application, it can be used to retrieve the desired data from SVN repositories. Chapter 4 reports on the setup and the results of the experiments that are devised to help answer the research questions. Threats to the validity of this study can also be found in that chapter. Naturally, similar work has been done in the field of software evolution and the investigation of code smells and anti-patterns. The contributions that are most related to this graduation project are outlined in chapter 5. Conclusions are drawn from the results of the experiments in chapter 6, which also considers the research questions in retrospect and answers them by referring to the results. Next to this, the contribution of this graduation project to the field of software evolution is given, along with recommendations for future work.

Chapter 2

Background

This chapter provides theoretical background information and a deeper understanding on the subjects related to this graduation project. In section 2.1, a basic description of *code smells* in general is given and the specific types of code smells used in this graduation project are elaborated on. Special techniques operationalised as software applications exist in order to find code smells and anti-patterns. Section 2.2 shows the nature and origin of these so-called *code smell detection tools* that were used for the application developed in this graduation project. Finding information about the lifespan of code smells in software projects was made possible by *mining software repositories*. Section 2.3 explains this technique and the utility SVNKit [8] through which MSR can be applied.

2.1 Code Smells

There is no widely accepted definition of code smells. In the previous chapter, code smells are described as symptoms of a deeper problem, also known as an anti-pattern. In fact, code smells can be considered anti-patterns at programming level rather than design level. Smells such as large classes and methods, poor information hiding and redundant message passing are regarded as bad practices by many software engineers. However, there is some subjectivity to this determination. What developer A sees as a code smell may be considered by developer B as a valuable solution with acceptable negative side effects. Naturally, this also depends on the context, the programming language and the development methodology.

The interpretation most widely used in literature is the one by Beck and Fowler [22]. They see a code smell as a structure that needs to be removed from the source code through refactoring to improve the maintainability of the software. They also claim that informed human intuition is the best tool to label a piece of source code as a code smell and measure its intensity. This is a plausible statement, but it does not render automatic measurement impossible or redundant. Most code smells can be measured by using *software metrics*. For example, a large class is bound to have many *lines of code* (LOC), a metric whose value can easily be computed by an automated code smell detection utility. The specific detection of code smells by such tools is usually based on a collection of metrics who each have a certain threshold. If enough thresholds are exceeded, then the detection tool marks the

code fragment under investigation as a code smell, which can then be resolved by applying a suitable refactoring. Beck and Fowler also provide a detailed instruction set for various code smells on how to refactor them.

The following subsections describe the code smell types that were considered in this graduation project, along with their commonly accepted refactorings and metrics.

2.1.1 God Class

In object-oriented programming, a class should usually encompass only one design concern. However, when a system grows over time, this principle can easily be violated. This results in large, complex, inelegant and low-cohesive classes that are difficult to understand and maintain. In other words, a class is a *God Class*, if it implements more than the concept it was originally designed for.

There are basically two types of God Classes:

1. A class that contains many of the system's data as attributes without implementing much logic is called a *Data Class*;

2. A class that implements a large part of the functionality of a software system. This is typically characterised by a great number of complex methods and possibly also many attributes. This kind of God Class is called a *Behavioural God Class*.

Whenever a God Class is mentioned in this thesis, a class of the second type is meant. Data Classes form a specific type of code smell on their own. They are considered as a separate smell in this thesis and will be introduced in subsection 2.1.3.

Demeyer et al. [20] mention several criteria for identifying a Behavioural God Class and how to refactor it. It is typically a procedural giant that assumes too many responsibilities. The infected instance can be a single class implementing an entire subsystem, consisting of hundreds of variables and methods, as well as duplicated code. Due to its size and complexity, a God Class often takes a long time to compile and uses a lot of memory. Almost every change to the software application may also necessitate a change to this class. Since it covers so many design concepts, it is difficult to reuse, test and maintain it. Usually, developers can not give a clear and concise answer on its purpose. God Classes often occur in combination with Data Classes. In such cases, a God Class is in charge of handling a large part of the system and treats multiple Data Classes as glorified data structures.

The most natural refactoring methods for God Classes rely on moving behaviour and responsibilities away from the infected class. This implies that functionality is moved to collaborating classes and that new classes can also be extracted from the God Class as cohesive and independent pieces of functionality. The latter is a refactoring method called *Extract Class*. If the component makes more sense as a subclass, then this is also possible. Next to this, redundancy such as duplicated code should be minimised, which can be done by extracting methods that implement this code once and calling those methods whenever it is needed.

Lanza and Marinescu [31] present a method to identify God Classes, based on three main characteristics and the use of metrics:

1. The infected class uses more than a few attributes of other simpler classes. The *Access To Foreign Data* (ATFD) metric is used to measure this aspect and counts the number of accesses of a class to foreign data.

2. The infected class is large and complex. This is expressed using the *Weighted Method Count* (WMC) metric, which represents the sum of the cyclomatic complexities of the methods of a class. The cyclomatic complexity can be defined as the number of linearly independent execution paths in a program's source code.

3. The infected class has much non-communicative behaviour, i.e. there is low cohesion between the methods of that class. The *Tight Class Cohesion* (TCC) metric is used, which measures the relative number of methods accessing the same data field.

The rationale behind the first metric is that a strong dependence on data of other classes is the most significant symptom of a God Class. As for the other two metrics, small classes are discarded because they are less relevant and cohesive classes are ignored because high cohesion indicates internal harmony between the parts of the class.

The values of all three metrics have to exceed a certain threshold before a class can be labelled as a God Class. The ATFD value is directly proportional to the probability that a class is or will become a God Class. Therefore, the threshold has been set to a few accesses to foreign data. In other words, a God Class uses more than a few attributes of other classes. The WMC metric has a minimum limit of "Very High", as God Classes have an exceedingly high complexity. This value usually depends on the context, but a trained software engineer can easily estimate the cyclomatic complexity by finding an extremely high amount of (nested) conditional and iteration statements. Lanza and Marinescu provide a statistical WMC threshold of 47 in their study based on 45 Java projects. The value of the TCC metric ranges from zero (no cohesion) to one (fully cohesive). A threshold of one third is sufficient for the detection of a God Class. It indicates that in an infected class less than a third of the method pairs have the usage of the same attribute in common.

2.1.2 Feature Envy

The idea behind object-oriented programming is that data and the processing operations on those data are kept together in objects as much as possible. This implies that a system typically should have low *coupling*, which is the dependence between classes. A method violates that concept if it is more interested in a class other than the one it is actually in. In other words, the method is *envious of features* provided by a foreign module and suffers from *Feature Envy*. As such, this code smell represents a form of high coupling.

As with God Classes, the most significant aspect of Feature Envy is that it is a sign of a poor distribution of a system's intelligence. The most common focus of the envy is data, which are usually accessed directly or through accessor methods. Detection is based on counting the number of foreign data attributes that are used by a method. Excessive use of remote data while ignoring local data results in methods that are implemented at the wrong place. A change in those methods triggers changes in other methods. The same applies to

bugs, as they will also be propagated and ripple through the call chains. This is also called the *ripple effect* [31].

Since this smell falls into the category of an improper intelligence distribution, any infected method would benefit from being moved closer to the data it is so interested in, as proposed by Beck and Fowler [22]. This refactoring is also called *Move Method*. Sometimes only a part of the method suffers from envy. For example, this can occur if the accessed data belong to more than a few foreign classes. In that case, it is wise to first extract that portion as a separate method (*Extract Method*) and then move it to the class of its interest. If there is still confusion on where to place the envious method, then the class with the most data should be chosen as the destination. Moving operations closer to the data can help minimise ripple effects and maximise cohesion.

Distinguishing Feature Envy Methods from other methods can be difficult to do through code inspection. Lanza and Marinescu [31] propose three characteristic symptoms of this smell, as well as three metrics with appropriate thresholds:

1. The method uses more than a few foreign attributes. The *Access To Foreign Data* (ATFD) metric is the most suitable to measure this aspect and counts the number of accesses of a method to foreign data.

2. The method accesses more foreign than local attributes. The *Locality of Attribute Accesses* (LAA) metric relates the ATFD metric to the total number of accessed data in the analysed method.

3. The accessed foreign attributes belong to only a few other classes. The *Foreign Data Providers* (FDP) metric reflects this and counts the number of unrelated classes that contain the accessed foreign data.

The third condition is introduced because a distinction has to be made between a method that uses data from many different classes and a method that only envies two or three classes specifically. In the first case, the method acts like a controller operation, implementing more functionality than it was originally designed for. The second case identifies more pure Feature Envy Methods, as those operations are simply misplaced. This is reflected by a narrowly targeted dependency on the data of another class.

Finally, all three metrics have appropriate thresholds, which are all exceeded if a method suffers from Feature Envy. Such a method has an ATFD value that is greater than a few (about two to five) used data members. Its LAA limit is one third. This metric ranges from zero (all used attributes are foreign) to one (all used attributes belong to the enclosing class). The FDP threshold is set to a few (about two to five) foreign classes, which is not exceeded if a method is envious.

2.1.3 Data Class

According to Beck and Fowler [22], a *Data Class* is a class that only contains fields and corresponding getting and setting methods. They also claim that each class requires effort to maintain and understand. When a class is not doing enough, it needs to be removed

or its responsibility needs to be increased. As explained previously, Data Classes are a type of God Class, except that they do not contain complex functionality. Such classes are dumb data holders, but are almost certainly being manipulated in far too much detail by other classes. The maintainability, testability and understandability of a software system is reduced due to the presence of this code smell.

The lack of functionally relevant operations in a Data Class may indicate that related data and behaviour are kept in separate places [31]. This is possibly an indication of a design that is not object-oriented. Like Feature Envy Methods, Data Classes are the manifestation of a poor data-operation proximity. Encapsulation and data hiding of attributes may also lack in such classes, while these are fundamental principles to obtain a good object-oriented design. Data Classes break these concepts, because they let other classes see and possibly manipulate their data, leading to a fragile design.

Beck and Fowler [22] mention several ways to help refactor a Data Class. In its early stages, the class may have public fields, which should be made private or protected as soon as possible using the refactoring *Encapsulate Field*. Collection attributes should also be properly encapsulated. *Remove Setting Method* has to be applied to any field that should not be changed. A more important refactoring method is to find out where accessor and mutator methods are used by other classes. The designer can then move functionality from those classes into the Data Class by using *Move Method*, preceded by *Extract Method* if necessary. This will increase data-operation proximity.

Lanza and Marinescu [31] try to identify classes that provide almost no functionality through their interfaces, as well as classes that define many (public) data fields and getting and setting methods. There are two aspects that can help with the detection of Data Classes:

1. A large part of the interface of the infected class shows data rather than operations. This is measured by the *Weight Of Class* (WOC) metric and represents the ratio of the number of non-accessor methods to the total number of interface members.

2. The class reveals many attributes and is not complex. The absolute number of data and getting and setting methods should be high. There are two cases:

   (a) The classical Data Class is not very big, has almost no functionality and only provides some data, accessors and mutators. Here, there is bound to be little public data and the class does not have a high *Weighted Method Count* (WMC) value. Therefore, the only requirement is that the class has more than a few public data holders, expressed by the sum of the *Number Of Public Attributes* (NOPA) and *Number Of Accessor Methods* (NOAM) metrics.

   (b) The class is rather large and looks “normal”. It defines some functionality, but its public interface contains a significantly high number of data, accessors and mutators, apart from the provided services. In this case, if the class is to be considered a Data Class, it needs to provide a lot of public data. At the same time, the complexity of the class (WMC) may be considerably high, up to the limit of excessively high. If the complexity exceeds this limit, then the class does not conceptually fit the Data Class definition.

In each of the two cases, it holds that a low WOC value is needed, since it indicates low functionality in a class. This metric ranges from zero (all methods are accessors or mutators) to one (all operations are non-accessor methods). The threshold is set to one third. The two cases represent the distinction between small and large Data Classes. On the one hand, if a class implements little functionality, then it does not have to define many data attributes and corresponding accessors and mutators. The WMC value can be at most "High" (statistically, 31 according to Lanza's and Marinescu's study based on 45 Java projects), while the sum of the NOPA and NOAM values should be at least more than a few. On the other hand, if a class contains more than little functionality, the number of data and getting and setting methods needs to be large as well. Here, the WMC threshold can be slightly higher, as it is set to "Very High" (statistically 47). The sum of the NOPA and NOAM values should then be at least more than many.

2.1.4 Message Chain Class

Khomh et al. [29] state that a *Message Chain Class* occurs when its (data access) functionality is implemented using a long chain of method invocations or temporary variables between different classes. A Message Chain Class may impact change- and fault-proneness due to the high number of indirections. This makes the code dependent on relationships between many potentially unrelated objects and reduces a developer's view of the context, which may lead to more defects.

Beck and Fowler [22] propose that this smell can be refactored by using *Hide Delegate*. This makes the caller depend solely on the object at the head of the chain. For example, rather than *a.b().c().d()*, method *d()* is placed on object *a* and possibly also on objects *b* and *c*. In principle, this can be done on every object in the chain, but this often turns each intermediate object into a middle man. A better alternative is finding out what the resulting object is used for. *Extract Method* may then be used to extract a piece of the code that uses it, after which *Move Method* is used to push it down the message chain.

The use of metrics for finding Message Chain Classes is unnecessary, since they can be detected by computing the number of transitive calls of a class to other classes. Naturally, a threshold of this number is needed. The code smell detection tool Ptidej used in this graduation project upholds a limit of three invocations. In other words, if four or more invocations are made, then the original calling class is considered a Message Chain Class.

2.1.5 Long Parameter List

In the procedural programming era, it was a common convention to pass all data needed by routines as parameters [22]. The alternative was to use global data, which was an unfavoured practice. With the arrival of object-oriented programming, local and foreign objects can be used for data access. The host class of the method usually contains the majority of what is needed and only the data that are highly essential to an operation need to be passed as parameters. This implies that the size of parameter lists in object-oriented programs is much smaller than in traditional programs. In object-oriented programs, operations with more than three parameters are generally considered to suffer from this smell, taking exceptional

cases into account. That threshold is assumed in this graduation project. Long Parameter Lists tend to become inconsistent and difficult to use and are constantly updated as more data are needed. This makes them hard to understand and maintain.

The refactoring *Replace Parameter with Method* can be applied when data from one parameter can be obtained by making a request to an existing object. This object may be a field or another parameter. Furthermore, *Preserve Whole Object* can be used to gather a bunch of data from one object and replace these data with the object itself. If there are several parameters with no logical object, a new dedicated object can be created through the refactoring *Introduce Parameter Object*, which is then passed as a parameter.

There is one important exception to such refactorings. If a dependency from the called object to the larger object is unfavoured, then unpacking data fields and sending them along as parameters is reasonable with the negative consequences in mind.

2.2 Code Smell Detection Tools

Code smells can be detected manually, but this requires the software engineer to have an experienced eye and usually a deeper knowledge of the system. Extensive research has been devoted to develop several techniques and utilities to do this automatically. Most code smell detection tools depend on the use of software metrics and corresponding thresholds.

Because no application yields perfect results, human intuition should never be replaced. However, the approach proposed in this graduation project requires that all code smells in a software project are found over multiple revisions. This implies that the results are very data-intensive and the process of finding them is time-consuming. For an efficient and thorough study, these results have to be structured and stored in an appropriate way. Therefore, it is better that code smell instances are found automatically, so that they can be forwarded to a logging layer immediately.

The following subsections provide background information on the detection utilities that were used in this graduation project. These tools were chosen because of their availability, their high effectiveness and their relatively favourable computation time and memory usage.

2.2.1 JDeodorant

JDeodorant is an Eclipse plug-in that employs a variety of novel methods and techniques in order to identify code smells in Java programs and to suggest appropriate refactorings that resolve them [44]. Moreover, the tool pre-evaluates the effect on design quality of all refactoring suggestions, assisting the user to determine the most effective sequence of refactoring applications. JDeodorant has been developed at the Department of Applied Informatics of the University of Macedonia in Greece. Its source code is available upon request for researchers and academics after agreeing to a Licence Agreement for Academic Use.

In order to control the number and the quality of the reported refactoring opportunities, JDeodorant provides a preference page where the user can define various threshold values. For example, the minimum number of statements that a method should consist of in order to

be examined for potential refactoring opportunities can be customised. For this graduation project, the default values were used.

JDeodorant mainly employs APIs belonging to the Eclipse Java Development Tools (JDT) Core, which defines the non-UI infrastructure. The plug-in uses the *ASTParser* class to analyse the relationships between system entities and apply refactorings on source code. Next to this, it employs the *ASTRewrite* class to apply the refactorings and provide undo functionality. As such, JDeodorant depends heavily on the notion of an *abstract syntax tree* (AST). Information from the AST may be reused on several occasions for a different purpose. Therefore, the architecture of JDeodorant supports the reuse of this information without permanently storing it in memory. This is achieved by providing an intermediate representation of the required Java elements and a mechanism that enables the recovery of AST nodes in a quick and efficient manner.

Currently, the tool can detect four kinds of code smells, namely God Classes, Feature Envy Methods, Type Checking code and Long Methods. Moreover, it immediately determines possible refactorings and presents them to the user, who can decide to let JDeodorant apply them or not. God Classes are resolved through appropriate *Extract Class* refactorings. Feature Envy Methods are fixed by suggesting appropriate *Move Method* refactorings. Type Checking problems can be rectified through *Replace Conditional with Polymorphism* and *Replace Type Code with State/Strategy* refactorings. Finally, Long Methods are resolved by suggesting appropriate *Extract Method* refactorings.

2.2.2 Ptidej

Ptidej [5] stands for *Pattern Trace Identification, Detection and Enhancement in Java* and is a set of tools to evaluate and enhance the quality of object-oriented programs, promoting the use of patterns at language level, design level and architectural level. Design patterns, anti-patterns and code smells can be detected in any Java source code using this utility. The development of Ptidej started in 2001 at the Department of Computer Science and Operations Research of the University of Montréal in Canada and the tool is currently still being improved. Its original goal was to study code generation and identification of patterns. Since then, it has evolved into a complete reverse-engineering tool suite that includes several identification algorithms. It includes the module *DECOR* (DEtection and CORrection), which allows the detection of design defects. Through its user interface, the user can create a model of a program from its source code and identify micro-architectures similar to a design pattern or call various generators, analyses and external tools on the program model.

Ptidej consists at least of the following relevant modules [25]:

- *Caffeine*: a dynamic analyser for Java based on a Prolog engine and the Java debug interface to define relationships among classes precisely.
- *Ptidej UI*: a library of graphic widgets to display models of programs and dynamic data from Caffeine.
- *PADL* (Pattern and Abstract-level Description Language): a meta-model to describe the structure of object-oriented programs.

- *POM* (Primitives, Operators, Metrics): a library of software metrics to compute well-known metrics on program models.
- *Ptidej Solver*: an explanation-based constraint solver to identify micro-architectures similar to motif models in program models.
- A library of generators and analyses to be applied on program models.
- A library of design motifs from design patterns, including Chain of Responsibility, Composite, Observer and Visitor.
- Various parsers to build models of programs from different representations of source code, including C++ files and Java class files.
- Several UIs to access the functionalities provided by the Ptidej tool suite:
  - Parse and create models of programs.
  - Enhance models of programs with dynamic data from program executions.
  - Visualise created models.
  - Identify micro-architectures similar to a design motif model in a program model.
  - Visualise the identified micro-architectures.
  - Call generators, analyses and external tools on models.

As of 2007, Ptidej includes algorithms for idioms, micro-patterns, design patterns and design defects [35]. Idioms are low-level patterns specific to some programming languages and to the implementation of particular characteristics of classes or their relationships. Micro-patterns are well-defined idioms pertaining to the design of classes in object-oriented programming. To identify design defects, a *Domain-Specific Language* (DSL) is used to specify and automatically generate detection algorithms using templates. A DSL offers greater flexibility than ad hoc algorithms because domain experts and software engineers can manually specify and modify the detection rules using high-level abstractions, taking the context, environment and characteristics of the analysed systems into account.

2.3 Mining Software Repositories

The term *Mining Software Repositories* (MSR) has been coined to describe a broad class of investigations into the examination of software repositories. This includes sources such as the information stored in source code version control systems (e.g. SVN), requirements and bug-tracking systems (e.g. Bugzilla) and communication archives (e.g. e-mail) [26]. Such repositories contain a wealth of information and provide a unique view of the actual evolutionary path taken to realise a software system. These data often exist for the entire duration of a project and can represent thousands of versions with years of details about the development process. These details include properties like individual versions of the system, the changes and metadata about the changes (e.g. who made the change, why was it made and when was it made).

Software engineering researchers have developed and evaluated various approaches to extract relevant information and uncover relationships and trends from repositories in the context of software evolution. For example, Begel et al. [12] developed Codebook. This framework can discover transitive relationships between people, source code, test cases, defects, documentation and related artifacts by mining all kinds of software repositories.

One may be interested in the growth of a system, the change relationship between source code entities or components reuse. This activity is analogous, but not limited to the field of data mining and knowledge discovery, hence the term Mining Software Repositories. The premise of MSR is that empirical and systematic investigations of repositories will shed new light on the process of software evolution and the changes that occur over time by uncovering pertinent information, relationships or trends about a particular evolutionary characteristic of the system.

Researchers utilise software repositories in multiple ways. The most straightforward one is to directly use the facilities of source code repositories to get a particular version of the code. The individual versions and corresponding metadata can then be used to answer questions of interest using the adopted methodology. Some researchers limit their study to the metadata that are directly available from the repositories. These metadata are analysed to filter the differences and source code in a semantic manner. For example, the SVN comments and the textual description of a related bug report in Bugzilla can be used to categorize the source code changes as an attribute of corrective-maintenance activity. Going a step further, the data and metadata directly available from Subversion can be processed to facilitate fine-grained source code difference analysis. Usually, raw data from the repository are transformed into a format that can be processed easily. The processing module then typically performs some mining algorithm on the retrieved data, after which the results are presented to the user.

The purpose of MSR reduces to the questions that can be answered with it. Two classes of such *MSR questions* can be distinguished:

- The *market-basket question* (MBQ): if event A occurs, then how many times do events B and C occur? The answer is given by using a set of rules or guidelines describing situations of trends or relationships.
- *Prevalence questions* (PQ): Were certain functions added, modified or removed? How many and which of the functions are reused?

This graduation project will mainly deal with the second type, as the point of interest is the introduction and removal of certain code smell instances in source code repositories.

Chapter 3

SACSEA Implementation

This chapter presents the custom developed tool that helps to determine the lifespans of different code smell instances in a particular software system. The application is named *SACSEA*, which is an acronym of *Semi-Automatic Code Smell Evolution Assistant*. It can find code smells in multiple revisions of a VCS, use MSR to obtain the change history of each smell instance and generate a graphical and textual report containing the lifespans of each instance, along with any relevant metadata. Section 3.1 describes the component decomposition and internal workings of the application in detail. Next to this, section 3.2 outlines design decisions, obstacles and limitations.

3.1 Toolchain Structure and Operation

This section presents the features and components of SACSEA. It is written in Java as an Eclipse plug-in. An overview of its operation is depicted in figure 3.1.

The operation of SACSEA consists of the following phases:

1. *Initialisation*: First, the URL of the SVN repository under investigation is entered into the UI, as well as the type of code smell that has to be found. Then, the numbers of two revisions are entered. SACSEA will search for code smells in every revision between these two numbers. In other words, the user specifies a range of revisions that have to be examined.

2. *Detection*: Each revision within the user-specified range is checked out from the repository and imported into the workspace of Eclipse as a Java project (and built if necessary in order to generate *.class*-files). Then, the detection modules try to find code smell instances in that particular project. The results are saved to an XML-file, after which the project is removed from the workspace. Finally, the subsequent revision is extracted from the VCS and the process repeats itself until the end of the range has been reached.

3. *Difference computation*: When all revisions have been examined for code smells, the differences between every two consecutive XML-files are determined and stored in a CSV-database, according to a custom format.

4. *Output generation*: The differences resulting from the previous phase represent the beginning or the end of the lifespans of code smell instances. Based on the changes, these lifespans are computed and translated to a visual chart. Also, metadata of their beginning and end revisions are printed in text, such as the developer responsible for the commit that introduced or removed the code smell and the commit date.

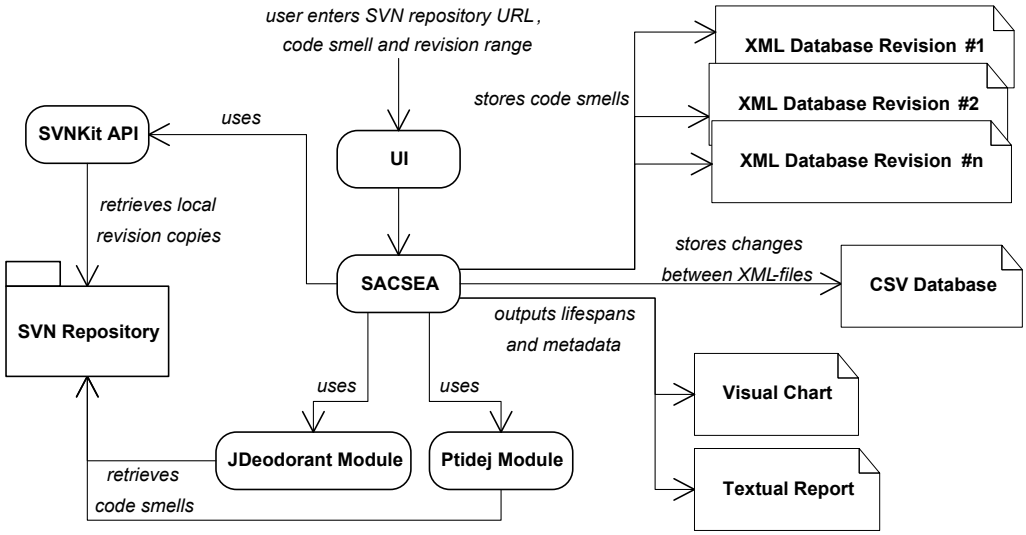


Figure 3.1: Overview of the operation of SACSEA.

The following subsections present an elaborate description of the steps outlined above.

3.1.1 Initialisation

SACSEA can be started by running it from source code as a Java application. This initialises a new Eclipse instance. Once SACSEA has started up, the UI becomes visible outside this new instance. As can be seen in figure 3.2, it consists of several text input fields, radio buttons and click buttons.

First, the URL of the SVN repository under investigation is entered, after which the button [LOAD!] is pressed. The application then retrieves the name of the SVN project and the current number of revisions and shows this information in the non-editable text fields.

The next step is to select a certain code smell by using the radio buttons. SACSEA can only find one type of code smell per run. As explained in chapter 2, the application currently incorporates two code smell detection tools. Each code smell can only be found by one specific detection tool and therefore only one of them is activated per run, depending on which smell type is chosen by the user.

Finally, two revision numbers need to be entered. The number entered in the text field "*From revision:*" represents the number of the first revision that is checked out and in which code smells are detected in the next phase. Every subsequent revision is processed in the

same way, where the revision indicated by the number in the text field "*To Revision:*" is the final one. The detection process is started by the button [DETECT!] and is described in detail in the next subsection.

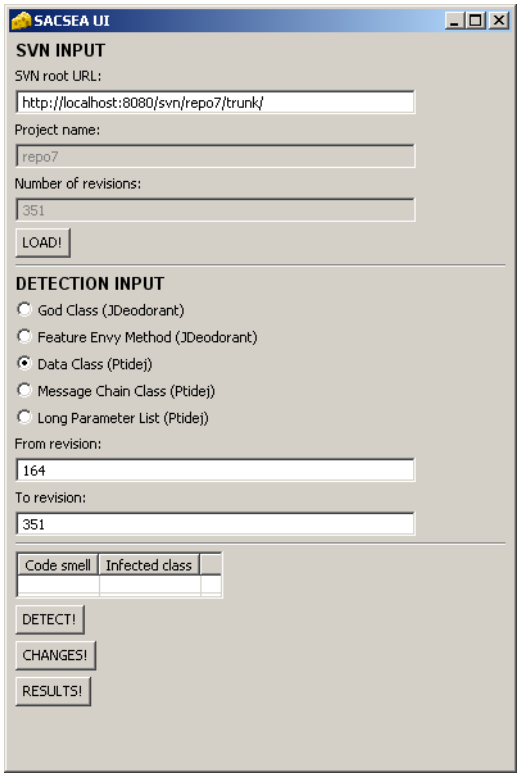


Figure 3.2: User Interface of SACSEA.

3.1.2 Detection

Once all the input values are entered in the UI and the button [DETECT!] has been pressed, SACSEA checks out the first revision specified by the user from the SVN repository as a working copy. This copy is then imported into Eclipse as a Java project using the *.classpath* and *.project* files inside the root directory, to which the entered URL points. If these files are not present, then the file *pom.xml* must be present. The project is then built by the utility *Maven* [4] using the command *mvn eclipse:eclipse*, which automatically generates the files *.classpath* and *.project*. Although importing a project using this tool takes slightly more time, the chance of having compile errors is smaller, since Maven resolves any missing external dependencies. If the root directory does not contain a *pom.xml* file or a combination of *.classpath* and *.project* files, then the Eclipse instance can not import this as a Java project and the working copy is immediately deleted from the local disk and not considered for code smell detection. SACSEA will continue with the next revision. However, if these

files are present, then the working copy is imported into Eclipse. Two cases can now be distinguished:

1. A code smell type was chosen that will be detected by the detection tool JDeodorant. In this case, an abstract syntax tree is internally created and parsed, which is then used to find code smell instances. If a Java source file (a file with the extension *.java*) from the SVN repository contains compile errors, then it is excluded from detection. Smells in the remaining compilable source files are temporarily stored in a custom array.

2. A code smell type was selected that will be detected by the detection tool Ptidej. In this case, the imported Java project must first be built. That is, *.class*-files have to be generated, since Ptidej builds an internal model based on these files. If the project can not be built, it is removed from the Eclipse instance and the process will continue with the next revision. Projects can be built when a *.classpath*-file and a *.project*-file are present. If the build was successful, then the code smell detection is started and any smell instances found are temporarily stored in a custom array.

After the detection on a single revision is complete, the results are stored in an XML-database. This is done using *XOM* (XML Object Model) [11], an open source API that provides Java functions for creating, editing and processing XML-files.

The XML-database has the following structure:

```
<root>
  <metadata>
    <projectname>FishHawk</projectname>
    <projecturl>https://fishhawk.svn.net/svnroot/trunk/</projecturl>
    <revision>100</revision>
    <smell>God Class</smell>
  </metadata>
  <occurrencesFound>
    <class0>fishhawk.draw.Canvas</class0>
    <class1>fishhawk.gui.MainGUI</class1>
    <class2>fishhawk.run.Generator</class2>
  </occurrencesFound>
</root>
```

Two major tags can be recognised as data holders within the <*root*>-tag: <*metadata*> and <*occurrencesFound*>. The first can be considered the header of the file, containing information regarding the project's name, the URL of its SVN repository, a certain revision and the examined code smell type. The latter represents the actual meat of the investigation and contains the code smell instances found in that particular revision of the system. The fictional example above shows three God Classes (*Canvas*, *MainGUI* and *Generator*) in revision 100 of project FishHawk.

Once the XML-file has been created, the current Java project is removed from the Eclipse instance and the detection process repeats itself by checking out the subsequent revision.

3.1.3 Difference Computation

When all revisions have been examined, there exists an XML-database for each of them, containing the code smells that were found. If no smell instances were found in a certain revision, then it will still have an XML-file. In this case, the <*occurrencesFound*>-tag has no contents. When the button [CHANGES!] is pressed, the XML-files are sorted in ascending order of revision number and every pair of consecutive files is compared for differences in code smell instances.

Basically, there are only two types of relevant differences:

- A code smell instance is present in the XML-file of revision n, but not in that of revision n+1. This means that it stopped being a code smell instance in revision n+1 under the influence of the changes of the corresponding commit.
- A code smell instance is present in the XML-file of revision n+1, but not in that of revision n. This means that it started being a code smell instance in revision n+1 under the influence of the changes of the corresponding commit.

Each of these differences found in the comparison is stored as one line in a CSV-file according to the following format:

```
100.xml;example.ExampleClass;101.xml
```

This says that the code smell instance *example.ExampleClass* was found in revision 100, but not in revision 101. This corresponds with the first difference type (removal of a smell from the infected instance) described above. In a similar way, the following line corresponds with the case where a smell is introduced to an instance in a certain revision (i.e. *example.ExampleClass* was found in revision 101, but not in revision 100):

```
101.xml;example.ExampleClass;100.xml
```

In conclusion, the CSV-file that results from this phase contains all the introductions and removals of code smells in classes and methods in certain revisions, which respectively correspond with the beginning and end revisions of the lifespan of an infected instance.

3.1.4 Output Generation

The output generation phase starts when the user presses the [RESULTS!]-button. Each smell instance is placed inside a hash map, using its name as the key and its lifespan as the corresponding value. This value is made up of ranges: intervals of revisions in which an entity is considered a code smell instance. For example, if in a particular system the class *example.ExampleClass* is a God Class from revision 30 to 180 and from revision 240 to 300, the key of the hash map is *example.ExampleClass* and the corresponding value is *<[30,180] , [240,300]>*.

This results in a hash map that contains each smell instance and its lifespan. A textual report is made of these data, along with metadata, such as the commit date and the developer

who made the commit. Finally, two Gantt charts are generated using the open source Java API *JFreeChart* [3]. They both show the lifespan of each smell instance, visualised by bars. The only difference between them is that one chart shows the lifespans in terms of revisions and the other in terms of dates on which the corresponding revisions were created. An example of the first type can be seen in figure 3.3.

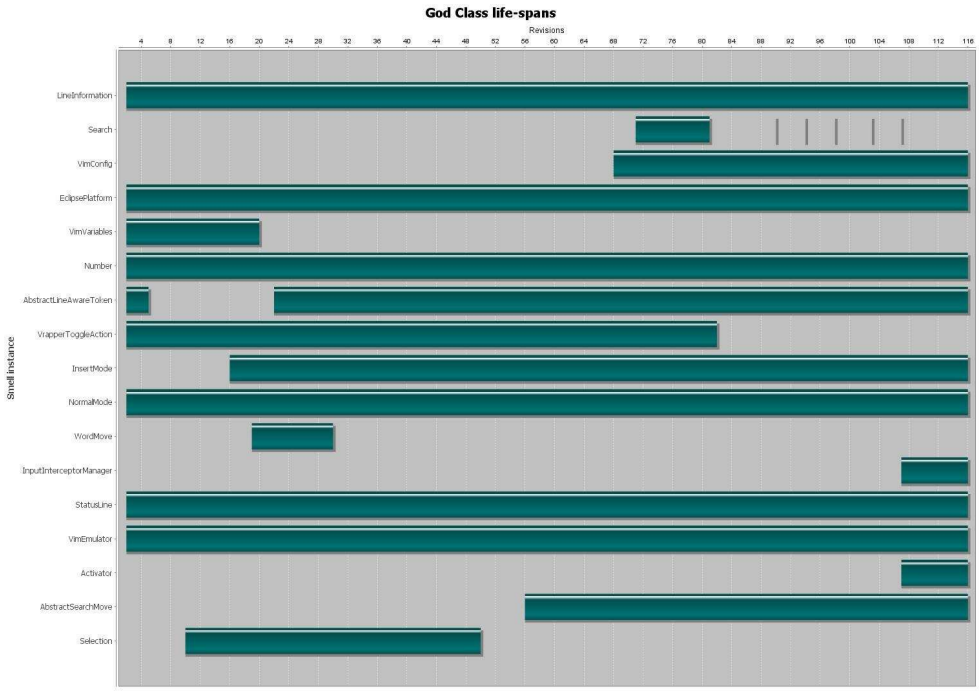


Figure 3.3: Example of a Gantt chart as output.

3.2 Design Decisions, Obstacles and Limitations

SACSEA has been developed in Java as an Eclipse plug-in, because the code smell detection modules need to operate on revisions that are imported as Java projects in an instance of Eclipse and also require Java source files to build an internal model on which the detection is performed. The detection modules were derived from existing tools. Writing a custom utility from scratch can be considered as a whole new project. As such, it is time-consuming and out of the scope of this graduation project. Also, SACSEA can only analyse SVN repositories. This was not a large limitation, since all of the included software projects were developed using SVN. The collection of software systems that were eventually not included in the case study contained a few projects, which were built with CVS. The *cvs2svn* [2] utility was used to convert them to a Subversion repository.

As mentioned in the previous chapter, SACSEA generates very data-intensive results and its operation takes up much time. Therefore, any form of optimisation that increases

its efficiency is desirable. One example is to use local SVN repositories rather than remote systems. This comes down to copying the entire repository from its hosting location, such as Sourceforge [6] or Tigris [9]. This can be done by using the utilities *rsync* or *svnsync*. In this case, the tool *VisualSVN Server* [10] was used to host several copied repositories on a local machine. In terms of contents and metadata, these repositories are exact replicas of their original remote counterparts. SACSEA can connect to these copies and function just as it would when the repositories are hosted remotely. The advantage of this approach is that any network latency is eliminated. This reduces the time to retrieve a working copy from a certain revision and its metadata. Furthermore, working with remote repositories requires a continuous Internet connection, which may be interrupted for various reasons. Given the long running time of SACSEA on systems with many or large revisions, this is an undesirable side effect, since the application does not check for a connection.

Whenever a revision has been examined for code smells, SACSEA continues with the subsequent revision. A new working copy is created by considering the previous working copy and performing updates on it rather than deleting the whole directory from the local disk and downloading the new revision in its entirety. This prevents unnecessary write actions and saves time, especially when projects include large library files.

SACSEA incorporates a code smell detection tool (JDeodorant) with an AST-based approach, which needs parsable Java source files. Naturally, there are software systems containing classes that can not be analysed by JDeodorant. If such a class is encountered, SACSEA ignores it and continues with the next class. This results in a partial detection, meaning that there may be code smell instances in certain revisions that are not shown in the actual output.

The previous design obstacle also applies to the other code smell detection tool (Ptidej), which needs *.class*-files in order to perform detection on the contents of a revision. The entire revision as a Java project must be able to build to generate these *.class*-files. If a project is unable to build (e.g. due to compile errors or a missing external dependency), then no *.class*-files are created and the entire revision is skipped for detection. Furthermore, Ptidej can only return classes as code smell instances, unlike JDeodorant that can also return methods suffering from code smells. The limited operation of both code smell detection tools is a threat to validity, which is described in detail in section 4.3.

SACSEA and its source code will be made publicly available by the faculty of EEMCS of the Delft University of Technology. It may only be used for academic purposes, due to the external code smell detection tools, which are incorporated in the source code. However, reusing and extending its functionality is allowed and proposed as future work in chapter 6.

Chapter 4

Case Study

This chapter presents the case study that was performed to examine the lifespans of five types of code smells. Seven software projects were selected according to several criteria and analysed using SACSEA. In section 4.1, the setup and approach of the experiments are described, including an introduction of the systems under investigation. Section 4.2 presents and discusses the results. Threats to the validity of this research are reported in section 4.3.

4.1 Experiment Setup

The ultimate objective of this graduation project is to answer research questions related to the evolution of code smells. As mentioned in chapter 1, the general approach to achieve this goal is to develop a software application, which is then used to mine empirical data from seven software projects. These data are analysed and statistics are derived from them to help answer the research questions. This section reports on the seven systems and the adopted approach.

4.1.1 Approach

The research questions must be dealt with through observations. In other words, clear-cut answers can not be given, since they are based on empirical data. These data must come from reliable sources. Finding these sources is not trivial, because they have to fulfil certain criteria and may still not be fully justified for use in the case study. Furthermore, the data retrieved in this graduation project are not exhaustive, as the use of more sources will undoubtedly result in answers that are more polished. However, it is not desirable to gather an excessive amount of sources in this graduation project, since the analysis by SACSEA takes a long time in terms of days depending on its input. Analysing more software systems is proposed as future work in section 6.3.

In this case study, the aforementioned data sources are in fact seven Java projects with a substantial development history, each stored in a version control system. These projects will be introduced in the next subsection. SACSEA determines the lifespans of any code smell instances found in each of these software systems. Some useful statistics are derived from these data, which are then used to help answer the research questions. Chapter 1 already

touches upon these statistics per research question, which are repeated here and described in detail.

RQ1 *Are some types of code smells refactored more and quicker than other smell types?*

The lifespan of one infected class or method in a system is not representative for the priority that is attached to resolving a certain code smell type. A better approach would be to take the lifespans of all detected code smell instances of one specific smell type in the software project and compute the average lifespan within the examined revisions. Next to this, the collection of individual lifespans per system is also used to determine the five-number summary for a clear view of the distribution of the data. These summaries are visualised as boxplots in Appendix D. Because the amount of investigated revisions differs per system, the average lifespan is also expressed as a percentage (i.e. the average lifespan expressed in number of revisions divided by the total number of examined revisions, multiplied by 100%). This calculation is repeated for all other subject systems, which results in a percentage for each of them. To form a fair indication of the average lifespan in all systems, the average of these percentages is calculated. Naturally, this average value is also expressed as a percentage and represents the average lifespan of one specific code smell type over all subject systems. These calculations are then repeated for all other code smell types investigated in the case study. It is not possible to compute the average amount of time that a code smell instance exists inside a system, because the time interval between any two commits varies greatly. For example, a system can have five commits on one day at an early stage of the development life cycle or only one commit per week at a later time.

RQ2 *Are code smells refactored more at an early or a later stage of a system's life cycle?*

Here, the same approach as mentioned for RQ1 applies. The only difference is that only the first 20% and the last 20% of the examined revisions of the software projects are considered. Thus, each system will have two percentages per smell type, representing the average lifespan of smell instances in the earliest and latest investigated revisions. The average of the percentages of all systems is calculated, resulting in two percentages per smell type: the average lifespan of the earliest revisions and the average lifespan of the latest revisions over all subject systems.

RQ3 *Do some developers refactor more code smells than others and to what extent?*

The goal is to count the number of times a code smell instance of a certain smell type is resolved. The name of the responsible developer is stored whenever he or she performs a corresponding activity. This will result in a list of developers and per smell type the number of instances they refactored. However, there are various reasons for removing a smell.

Intentional refactorings must be distinguished from removals that were the side effect of bug fixes or the renaming or deletion of entire classes or methods.

RQ4 *What refactoring rationales for code smells can be identified?*

To answer this question, the log messages of each commit responsible for the removal of a smell are examined. Similar to the approach for RQ3, deliberate refactoring activities must be identified. Ideally, the log messages are clear and representative for the actual changes. Naturally, this is not always the case and therefore insignificant refactoring rationales, like dead code elimination, are also included in the answer to this research question.

4.1.2 Subject Systems

Over ten software systems have been analysed using SACSEA. After assessing the usability of the output, the following seven projects were selected for the experiments:

- *CalDAV4j*: CalDAV4j is a protocol library that extends the WebDAV client library (which itself is an extension of the Apache HttpClient library) to allow high level manipulation of CalDAV calendar collections as well as lower level CalDAV protocol interactions.
- *Evolution Chamber*: This optimisation program helps players of the video game StarCraft 2 to find the fastest time and best ordering to create playable units with certain characteristics. It does so by applying a genetic algorithm.
- *JDiveLog*: JDiveLog is an open source diver's logbook for logging scuba dives. It manages all important dive data, as well as the pictures taken during the dive.
- *jGnash*: jGnash is a cross platform personal finance manager that supports users in tracking their finances. It is a double entry system with support for multiple currencies and can import Gnucash and QIF files.
- *Saros (Distributed Party Programming)*: Saros is an Eclipse plug-in for distributed collaborative text editing that can support arbitrarily many participants at once. All members of a session have an identical copy of an Eclipse project and Saros keeps these copies in sync as editing progresses.
- *VLCJ*: This project provides Java bindings to allow an instance of a native VideoLAN VLC media player to be embedded in a Java AWT Window or Swing JFrame.
- *Vrapper*: This Eclipse plug-in acts as a wrapper for Eclipse text editors and provides a Vim-like input scheme for editing text and moving it around. Two branches of this project are examined in this case study, which significantly differ in content. Before revision 116, there was a single *trunk branch*, which has been analysed from the beginning of the project until that revision. Then, a reorganisation in the structure of

the VCS was made, resulting in several branches. The *core branch* has been analysed from revision 121 until one of the latest revisions.

The selection of these projects was subject to multiple criteria. First, the systems need to be written in Java, because the detection modules of SACSEA can not process other programming languages. Moreover, they should allow free access to their repository on Subversion. A project also has to be in a mature development stage, meaning that it has to contain enough analysable revisions from which code smell lifespans of significant size can be determined. Diversity among the projects needed to be present to avoid bias. This was achieved by keeping the following aspects in mind during the selection process:

- *Nature*: Was the software system originally created as a commercial or as an open source project? In this study, only open source projects are considered. Evaluating industrial systems is an activity that is proposed as future work.
- *Domain*: For what problem or purpose was the system built and what is the context?
- *Strength*: Is the software system just a humble open source project for personal use or does it have the potential to be effectively used in industry?
- *Strictness of the development process*: The extent to which developers have to abide by development rules, such as guidelines for programming and committing. Usually, this is reflected by development manuals displayed on the software repository and by revisions without compile errors.
- *Number of analysable revisions*: A revision is considered analysable if at least one of the used code smell detection tools is able to extract a significant amount of code smell instances from the classes inside that revision.
- *Number of active developers*: Developers are considered active if they committed on a regular basis, regardless of their activity at any point in the development life cycle. Roughly, this means that a developer either participates occasionally throughout the entire range of investigated revisions or is responsible for a large part of the commits in a certain period of the life cycle.
- *Size*: The number of classes to be analysed per revision in the branch of interest. Because a project grows over time, the number of classes in the latest revision is meant here. Typically, the number of classes in the previous revisions is smaller.
- *Age*: When was the project first committed on Subversion and when was the last commit made? Note that a project can already be mature in its earliest revisions, since developers may not have used SVN when development started.

Table 4.1 provides an overview of these aspects. Per system, it shows the domain, the strength, the strictness of the development process, the total number of investigated revisions, the approximate number of active developers and the size, i.e. the number of analysable classes in the latest revision. Because the number of examined revisions differs

System	Domain	Strength	Development process	Revisions analysed	Active developers	Size (number of classes)	Investigation period
CalDAV4j	Protocol Library	Industrial	Strict	318	5	125	Oct 2007 - Mar 2011
Evolution Chamber	Genetic Algorithm	Open Source	Strict	282	11	481	Oct 2010 - Mar 2011
JDiveLog	Dive log manager	Open Source	Loose	872	13	331	Mar 2005 - Mar 2011
jGnash	Finance manager	Open Source	Loose	1493	1	466	Dec 2007 - Feb 2011
Saros	Distr. programming	Industrial	Strict	2482	26	821	Sep 2006 - Sep 2010
VLCJ	Java bindings	Industrial	Loose	1502	1	241	May 2009 - Apr 2011
Vrapper (Base)	Text editor wrapper	Open Source	Loose	115	2	119	Dec 2008 - Apr 2009
Vrapper (Core branch)				231	2	229	Apr 2009 - Apr 2010

Table 4.1: Overview of the systems under investigation.

per smell type for some projects, the value per subject system shown in the table is the largest number of revisions that were analysed. The investigation period of a system consists of the dates of the first and last revisions that were examined in the case study.

As can be seen in the table, there is some diversity between the subject systems with regard to strength, strictness of the development process, number of revisions, number of classes and investigation period. Naturally, all values given to these aspects are open to discussion and they may deviate in reality. However, it is unlikely that this greatly threatens the diversity.

4.2 Results

This section presents and discusses the results of the case study. For each research question, the corresponding statistics described in the previous section are displayed for each subject system per code smell type. Due to the functional limitations of the detection utility Ptidej, instances of the smell Long Parameter List are expressed as classes and not as methods.

In other words, infected instances are classes that contain at least one method using more than three parameters. Furthermore, some smell types have a different number of analysed revisions than other types in the same subject system, due to the differences between the detection tools and their limited capabilities at times. Therefore, average lifespans have been expressed in number of revisions as well as percentage. More detailed results of all the experiments can be found in Appendix B, C and D.

For convenience, the different code smell types have been abbreviated in the tables as follows:

- God Class - *GC*
- Feature Envy - *FEM*
- Data Class - *DC*
- Message Chain Class - *MCC*
- Long Parameter List Class - *LPLC*

4.2.1 RQ1

This research question was defined as: "*Are some types of code smells refactored more and quicker than other smell types?*" and is answered by comparing the average lifespans of different code smell types found in the subject systems. Per smell type, table 4.2 shows the average lifespan: the sum of the lifespans of all code smell instances detected in each system divided by the total number of these instances. All values are expressed in number of revisions and have been rounded to the nearest integer. Because the analysis of some smell types included a different number of revisions than other types in the same subject system, table 4.3 presents this average lifespan expressed as a percentage. This percentage is the average lifespan in terms of revisions divided by the total number of analysed revisions per smell type, multiplied by 100%. This table also displays per smell type the average percentage over all systems and the corresponding standard deviation. Appendix B contains the total number of analysed revisions for each system per smell type.

	GC	**FEM**	**DC**	**MCC**	**LPLC**
CalDAV4j	135	71	173	167	212
Evolution Chamber	145	85	78	84	130
JDiveLog	419	320	447	313	372
jGnash	883	792	810	669	1025
Saros	680	566	602	643	851
VLCJ	533	421	1007	474	541
Vrapper (Base)	70	55	84	84	62
Vrapper (Core branch)	113	160	153	132	110

Table 4.2: Average lifespans in terms of revisions.

	GC	FEM	DC	MCC	LPLC
CalDAV4j	42.38%	22.39%	54.55%	52.52%	66.52%
Evolution Chamber	51.40%	30.27%	27.52%	29.68%	46.08%
JDiveLog	54.37%	36.71%	51.28%	35.91%	42.67%
jGnash	67.81%	60.83%	81.97%	70.78%	68.66%
Saros	32.56%	24.31%	24.24%	26%	34.29%
VLCJ	35.46%	29.11%	67.03%	31.58%	35.98%
Vrapper (Base)	60.82%	47.58%	72.92%	72.73%	53.48%
Vrapper (Core branch)	48.79%	71.90%	66.02%	57.14%	47.62%
Total average	*49.20%*	*40.39%*	*55.69%*	*47.04%*	*49.41%*
Standard deviation	*12.10%*	*18.08%*	*20.79%*	*18.76%*	*12.81%*

Table 4.3: Average lifespans in percentage.

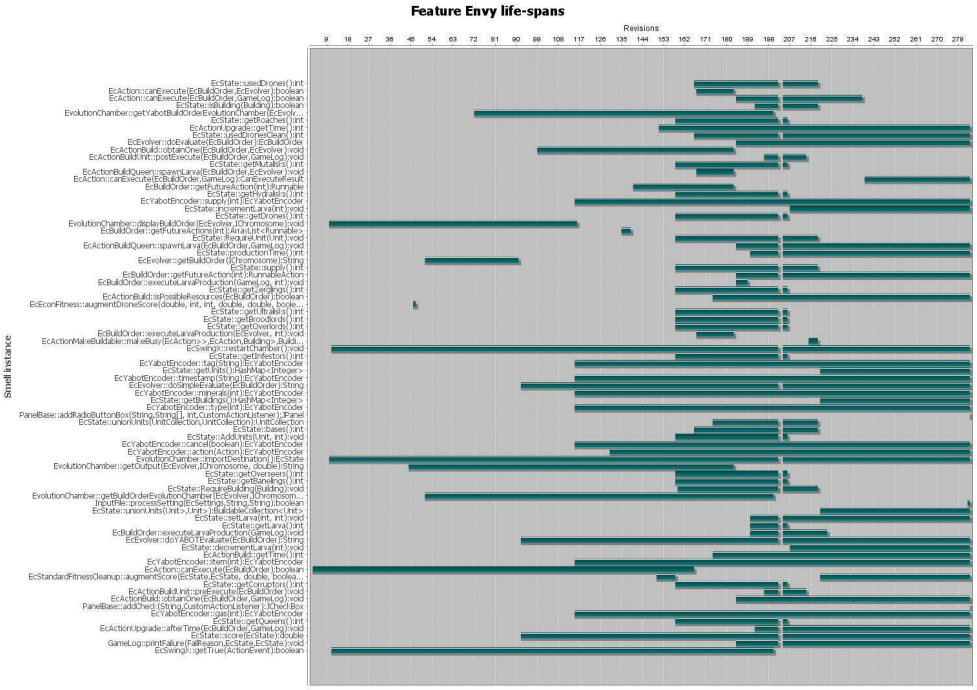


Figure 4.1: Lifespans of Feature Envy Methods in Evolution Chamber.

As can be seen in table 4.3, the differences between the average lifespans over all subject systems are not large. Nevertheless, something can still be said about the results. The Feature Envy Method smell instances have the shortest lifespan on average. The highest lifespan can be found in the core branch of Vrapper, however it has only three Feature Envy Methods. Looking at the Gantt charts per system, most infected methods start suffering from this smell after several revisions. Also, the majority of the instances are removed

relatively quickly. An example of this phenomenon can be seen in figure 4.1. Next to this, the boxplot in figure 4.2 is one example of a system containing individual lifespans being concentrated in the lower range of revisions, meaning that most infected instances have a short existence. The cause of this dynamic behaviour might be that a method can easily be refactored or removed, either intentionally or coincidentally as a side effect of a maintenance activity. This is plausible as the commits responsible for removals include contents and log messages that contain changes to methods and classes with the intent to implement new functionality.

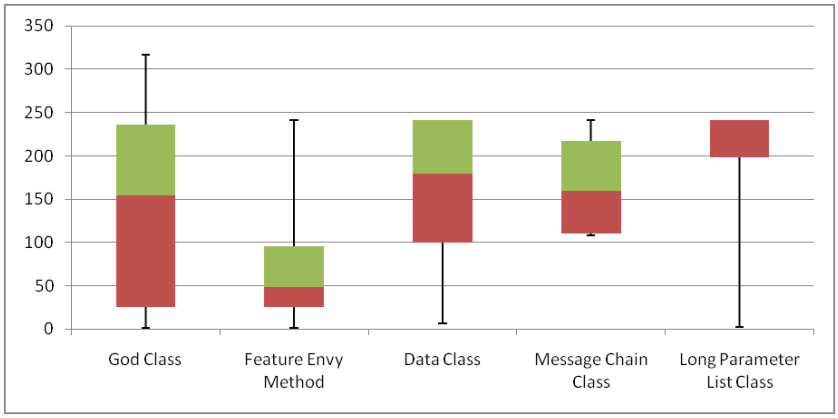


Figure 4.2: Boxplot of lifespan distribution in CalDAV4j.

The God Class and Long Parameter List Class are almost identical with regard to the total average and standard deviation. Compared to Feature Envy Methods, the refactoring behaviour of these two code smells is more static. In other words, once a smell instance of these two smell types is introduced, it has a lower chance of being removed quickly. This could be an indication of the notorious difficulty of refactoring a God Class [31] and the carelessness of developers of having many parameters in their methods. Especially Long Parameter Lists are coincidentally removed rather than deliberately.

The analysis on Data Classes shows that this type has the largest total average, but the average lifespans of most systems also deviate greatly from that value. According to the Gantt charts, a clear trend can be seen: the number of Data Classes increases over time and refactoring takes place seldom. The classes that do not get refactored are typically created to be dedicated data holders or small libraries. The instances that do get removed are scarce and have a very short lifespan. Usually, these classes are renamed and live on as other instances or, more importantly, are created with the intent to add functionality a few revisions later.

Message Chain Class is the only smell type of which relatively few instances have been found in many projects. Also, there is no clear pattern of smell introduction and removal. The Gantt charts of most subject systems show that many infected classes have a lifespan of approximately 50% of the investigated revisions. Most of the time, these instances are either removed at random points in the life cycle or introduced in the latest revisions and

not refactored at all. Figure 4.3 shows an example of this discovery. This suggests that there is little deliberate refactoring activity and that any introductions or removals are the consequence of other development activities.

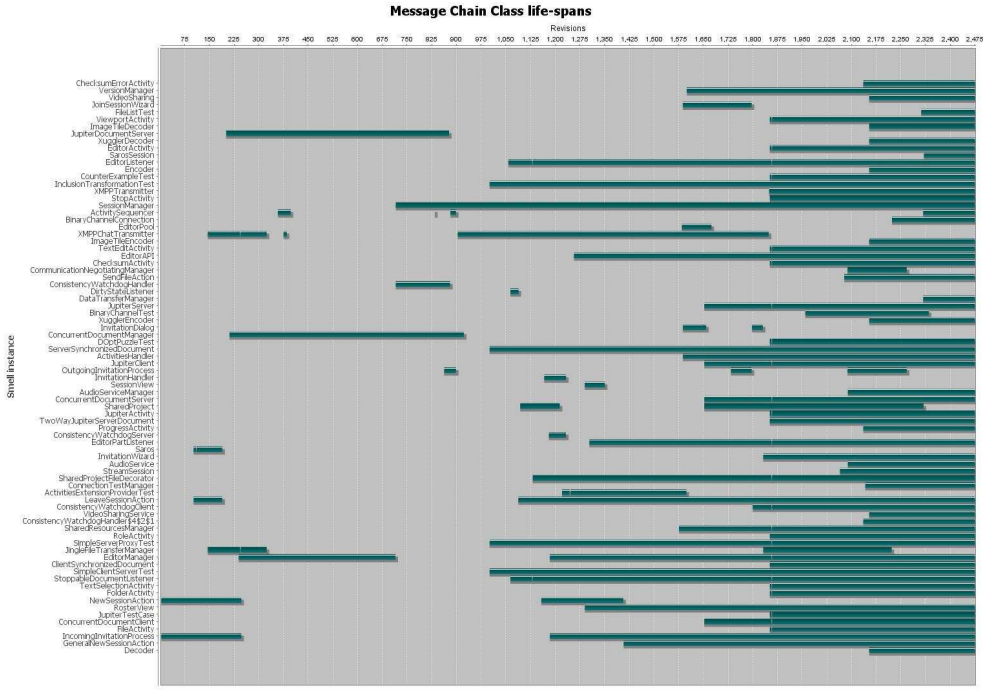


Figure 4.3: Lifespans of Message Chain Classes in Saros.

Concluding remarks

In most analysed software projects, code smell instances have an average lifespan of about 50% of the examined revisions. Feature Envy Methods seem to be resolved the most, due to the fact that they are more susceptible to accidental or deliberate refactoring activities. Also, God Classes are known in literature for being hard to refactor [31], which may be the reason why they live longer in the subject systems. Given the increasing introduction and scarce removal of infected instances over time, Data Classes and Long Parameter Lists are not regarded as a liability to the overall reliability of a software system by many developers.

The results imply that software developers do not prioritise the removal of code smells, even if they are aware of the risks. Some smell types are resolved quicker than other types, but the question remains if these refactorings are always intentional or the consequence of other maintenance activities. The answers to the following research questions will provide better insight into this matter.

4.2.2 RQ2

The definition of this research question was: *"Are code smells refactored more at an early or a later stage of a system's life cycle?*" and is answered by determining the average lifespan in the first 20% and the last 20% of the examined revisions of all subject systems. Similar to table 4.2, the average lifespans of smell instances within the youngest versions are shown in terms of revisions and in percentage in table 4.4 and 4.5, respectively. The same applies to the data from the oldest revisions, which are presented in table 4.6 and 4.7. Again, the total average is computed, along with the corresponding standard deviation. Note that these average values are based on the lifespans of code smell instances that may exist outside the subsets of 20% of the revisions. For example, if 500 versions are examined and a smell instance exists in the first 200 versions, then its lifespan within the first 20% of the analysed revisions is considered to be 100 revisions.

	GC	**FEM**	**DC**	**MCC**	**LPLC**
CalDAV4j	52	0	0	0	0
Evolution Chamber	49	32	1	25	26
JDiveLog	102	94	73	73	99
jGnash	134	138	72	60	173
Saros	226	208	246	192	298
VLCJ	107	115	194	173	0
Vrapper (Base)	18	14	20	19	12
Vrapper (Core branch)	27	0	40	38	47

Table 4.4: Average lifespans within the youngest 20% in terms of revisions.

	GC	**FEM**	**DC**	**MCC**	**LPLC**
CalDAV4j	81.25%	0%	0%	0%	0%
Evolution Chamber	85.96%	56.76%	2.19%	49.86%	46.20%
JDiveLog	66.01%	53.43%	41.45%	41.71%	56.33%
jGnash	51.28%	52.77%	36.36%	31.60%	57.86%
Saros	54.04%	44.62%	49.42%	38.75%	59.94%
VLCJ	35.62%	39.71%	64.37%	57.48%	0%
Vrapper (Base)	78.62%	58.70%	86.34%	84.06%	52.17%
Vrapper (Core branch)	56.74%	0%	84.04%	81.21%	100%
Total average	*63.69%*	*38.25%*	*45.52%*	*47.33%*	*46.56%*
Standard deviation	*17.39%*	*24.41%*	*32.89%*	*27.27%*	*33.01%*

Table 4.5: Average lifespans within the youngest 20% in percentage.

According to table 4.4, CalDAV4j seems to have no smell instances for almost all smell types in its early revisions. This is due to the fact that those revisions were unexpectedly not analysable. Nevertheless, the overall results show a clear pattern for all smell types. The younger revisions have a significantly lower total average than the latest versions. As

can be seen in the Gantt charts, the main cause is that the number of long-living code smell instances increases over time. Figure 4.3 happens to show an example of that growth. Naturally, as a system expands, it will contain more classes and methods that may or may not get infected. The results hint towards the lack of concern or awareness of the developers regarding code smells. A more thorough investigation is needed in order to strengthen the validity of this presumption.

	GC	FEM	DC	MCC	LPLC
CalDAV4j	56	45	64	64	64
Evolution Chamber	54	51	55	57	52
JDiveLog	132	151	165	154	166
jGnash	250	249	197	176	280
Saros	274	343	401	417	414
VLCJ	230	205	301	203	301
Vrapper (Base)	20	23	21	23	23
Vrapper (Core branch)	35	45	46	46	39

Table 4.6: Average lifespans within the latest 20% in terms of revisions.

	GC	FEM	DC	MCC	LPLC
CalDAV4j	86.96%	69.85%	100%	100%	100%
Evolution Chamber	94.32%	89.38%	96.07%	100%	91.81%
JDiveLog	85.43%	86.31%	94.49%	88.22%	94.71%
jGnash	95.72%	95.51%	99.34%	93.37%	93.81%
Saros	65.57%	73.59%	80.71%	84.18%	83.20%
VLCJ	76.26%	71.08%	100%	67.44%	100%
Vrapper (Base)	84.95%	100%	89.57%	100%	100%
Vrapper (Core branch)	74.47%	100%	98.45%	97.66%	82.55%
Total average	*82.96%*	*85.72%*	*94.83%*	*91.36%*	*93.26%*
Standard deviation	*10.26%*	*12.71%*	*6.71%*	*11.35%*	*7.14%*

Table 4.7: Average lifespans within the latest 20% in percentage.

The standard deviation for the early revisions is higher than for the latest revisions. This is mainly due to the fact that the youngest examined revisions of some systems are already at a mature point in the development life cycle and thus initially contain more code smells (and removals) than revisions that actually mark the birth of a project. The latter type will usually contain very few code smell instances that are not refactored quickly, causing the average lifespan to be high for systems with relatively little examined revisions like Evolution Chamber and Vrapper.

Concluding remarks

For all subject systems and smell types, the first 20% of the examined revisions reveal

a substantially lower average lifespan than the last 20%. The main cause is the introduction of more code smell instances over time. Every software project grows as time progresses. However, an increasing number of smell instances may be an indication of little refactoring activity throughout the development life cycle.

The results of the case study show that most infected instances at the beginning of a system's life cycle are bound to be refactored within a few revisions. However, the number of long-living infected instances increases over time. Compared to the earlier revisions of a system, the latest versions show significantly less refactoring activities. This implies that the willingness to resolve code smells decreases over time.

4.2.3 RQ3

The question was posed as: "*Do some developers refactor more code smells than others and to what extent?*" The approach here is to count the number of refactored code smell instances per developer. Table 4.8 shows the names of the developers participating in the subject systems with the number of all infected instances that were removed at some point, according to SACSEA. These values also include erroneous removals, due to the occasional faulty behaviour of the integrated smell detection tools. Note that one commit can contain multiple removed smell instances, which can be the consequence of moving functionality across classes and methods.

However, not all removals are the result of dedicated refactoring activities. The number of resolved instances in table 4.8 must be reduced by the number of times a coincidental refactoring occurred. For this, the log messages of all commits responsible for a smell removal have been manually examined and categorised based on the cause of the removal. Signal words like "*Refactored*", "*Extracted class*" and "*Clean-up*" are usually indications of true refactoring activities. Naturally, these words are no guarantee and a developer's perspective on the manifestation of code smells may differ from the ones assumed by the detection tools used in this graduation project. Table 4.9 shows the names of the developers with the approximate amount of resolved instances that were the consequence of deliberate refactoring per code smell. Also, the last column in both tables shows the total number of relevant commits per developer within the range of analysed revisions.

There seems to be very little intentional refactoring activity in all subject systems. No useful refactoring activities were found in JDiveLog. According to the commit logs, the values shown in table 4.8 mainly consist of smell removals that were the side effect of maintenance work, such as moving and renaming infected instances, but also of bug fixes and the implementation of new functionality. Some removals were part of a larger, unrelated refactoring and are therefore not included in table 4.9. The log messages also reveal that most changes are committed per task. Exceptions include corrections to previous commits and small changes.

Table 4.9 does show that some developers refactor more than others. This varies from systems in which only one engineer resolves code smells (e.g. CalDAV4j) to projects in which more developers refactor and only one does so more frequently (e.g. Evolution Chamber). Moreover, the systems jGnash and VLCJ that have only one software engineer provide an indication on how concerned that developer is with code smells.

	Developer	GC	FEM	DC	MCC	LPLC	Commits
CalDAV4j	bobbyrullo	7	1	1	1		*78*
	robipolli	18	17	2	1	1	*111*
Evolution Chamber	nafets.st	8	61		6		*15*
	bdurrer	2	8	2	1		*15*
	domagala.lukas	1	4	1	52	4	*36*
	mike.angstadt		3				*41*
	fritley		3	3	1		*70*
	brendan.speer		1	1		1	*4*
	netprobe		1				*16*
JDiveLog	onlinervolker	13	49		2		*93*
	andre_schenk	5	8	4		1	*107*
	vkorecky	12	37		3		*48*
	sjomik	1					*2*
	szdavid1	3	1	1			*52*
	pellmont	46	53	3	3	1	*456*
	Levtraru		2				*3*
jGnash	ccavanaugh	196	246	2	26	17	*1355*
Saros	sotitas	6	4	1	1	1	*7*
	chris_fu	13	15	3	4	7	*141*
	coezbek	108	128	4	19	13	*784*
	Arbosh	1					*6*
	k_beecher	2	7		1	2	*31*
	wojtus	1	13			6	*1*
	hstaib	1					*9*
	marrin	33	58	3	25	4	*130*
	orieger	19	20	2	7	3	*119*
	ahaferburg		10	55		1	*69*
	s-ziller	8	5				*63*
	starkmann	1					*3*
	testvogel	1					*4*
	szuecs	5	2		1	1	*14*
	ldohrmann	7	2				*15*
	djemili	1	1				*43*
	marcus-fu		1				*3*
	ornis			2		1	*27*
VLCJ	wm.mark.lee	39	15	13	4	2	*878*
Vrapper (Base)	weissi	2					*6*
	waweee	9	2	3	5	1	*102*
Vrapper (Core)	kgoj	6		4	1	1	*75*
	waweee	15		2	7		*81*

Table 4.8: Number of code smell removals, as found by SACSEA.

	Developer	GC	FEM	DC	MCC	LPLC	Commits
CalDAV4j	bobbyrullo						*78*
	robipolli	2			1		*111*
Evolution Chamber	nafets.st	1					*15*
	bdurrer	2	7	2			*15*
	domagala.lukas					1	*36*
	mike.angstadt						*41*
	fritley		1				*70*
	brendan.speer						*4*
	netprobe						*16*
JDiveLog	onlinervolker						*93*
	andre_schenk						*107*
	vkorecky						*48*
	sjomik						*2*
	szdavid1						*52*
	pellmont						*456*
	Levtraru						*3*
jGnash	ccavanaugh		1				*1355*
Saros	sotitas						*7*
	chris_fu	1					*141*
	coezbek	3	1			1	*784*
	Arbosh						*6*
	k_beecher		7			1	*31*
	wojtus						*1*
	hstaib						*9*
	marrin	1					*130*
	orieger						*119*
	ahaferburg						*69*
	s-ziller		1				*63*
	starkmann						*3*
	testvogel						*4*
	szuecs						*14*
	ldohrmann						*15*
	djemili						*43*
	marcus-fu						*3*
	ornis						*27*
VLCJ	wm.mark.lee	3	3		1		*878*
Vrapper (Base)	weissi						*6*
	waweee	1			2		*102*
Vrapper (Core)	kgoj			1			*75*
	waweee						*81*

Table 4.9: Number of code smell removals, caused by intentional refactorings.

Concluding remarks

Table 4.9 reveals that there is sporadic refactoring activity, compared to the numerous amount of smell instances found in each subject system. Still, at most two developers per subject system refactor more smell instances than the others. Some of them are the only ones within the project responsible for an occasional removal. Others refactor a few more infected instances than other developers. As mentioned before, the majority of the code smell instances found in the experiments were resolved as a consequence of other maintenance activities or the implementation of new functionality. Most developers do not seem to recognise the added value of refactoring in a system with high commit activity, implying low awareness or concern regarding code smells.

4.2.4 RQ4

This research question was stated as follows: "*What refactoring rationales for code smells can be identified?*" Chapter 1 mentioned examples of refactoring rationales, such as the introduction of a dedicated refactoring phase and the accommodation of new functionality. The logs and contents of the corresponding commit and its neighbouring commits must be examined to find some evidence of such motives. However, due to the scarcity of intentional refactoring activities in the subject systems, finding rationales will be difficult. Therefore, code styling rationales are also considered here, such as dead code elimination. The most common rationales for resolving the smells considered in this case study are listed below. These rationales have been derived from the log messages and source code inspection.

- *Cleaning up dead or obsolete code*: Many subject systems contain a few revisions in which duplicate, unused or old classes and methods are removed. Occasionally, this results in the removal of a smell instance, albeit accidental. Some of these activities may not be considered as true refactorings, but the underlying motives are usually not subject to a larger refactoring. In other words, sometimes developers just see and grab an opportunity to clean up.

- *Dedicated refactoring*: Similar to dead code elimination, there are some cases in which developers refactor for the sole purpose of refactoring. This often comes down to restructuring libraries (Data Classes) or generalising large classes through the use of interfaces. The question arises whether the developers are aware of the specific code smell that infects a certain software entity. The answer to this question remains unknown, however it is reasonable to assume that their "programming instinct" tells them when a class is growing out of proportion, for instance.

- *Maintenance activities*: The majority of the refactorings are coincidental, as a side effect of intentional bug fixes or implementing new functionality. This causes many classes and methods to be removed for that purpose, including the infected instances. Whether these activities also integrate an implicit refactoring activity is uncertain, though some smell instances are indeed resolved by the changes. Next to this, many software entities are renamed in these activities. SACSEA sees this phenomenon as

the removal of one infected instance and the immediate introduction of another, while they are in fact one and the same instance. This is not considered a refactoring in this study.

Although the list above does not include very exciting motives, there are some commits in which more explicit refactorings have been performed. The corresponding log messages mention rationales like the accommodation of new functionality, performance enhancement and readability improvement. However, these commits do not contain refactorings that actually resolve code smell instances found by SACSEA. This might imply a plausible flaw in the detection modules, the different perspectives of developers on code smells or the fact that the refactored instance was not really infected in the first place, in which case it was only resolved with the original goal in mind.

Concluding remarks

Within the limits of the case study, it seems that the act of refactoring has always been done on a small scale and not for a higher purpose. However, there are some cases in which this higher purpose is present, but fall outside the scope of the case study, since these refactorings did not cause the removal of a code smell as detected by SACSEA. The rationales briefly mentioned in chapter 1 have not been found in the case study. However, there were various other rationales to be recognised, such as code clean-up and deliberately resolving code smells. Motives with regard to performance and readability were found in commits that were not marked by SACSEA as responsible for smell removals and therefore fall outside the scope of the case study. This hints towards a definite awareness of code smells among developers, although infected instances only seem to be removed when the time is convenient, which explains why there is little refactoring activity in the subject systems. The intent to refactor is there, when there is time and effort to spare.

4.3 Threats to Validity

Much effort was spent to perform the research in this graduation project in a structured and correct manner. However, the case study was subject to certain conditions, which could influence the eventual outcome. This section describes the aspects that may threaten the validity of this study.

4.3.1 Internal Validity

Threats to internal validity concern aspects that can influence the observations. First, no causal relationship among the investigated variables (e.g. code smell types, refactoring rationales and number of active developers) is claimed in this graduation project. The results were discussed, while taking the characteristics of all subject systems and developers into account and trying to come up with interpretations to the findings.

The code smell detection tools used for the case study were originally designed for a limited academic purpose. This means that the developers had a certain goal in mind during

development and as such, they built their applications toward that goal. The disadvantage of that approach is that the tools may not extend well to other goals, like the ones set in this graduation project. This limitation is reflected by strange and unpredictable behaviour, that prevents the emission of flawless results. For example, the applications may identify code smell instances, which are not considered as such by a human expert. As a result, code smells may not be subject to a refactoring activity for many revisions. False negatives may also occur for similar reasons, in which case the code smell instance is not shown in the results at all. By using two tools with different detection approaches and investigating subject systems that have relatively little unparsable revisions, this threat is slightly reduced. In order to nullify it, this research has to be replicated using one or more detection tools that are considered (functionally and commercially) mature and allow the user to enter custom code smell definitions. However, such definitions will always be subject to discussion.

Two code smell detection tools were chosen for this graduation project to avoid potential bias in detection, since both of them have a different approach in detecting code smells. Both tools are responsible for a set of smells to detect, which is disjoint from the set of the other. The question arises whether and to what extent different smell instances were to be found if one tool were to be responsible for the set of the other and vice versa. This is part of a larger question: How would the results be influenced if other detection approaches were used for the same code smells? A study that addresses this issue is proposed as future work in section 6.3.

SACSEA was thoroughly tested, but it remains an academic prototype. The toolchain may contain unforeseen defects. It must be mentioned that the integrated code smell detection tools also form a liability, regardless of their application in other research. For instance, JDeodorant uses an AST-based approach, which needs parsable Java source files. There are some revisions in the subject systems, in which source code with missing dependencies and compile errors has been committed. Depending on the responsible detection utility, the source file or the entire revision is skipped for detection. Consequently, there is no useful data available for some revisions of four systems. The percentage of unparsable revisions varies from 1% to 25%. This affects the results in two ways. Either the lifespan of some long-living code smell instances in unparsable revisions is not shown in the visual reports, which causes the average lifespan in a system to be lower than it actually is. On the other hand, some instances that only exist in the range of those revisions may not be shown at all, causing the average lifespan in a system to be higher than it actually is. This threat also affects the information about refactoring activities. This risk has been kept to a minimum by analysing a selection of revisions of several software projects and selecting those systems with the least data loss.

Selection criteria were devised to ensure diversity among the subject systems and avoid bias. However, it is highly likely that these requirements may not be fully justified or there are other aspects that were not taken into consideration. Also, the initial development steps of some systems may not be present in a VCS, because developers probably start using a repository when the project is mature enough. This obstacle has been countered by including some software repositories in the case study that do contain this information, taking the primary selection criteria into account.

Renaming classes and methods is a common practice in software development, but it is

considered a nuisance in this study. SACSEA is unable to determine if an entity in revision n has been renamed in revision n+1. This will usually be shown in the output as a smell instance ceasing to exist at one revision and another smell instance being introduced in the subsequent revision. This phenomenon had to be recognised manually and not mistaken for a refactoring.

4.3.2 External Validity

External validity threats deal with the generalisation of the results. The most obvious threats are in this case the number of subject systems and their scope. A great amount of effort has been spent on achieving diversity among the subject systems. Still, the case study was performed with seven open source projects written in Java. Therefore, it is possible that the results will not fully hold for other similar projects, industrial systems or applications developed in other programming languages or paradigms. The investigation of this issue is proposed as future work.

4.3.3 Construct Validity

Threats to construct validity concern the relation between theory and observation. A serious threat lies in the identification of refactorings, which is based on the commit logs of the VCS. Indeed, the commit log of the revision in which a smell disappears can be retrieved. However, they may not accurately reflect the commits related to a smell removal, because developers show different behaviour for committing their changes, e.g. periodically or task-based. Also, deliberate refactorings must be distinguished from other coding activities that coincidentally result in the removal of a code smell. Log messages have to be inspected manually to make this distinction, which is usually clear, taking the aforementioned threat into account.

The subjective nature of identifying code smells is also a threat to validity. This activity is captured in the detection tools, which make different assumptions than other developers. This risk is minimised by choosing detection utilities, which base their smell identification on definitions from literature.

Chapter 5

Related Work

Software maintenance plays a significant role in the development process. Several studies, among them the work of Beck and Fowler [22], show that code smells and anti-patterns have a negative influence on software quality. If no action is taken in a timely manner, then a software system will deteriorate over time. There are numerous examples available of the study and development of code smell detection techniques [31] [41]. However, there are also contributions to the investigation of the evolution of code smells and anti-patterns [33] [39]. This chapter presents some examples of these contributions, which are closely related to this graduation project.

5.1 CodeVizard

Zazworka and Ackerman developed a framework called *CodeVizard* [1] [45], which can mine data from source code repositories at source file level and hence reveal the evolution of those systems. The tool focuses on areas of risk, such as increasing software complexity, degrading architectures, process violations and also code smells. CodeVizard also offers various visualisations for examining the infected entities and their change history.

CodeVizard's main workflow starts with reading and converting data from CVS or SVN repositories and storing them into a relational database system for rapid access. Then it facilitates processing these data (e.g. computing more than 70 software metrics for Java and C# code). It also provides a set of interconnected views to analyse the processed data:

- The *System View* visualises repository contents over time and allows the inspection of software metrics. The visualisation is similar to the graphical output of SACSEA: the lifespans of code smell instances are represented by coloured bars. This view also shows small coloured change bars that represent metadata (e.g. when components were modified and by whom). SACSEA only provides this information as text.
- The *Code View* gives insight into the development and evolution of a single file by displaying its source code over a certain time period.
- The *Metric View* presents various software metrics, such as lines of code. This is also a feature that SACSEA lacks, since it was not necessary to include this information

in order to answer the research questions of this graduation project. However, the implementation of such functionality is proposed as future work.

CodeVizard has been used to support several empirical studies with different research goals, including the evolution of code smells. One of these contributions was made by Olbrich et al. [38], who were interested in the number of code smells that change over time and in the effect of code smells on component development in terms of frequency and size of changes. They performed a case study in the same manner as the experiments carried out in this graduation project. They selected two Java systems, Lucene and Xerces, which had to meet multiple criteria. CodeVizard was applied on these applications and returned information on the evolution of the code smells God Class and Shotgun Surgery. The results showed that entities infected with these code smells have a higher change frequency. Such classes seem to need more maintenance than non-infected classes.

5.2 The Impact of Smells and Anti-patterns on Software Change-proneness

Similar to Olbrich et al. [38], Khomh et al. performed a study [28] regarding the same research question: *"Are classes with code smells more change-prone than classes without smells?"* For this, they used their own solution DECOR [36] to specify and find code smells in the systems Azureus and Eclipse. The analysis was based on releases rather than revisions of the VCS. 29 code smell detection algorithms, provided by DECOR, were applied on a few releases to obtain the sets of infected classes. The results of the study provided empirical evidence that classes with code smells are more subject to change than others in almost all considered releases of both systems and that specific smells are more correlated than others to change-proneness.

A study with the same approach was done by Khomh et al. [29], but anti-patterns were the target of interest in this case. Using DECOR, thirteen anti-patterns in several releases of the systems ArgoUML, Eclipse, Mylyn and Rhino were detected in order to investigate the correlation with change- and fault-proneness. The authors showed that in almost all releases of each system, classes participating in anti-patterns are more change- and fault-prone than other classes and that certain kinds of anti-patterns have a higher impact than other types. Moreover, class size alone can not explain the higher change-proneness of infected instances. Finally, structural changes seem to affect more classes with anti-patterns than other classes. Qualitative explanations of the increase of change- and fault-proneness in infected classes were given using release notes and bug reports.

5.3 The Evolution of Smells in Object-Oriented Programs

Chatzigeorgiou and Manakos [17] explore the presence and evolution of three code smells by analysing past releases of two open-source systems. In contrast to other studies that mainly focused on the identification of refactorings, this research focuses on findings and assumptions regarding the problems themselves and the rationales behind their introduction

and removal during software evolution. The authors attempt to gain insight into the number of design problems over time, whether the evolution of a software system removes some of its code smells or only after targeted maintenance activities, the time of smell introduction in a system and the frequency and urgency of refactoring activities. The code smell detection tool JDeodorant is employed for the identification of code smells, which is also used in this graduation project. They state that the tool offers the possibility to detect non-trivial smells, which require a systematic and elaborate refactoring activity.

The results showed that the design problems persist up to the latest examined version in most cases, accumulating as the project matures. Moreover, a significant percentage of the code smells were introduced at the time when the method in which they reside was added to the system. Only a few smells were removed from the project and in the vast majority of these cases their disappearance was not the result of targeted refactoring activities but rather a side effect of adaptive maintenance. Even though the case study was performed under slightly different conditions, its outcome is very similar to the results found in this graduation project and therefore supports the assumption that code smells increase over time.

5.4 The Evolution of Smells in Aspect-Oriented Programs

Aspect-oriented programming (AOP) is a programming paradigm, which aims to increase modularity through the separation of cross-cutting concerns. Some code smells specific for AOP have already been introduced in literature [42]. Macia et al. [14] performed an exploratory study of code smells in evolving aspect-oriented systems. They investigated if and how code smells evolve in such software projects and defined new smells as a side effect. Due to the lack of automated detection techniques for finding smells in programs developed using this paradigm, the detection was done manually. The investigation focused on 18 releases of three aspect-oriented systems from different domains. The outcome of the study suggested that previously-documented smells might not occur as often as claimed. The analysis also revealed that newly-discovered code smells might occur more often than well-known ones and that the unknown smells seemed to be consistently associated with non-trivial refactorings and corrective changes.

5.5 Measuring Refactoring Effort through MSR

Refactoring of code smell and anti-patterns is supposed to improve the structure of existing source code in the long run in order to increase the changeability and maintainability of a software system. To analyse the impact of refactoring on software maintenance, Moser et al. [37] tried to find out how much refactoring software engineers do. In some cases, this information is directly available from the log messages in a VCS. A model is proposed on how to mine software repositories in order to obtain quantitative information on refactoring effort throughout the evolution of a software system. Next to this, the authors developed a prototype that implements the model and validated their solution by applying the tool to one close-to industrial software project and one open source project. Judging from the results,

the authors were able to distinguish refactoring effort from other forms of maintenance activities fairly well in most cases. This information is valuable for identifying the amount of refactoring done during maintenance, the developers who refactor and those who do not, the parts of a system that are not refactored and the impact of refactoring.

Chapter 6

Conclusions and Future Work

The first section of this chapter contains conclusions based on the results of the case study and presents answers to the research questions defined in chapter 1. Furthermore, the work carried out in this graduation project contributes to the discipline of software evolution, particularly on the subject of code smells. A list of these contributions is provided in section 6.2. Finally, since this project is just a small step in this research field, some proposals for future work are described in section 6.3.

6.1 Conclusions

In this graduation project, a tool called SACSEA was developed that computes the lifespans of code smell instances in a software repository. As a case study, SACSEA has been applied to seven software projects in order to answer four research questions regarding the lifespan of code smells and the refactoring behaviour of software engineers. The application is currently still a research prototype and requires further development if the work done in this graduation project is to be extended. Within the threats to validity, some useful results have emerged from the case study and are described below, along with a concluding answer per research question:

RQ1 *Are some types of code smells refactored more and quicker than other smell types?*

The first research question focuses on the overall average lifespans per code smell type, which could be an indication of the priority that developers attach to refactoring certain code smells. To this end, SACSEA was used to analyse seven Java projects over as many revisions as possible. The original intent was to determine the lifespan of code smells from the beginning revision until the head revision. Because such a large range of revisions was not feasible for some systems, a subset of reasonably substantial size was examined per software project. The average lifespan per smell type in each system was determined, based on the lifespans of individual code smell instances.

The initial expectations were that software engineers consider refactoring to be of less importance than actually programming new functionality. This was somehow reflected in the results. On average, code smell instances seem to have a lifespan of approximately 50% of the examined revisions. However, there were some small differences per code smell type, where Feature Envy Methods seem to be refactored more than God Classes, Data Classes and Long Parameter List Classes. On first sight, the cause of this phenomenon seems to lie in the fact that Feature Envy Methods are easier to refactor, either by accident or intentionally. Also, God Classes are proven to be difficult refactoring candidates [31], while Data Classes and Long Parameter Lists do not form a big threat in the eyes of many developers.

Overall, this implies that software engineers are not very much interested in refactoring code smells most of the time. In conclusion, some smell types are resolved more than others, but the results do not show whether these refactorings are always deliberate.

RQ2 *Are code smells refactored more at an early or a later stage of a system's life cycle?*

The question posed here concerns the point in the development life cycle at which a code smell is resolved. The approach for obtaining useful results required the first 20% and the last 20% of the revisions analysed for answering RQ1. The average lifespans per code smell type in these subsets of revisions were computed in the same manner as for RQ1.

Before the experiments were performed, it was presumed that the earlier revisions would show a lower average lifespan than the latest revisions, since more code smell instances are usually present in the latter subset. This hypothesis was found to be correct for the very same reason, holding for all code smell types. It is only natural that a system grows over time, but if the number of infected instances grows along with it, then this may be a sign of little refactoring activity throughout the entire life of a system.

According to the results, the majority of the smell instances in the early revisions subset of any subject system are resolved within a few revisions. However, their numbers do not outweigh the increasing number of infected instances that exist for a long period of time. Relative to the first 20% of the revisions of a system, the latest revisions do not contain many smell removals. This is also a sign that refactoring code smells is of little relevance to a developer.

RQ3 *Do some developers refactor more code smells than others and to what extent?*

This research question deals with the behaviour of developers regarding code smells. Are they familiar with the notion of code smells and if so, are they concerned with the infected instances in their own system? To shed some light on this issue, SACSEA was used on the seven subject systems to retrieve the number of intentional refactoring activities per developer within the examined revisions.

The initial assumption was that it is near certain that some developers refactor more than others. However, no assumptions have been made regarding how many refactoring activities

a developer is responsible for, compared to other software engineers. Within each subject system, usually one or two developers refactor more than the rest of their colleagues. The differences are not large: most of the time they are either the only ones who resolve smell instances or either refactor just a few more infected instances than other developers.

Most smell instances found in the case study were removed as a side effect of other maintenance activities or the implementation of new functionality. Refactoring does not seem profitable in a system with high commit activity. These results hint towards low awareness or concern among developers regarding code smells.

RQ4 *What refactoring rationales for code smells can be identified?*

Similar to RQ3, the subject of interest is the refactoring behaviour of developers. In this case, the log messages themselves must be retrieved rather than the number of commits responsible for the removal of a smell instance.

Chapter 1 already mentions a few possible motives. For instance, some infected classes or methods may need to be refactored before functionality can be added or tested. Another example is the introduction of a dedicated refactoring phase in the development life cycle. Finding such motives in the case study was one of the expectations, but was eventually not fulfilled. However, the commit logs show various other rationales, such as cleaning up dead or redundant code and refactoring for the purpose of code smell resolution. More rationales were found regarding performance and readability improvement. However, these motives were derived from commits that were not marked by SACSEA as responsible for smell removals and therefore fall outside the scope of the experiments.

This implies that developers are most certainly aware of code smells in their software projects, although they seem to resolve them for opportunistic reasons, which explains the relatively low refactoring effort in most subject systems.

6.2 Contributions

The contributions of this graduation project to the field of code smell evolution are listed below.

- *SACSEA.* An application that can analyse open source Java systems stored on SVN repositories, find different types of code smells in them within a user-specified range of revisions and generate a visual and textual output containing their lifespans and corresponding metadata. Initially developed to serve one sole purpose, it is also useful as a stand-alone product and can easily be reused or extended.

- *Experiment results.* Seven open source Java projects were analysed using SACSEA in order to give a fair answer to the research questions defined in chapter 1. The results show that developers are aware of code smells, but do not consider them to be a high priority during development.

- *Stepping stone for further research.* The tool and statistics that both resulted from this graduation project may be used to pursue further work in the field of software evolution.

6.3 Future Work

Since this graduation project is only a tiny step in the field of code smell evolution, more research is required to strengthen or validate the claims made in this study or broaden the knowledge by performing more studies using different variables. This section describes recommendations for future work.

6.3.1 Improve and Extend Functionality of SACSEA

SACSEA was built from scratch and is therefore a prototype. In order to accommodate future work, the application can be supplemented with more functionality that eases the recognition and clarity of code smells. An example is the calculation and visualisation of software metrics. They would have to be extracted from the integrated detection modules or the detected smell instances themselves. Metrics can help in determining how severe a software entity suffers from a certain code smell. Another idea for improvement is better visualisation, such as the multiple views as implemented in CodeVizard [1] [45]. Finally, SACSEA would also benefit from functionality that allows the newly obtained information described above to be logged in a structured way.

6.3.2 Investigating more Code Smells

Only five types of code smells were considered within the scope of this graduation project. Investigating more code smells would undoubtedly lead to more polished answers to the same research questions. Within this issue, it might be interesting to vary between popular smell types and smells that are relatively unknown among the public to find out the relation between coincidental and deliberate refactoring.

6.3.3 Investigating Design Smells and Anti-patterns

Similar to Khomh et al. [28] [29], using the same approach to examine design smells and anti-patterns, such as the Swiss Army Knife and Spaghetti Code, may be fruitful. Since design is an activity that engineers have to incorporate in the application, such research may provide insight into the awareness and concern that is given to design smells and anti-patterns.

6.3.4 Using other Code Smell Detection Tools or Approaches

As stated in section 4.3, SACSEA integrates external code smell detection utilities that were developed as prototypes for specific goals. Therefore, they may not extend well to the goals set in this graduation project and show some unexpected behaviour. As a result, some revisions can not be processed by SACSEA. The way to resolve this issue is by performing

this study again, using detection techniques that have proven to be reliable and are more robust. Besides, it is interesting to discover variations in the results if detection techniques are used that are different from the ones used in this graduation project.

6.3.5 Analysing Industrial Systems

The software projects investigated in the case study were all developed by the open source principle. Usually, but not always, this means that developers are not strictly bound by programming guidelines or deadlines. Although every effort has been spent to find several subject systems that incorporate a strict development process, no commercial or closed source projects were used in the case study, due to the limited availability of such systems. Another idea for future work is to redo this research using industrial projects for the case study and find out how a rigid development process or pressing deadlines affect the lifespan of certain code smells or if dedicated refactoring phases are introduced at some point in the life cycle. Also, the commit logs of such systems are bound to be more clear and reliable.

6.3.6 Analysing Projects in other OO-languages or Paradigms

Because SACSEA, or rather its detection modules, can only analyse applications written in Java, no research has been done using systems that were developed in other object-oriented languages or programming paradigms. It would be interesting to see how much the average lifespans in such projects differ from the lifespans that were determined in this graduation project. If this research were to be performed again with such systems, SACSEA would definitely have to be extended with functionality to support the detection of code smells in those systems.

Bibliography

[1] CodeVizard. http://hpcs.cs.umd.edu/index.php?id=21, Website last visited: June 2011. *Cited on pages 43 and 50.*

[2] cvs2svn. http://cvs2svn.tigris.org/, Website last visited: June 2011. *Cited on page 22.*

[3] JFreeChart. http://www.jfree.org/jfreechart/, Website last visited: June 2011. *Cited on page 22.*

[4] Maven. http://maven.apache.org/, Website last visited: June 2011. *Cited on page 19.*

[5] Ptidej. http://www.ptidej.net/, Website last visited: June 2011. *Cited on page 14.*

[6] Sourceforge. http://sourceforge.net/, Website last visited: June 2011. *Cited on page 23.*

[7] Subversion (SVN). http://subversion.apache.org/, Website last visited: June 2011. *Cited on page 3.*

[8] SVNKit. http://www.svnkit.com/, Website last visited: June 2011. *Cited on pages 4 and 7.*

[9] Tigris. http://www.tigris.org/, Website last visited: June 2011. *Cited on page 23.*

[10] VisualSVN. http://www.visualsvn.com/, Website last visited: June 2011. *Cited on page 23.*

[11] XOM. http://www.xom.nu/, Website last visited: June 2011. *Cited on page 20.*

[12] A. Begel, Y.P. Khoo, and T. Zimmermann. Codebook: Discovering and Exploiting Relationships in Software Repositories. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 125–134, 2010. *Cited on page 16.*

[13] K.H. Bennett and V.T. Rajlich. Software Maintenance and Evolution: A Roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 73–87. ACM, 2000. *Cited on page 4.*

[14] I. Macia Bertran, A. Garcia, and A. von Staa. An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, pages 203–214. ACM, 2011. *Cited on pages 3 and 45.*

[15] M. Bombardieri and F.A. Fontana. Software Aging Assessment through a Specialization of the SQuaRE Quality Model. In *IEEE Workshop on Software Quality, co-located event with ICSE 2009, Vancouver*, page 34, 2009. *Cited on page 1.*

[16] W.J. Brown, R.C. Malveau, H.W. McCormick III, and T.J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, 1998. (ISBN 0 471 19713 0). *Cited on page 2.*

[17] A. Chatzigeorgiou and A. Manakos. Investigating the Evolution of Bad Smells in Object-Oriented Code. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 106–115. IEEE, 2010. *Cited on page 44.*

[18] M. D'Ambros and M. Lanza. Reverse Engineering with Logical Coupling. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, page 189. IEEE, 2006. *Cited on page 4.*

[19] C. Darwin. *On The Origin of Species*. John Murray, 1859. *Cited on page 1.*

[20] S. Demeyer, S. Ducasse, and O. M. Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, Switzerland, 2003. (ISBN 978 3 9523341 2 6). *Cited on page 8.*

[21] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 109–118. IEEE Computer Society, 1999. *Cited on page 3.*

[22] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. (ISBN 0 201 48567 2). *Cited on pages 3, 7, 10, 11, 12, and 43.*

[23] D.J. Futuyma. *Evolution*. Sinauer Associates, Sunderland, MA, 2005. (ISBN 0 878 93187 2). *Cited on page 1.*

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Reading, MA, 1995. (ISBN 0 201 63361 2). *Cited on page 2.*

[25] Y.-G. Guéhéneuc. Ptidej: Promoting Patterns with Patterns. In *Proceedings of the first ECOOP workshop on Building a System using Patterns*, 2005. *Cited on page 14.*

[26] H. Kagdi, M. L. Collard, and J. I. Maletic. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. *Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007. *Cited on pages 4 and 15.*

[27] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System For Large Scale Source Code. *IEEE Transactions on Software Engineering*, pages 654–670, 2002. IEEE. *Cited on page 3.*

[28] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 75–84. IEEE, 2009. *Cited on pages 44 and 50.*

[29] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. *Empirical Software Engineering (EMSE)*, 6(1):141–175, 2011. *Cited on pages 12, 44, and 50.*

[30] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. *ACM SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005. ACM. *Cited on page 3.*

[31] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer-Verlag Berlin Heidelberg, Germany, 2006. (ISBN 978 3 540 24429 5). *Cited on pages 8, 10, 11, 32, 33, 43, and 48.*

[32] M.M. Lehman and J.F. Ramil. Towards a Theory of Software Evolution - And its Practical Impact. In *Proceedings of the International Symposium on Principles of Software Evolution*, pages 2–11. IEEE, 2000. *Cited on page 1.*

[33] A. Lozano, M. Wermelinger, and B. Nuseibeh. Assessing the Impact of Bad Smells using Historical Information. In *Ninth International Workshop on Principles of Software Evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 31–34. ACM, 2007. *Cited on page 43.*

[34] N.H. Madhavji. Introduction to the Panel Session - Lehman's Laws of Software Evolution, in Context. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 66–67, 2002. *Cited on page 1.*

[35] N. Moha and Y.-G. Guéhéneuc. Ptidej and DECOR: Identification of Design Patterns and Design Defects. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 868–869. ACM, 2007. *Cited on page 15.*

[36] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.F. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *Software Engineering, IEEE Transactions on*, 36(1):20–36, 2010. IEEE. *Cited on pages 2 and 44.*

[37] R. Moser, W. Pedrycz, A. Sillitti, and G. Succi. A Model to Identify Refactoring Effort during Maintenance by Mining Source Code Repositories. *Product-Focused Software Process Improvement*, pages 360–370, 2008. Springer. *Cited on page 45.*

[38] S.M. Olbrich, D.S. Cruzes, V. Basili, and N. Zazworka. The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems. *Empirical Software Engineering and Measurement, 3rd International Symposium on*, pages 390–400, 2009. IEEE Computer Society. *Cited on page 44.*

[39] S.M. Olbrich, D.S. Cruzes, and D.I.K. Sjøberg. Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10. IEEE, 2010. *Cited on page 43.*

[40] D.L. Parnas. Software Aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279–287. IEEE Computer Society Press, 1994. *Cited on pages 1, 2, and 4.*

[41] B. Pietrzak and B. Walter. Leveraging Code Smell Detection with Inter-Smell Relations. *Extreme Programming and Agile Processes in Software Engineering*, pages 75–84, 2006. Springer. *Cited on page 43.*

[42] E.K. Piveta, M. Hecht, M.S. Pimenta, and R.T. Price. Detecting Bad Smells in AspectJ. *Journal of Universal Computer Science*, 12(7):811–827, 2006. *Cited on page 45.*

[43] W. Scacchi. Understanding Open Source Software Evolution. *Software evolution and feedback: theory and practice*, pages 181–206, April 2003. *Cited on page 1.*

[44] N. Tsantalis. *Evaluation and Improvement of Software Architecture: Identification of Design Problems in Object-Oriented Systems and Resolution through Refactorings*. PhD thesis, University of Macedonia, Thessaloniki, August 2010. *Cited on page 13.*

[45] N. Zazworka and C. Ackermann. CodeVizard: A Tool to Aid the Analysis of Software Evolution. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, page 63:1. ACM, 2010. *Cited on pages 43 and 50.*

Appendix A

Glossary

Anti-pattern A commonly occurring design solution that will always generate negative consequences when applied to a recurring problem.

AOP Aspect-Oriented Programming. A programming paradigm that aims to increase the modularity of a software program through the separation of cross-cutting concerns.

AST Abstract Syntax Tree. A tree representation of the abstract syntactic structure of a program. Each node of the tree denotes a construct occurring in the source code. The syntax is "abstract" in the sense that it does not represent every detail that appears in the real syntax.

ATFD A software metric that stands for Access To Foreign Data. It counts how many attributes from foreign classes are accessed directly from the considered class.

Checkin See *Commit*.

Checkout The result of creating a local working copy from the repository. A user may specify a specific revision.

Code refactoring A disciplined way to restructure code, undertaken in order to improve some of the non-functional attributes of software. Code smells and anti-patterns can be resolved by applying refactorings: tiny changes in the source code that do not modify its functional requirements.

Code smell Symptom of anti-patterns. Examples are large classes and methods, redundant message passing and poor information hiding.

Commit Also known as checkin. The result of checking in a working copy into a software repository, thus creating a new revision.

Coupling The dependence between classes. According to one of the design principles in object-oriented programming, coupling must be kept as low as possible.

CSV Comma-Separated Values. This file format is widely used to store tabular data, which can be read in plain textual form in a text editor. Lines in the text file represent table rows. Commas or semicolons in a line separate the data fields.

CVS Concurrent Versions System, a software revision control system. Some software repositories used in this graduation project were originally CVS repositories, before they were converted by the cvs2svn utility to facilitate the operation of SACSEA.

Cyclomatic complexity A software metric that is used to measure the complexity of a program. It directly computes the number of linearly independent execution paths through a program's source code.

Data Class A class that typically only contains many data attributes and corresponding accessor methods.

Design pattern A reusable solution to a recurring problem in software design.

DSL Domain-Specific Language. A programming or specification language dedicated to a particular problem domain, a certain problem representation technique or a specific solution technique.

Encapsulate Field This refactoring makes a public attribute private and provides accessor methods for it.

Extract Class A refactoring that creates a new class and moves relevant attributes and methods to it from an old class.

Extract Method A refactoring that creates a new method from a fragment of another method.

FDP Foreign Data Providers. This software metric counts the number of unrelated classes that contain the foreign data that are accessed by a method.

Feature Envy A method suffers from this smell if it is more interested in a foreign class than its own host class.

God Class A big, complex, inelegant and low-cohesive class that implements a large part of the system's functionality, which makes it hard to understand and maintain.

Hide Delegate A refactoring that removes delegate entities from a call chain and makes the caller depend solely on the object at the head of the chain.

Introduce Parameter Object A refactoring that replaces a group of parameters with a newly created object.

LAA A software metric that stands for Locality of Attribute Accesses. It is defined as the result of the number of accessed foreign attributes divided by the total number of accessed data in the analysed method.

LOC Lines Of Code or Source Lines Of Code. A software metric used to measure the size of a software program by counting the number of lines in its source code.

Logical coupling Implicit and evolutionary dependencies between the artifacts of a system which, although potentially not structurally related, evolve together and are therefore linked to each other from an evolutionary point of view.

Long Parameter List A method suffers from this smell if it contains a certain minimum amount of parameters. In this graduation project, a method is infected if it has four or more parameters.

Maven A software utility for project management and build automation, mainly used for Java projects. Essentially, Maven dynamically downloads Java libraries and Maven plug-ins from one or more repositories. These units are used by the affected Java project as dependencies and are described using a *Project Object Model*, which is stored in a *pom.xml*-file.

Message Chain Class A class that implements its (data access) functionality by using a long chain of method invocations or temporary variables between different classes.

Metric See *Software metric*.

Move Method This refactoring creates a new method from another method with a similar body and moves it into the class it uses the most.

MSR Mining Software Repositories - See Software Repository Mining.

NOAM A software metric that stands for Number Of Accessor Methods. It is defined as the number of non-inherited accessors of the considered class.

NOPA A software metric that stands for Number Of Public Attributes. It represents the number of non-inherited attributes that belong to the considered class.

OOP Object-Oriented Programming. A programming paradigm that emphasises the use of objects, which are data structures consisting of data fields and methods together with their interactions.

Preserve Whole Object A refactoring that replaces parameters containing data from an object by passing the object itself as a parameter.

Project Object Model See *Maven*.

Refactoring See *Code refactoring*.

Remove Setting Method A refactoring that removes a mutator method for a data field that should not be changed.

Replace Conditional with Polymorphism This refactoring moves each case of a conditional statement to an overriding method in a subclass and makes the original method abstract.

Replace Parameter with Method A refactoring that removes a parameter by calling a method to retrieve the data that were originally passed as that parameter.

Replace Type Code with State/Strategy This refactoring replaces type code that can not be subclassed with a state object.

Repository See *Software repository*.

Revision The state of the contents of a software repository at some point in time.

Ripple effect The consequence of poor data-operation proximity. Changes in operations trigger changes in other methods in the same call chain. This will eventually lead to software defects, as bugs will also be propagated in the same manner.

SACSEA Acronym of Semi-Automatic Code Smell Evolution Assistant. This is the name given to the application that was developed for this graduation project and was used to determine the lifespans of certain code smells in a software repository.

Software evolution A specific discipline of software research that studies and manages the process of repeatedly making changes to software over time for various reasons. It tries to provide theoretical knowledge and a set of best practices in order to understand the causes and consequences of software ageing.

Software metric A measure of some property of source code, such as size, complexity and amount of reuse.

Software repository The repository is where the current and historical data of files are stored. More than often, a server is used to manage a software repository.

Software Repository Mining A field in software engineering research that aims to devise methods that extract metadata from software repositories to uncover evolutionary relationships. It is similar to the field of data mining.

Subversion A software versioning and a revision control system. The software repositories used in this graduation project are based on Subversion.

SVN Abbreviation of Subversion.

TCC A software metric that stands for Tight Class Cohesion. It counts the relative number of methods that access the same attribute.

UI User Interface.

VCS Version Control System. A system for managing multiple revisions of a software project.

Version See *Revision*.

WMC Weighted Method Count. This software metric sums up the cyclomatic complexities of the methods of a class.

WOC A software metric that stands for Weight Of Class. It represents the number of non-accessor methods divided by the total number of interface members.

Working copy The working copy is the local copy of files from a repository at a specific revision. All work done to the files in a repository is initially done on a working copy.

XML Extensible Markup Language. A standard of W3C for the syntax of formal markup languages with which structured data can be represented in the form of text. This representation is made to be read by machines as well as humans. The XML-format is widely used to store data and transmit data over the Internet.

Appendix B

Number of Analysed Revisions

As mentioned in chapter 4, the number of analysed revisions of each software project per smell type varies due to unparsable revisions and the limited operation of the integrated code smell detection tools. The two tables below show the number of revisions in each system per smell type that have been examined. Again, the abbreviations for the code smells used in section 4.2 are also used here for convenience.

	GC	FEM	DC	MCC	LPLC
CalDAV4j	318	318	318	318	318
Evolution Chamber	282	282	282	282	282
JDiveLog	770	872	872	872	872
jGnash	1302	1302	988	945	1493
Saros	2090	2326	2482	2472	2482
VLCJ	1502	1445	1502	1502	1502
Vrapper (Base)	115	115	115	115	115
Vrapper (Core branch)	231	223	231	231	231

Table B.1: Number of analysed revisions (RQ1, RQ3 and RQ4).

	GC	FEM	DC	MCC	LPLC
CalDAV4j	64	64	64	64	64
Evolution Chamber	57	57	57	57	57
JDiveLog	154	175	175	175	175
jGnash	261	261	198	189	299
Saros	418	466	497	495	497
VLCJ	301	289	301	301	301
Vrapper (Base)	23	23	23	23	23
Vrapper (Core branch)	47	45	47	47	47

Table B.2: Number of earliest and latest 20% of the analysed revisions (RQ2).

Appendix C

Gantt charts

Below, the Gantt charts per code smell type for every subject system are presented.

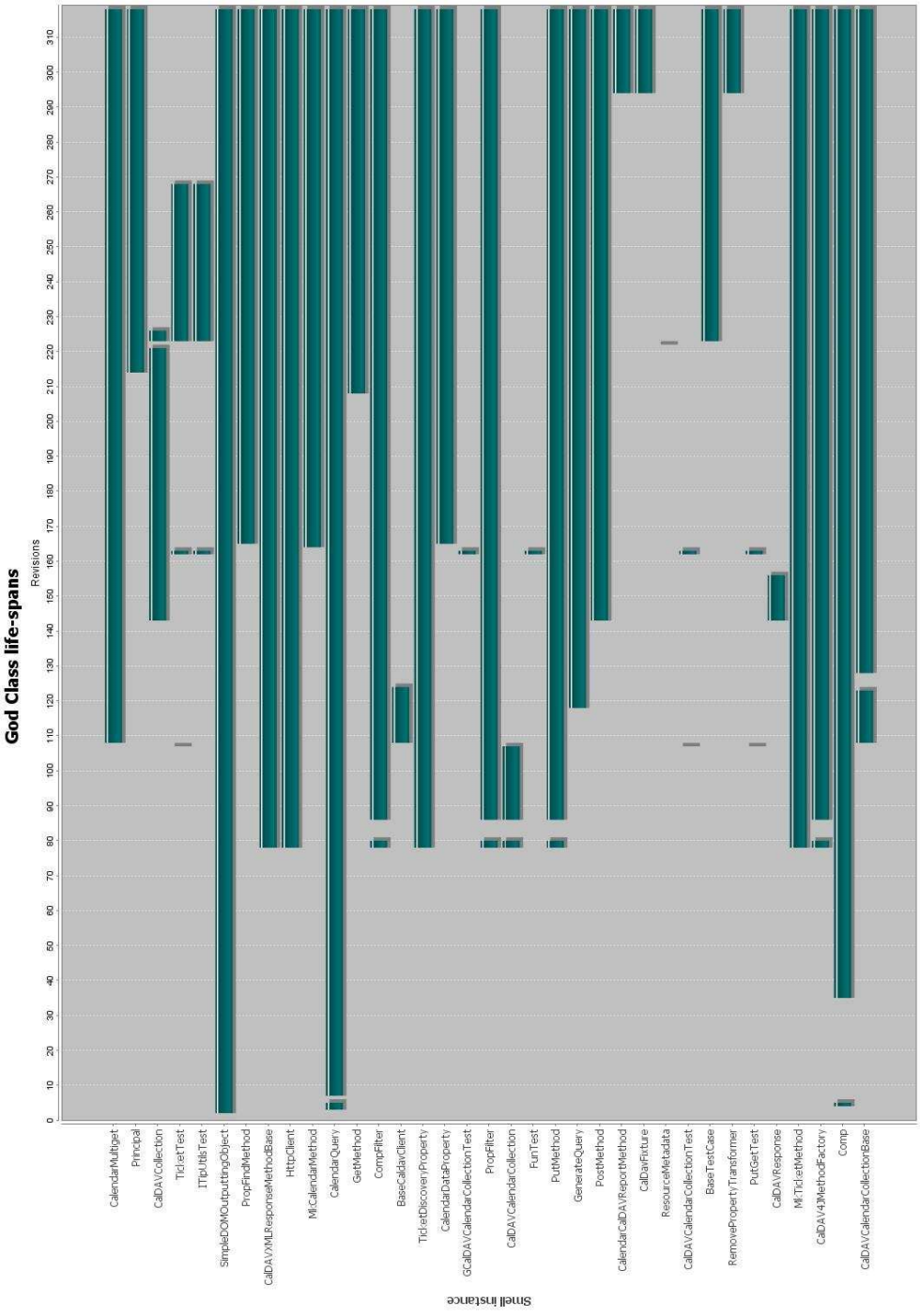


Figure C.1: Lifespans of God Classes in CalDAV4j.

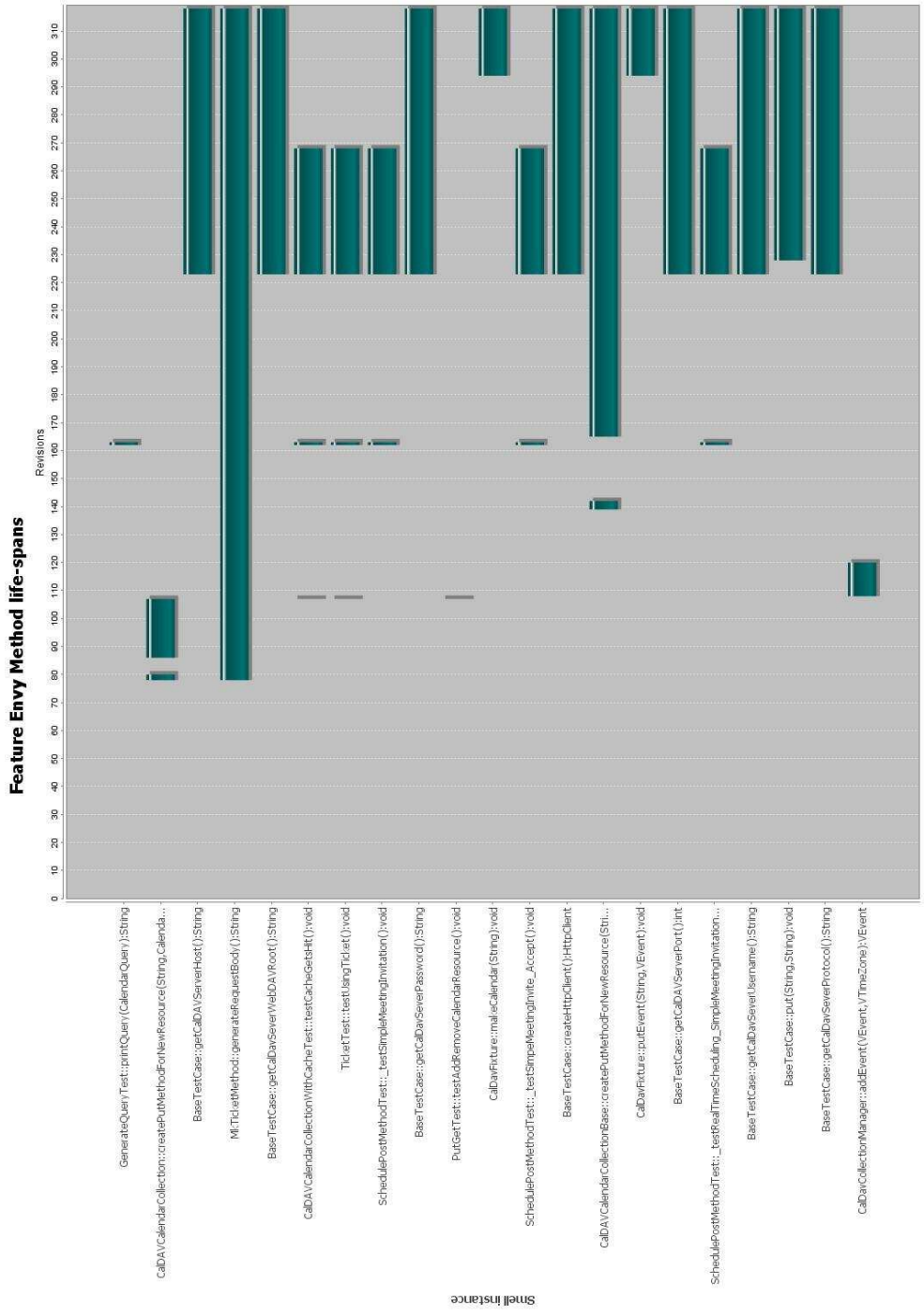


Figure C.2: Lifespans of Feature Envy Methods in CalDAV4j.

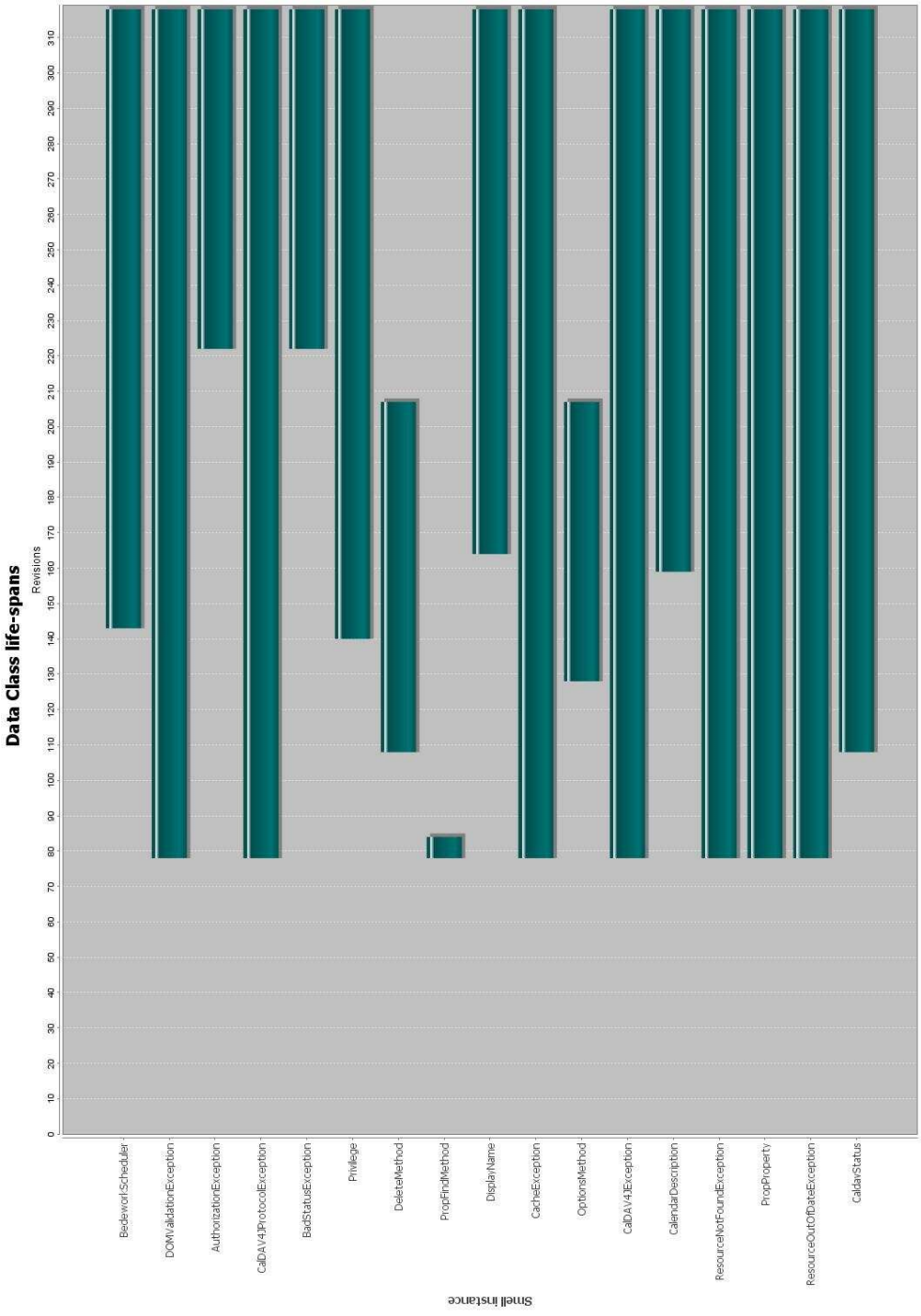


Figure C.3: Lifespans of Data Classes in CalDAV4j.

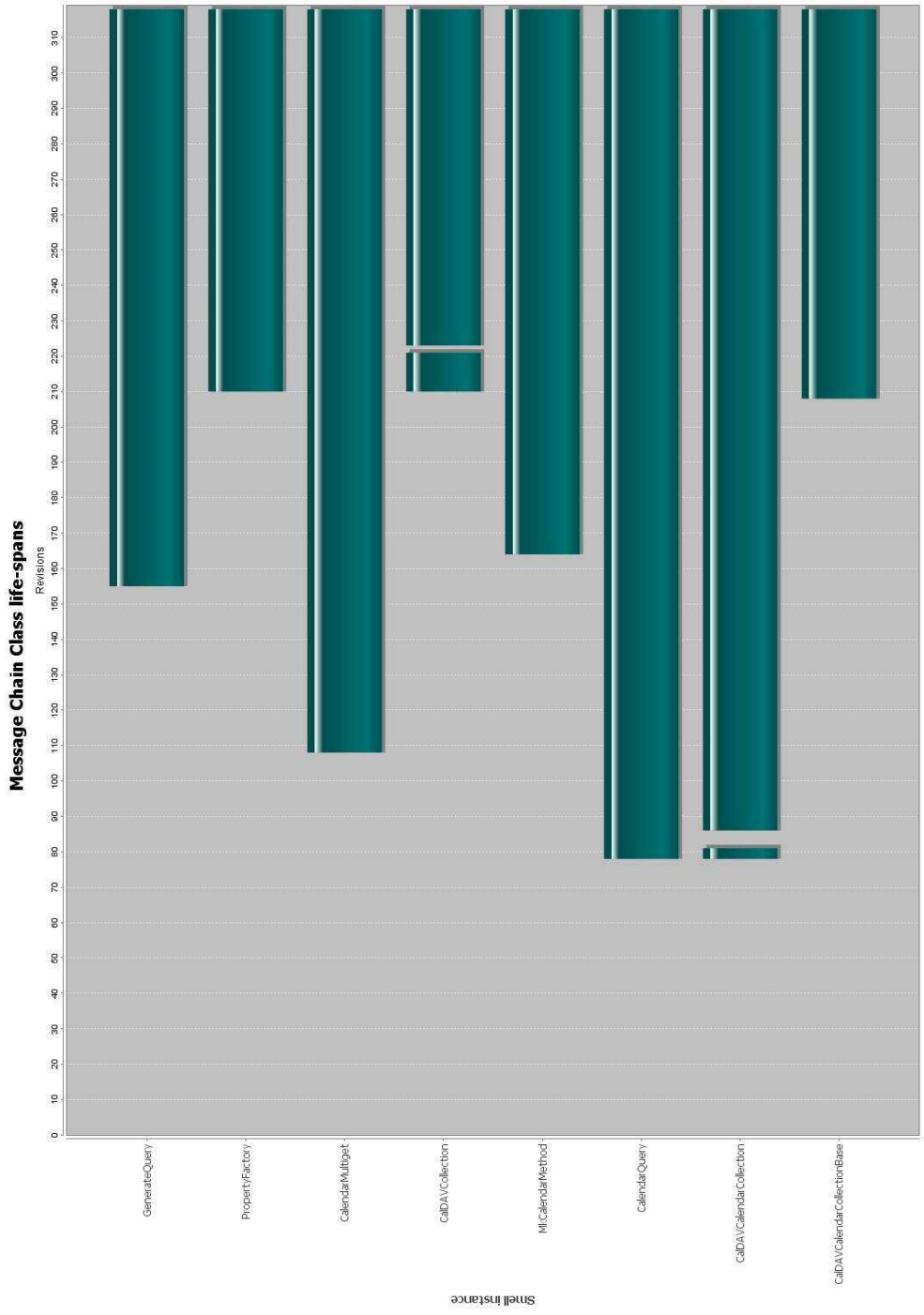


Figure C.4: Lifespans of Message Chain Classes in CalDAV4j.

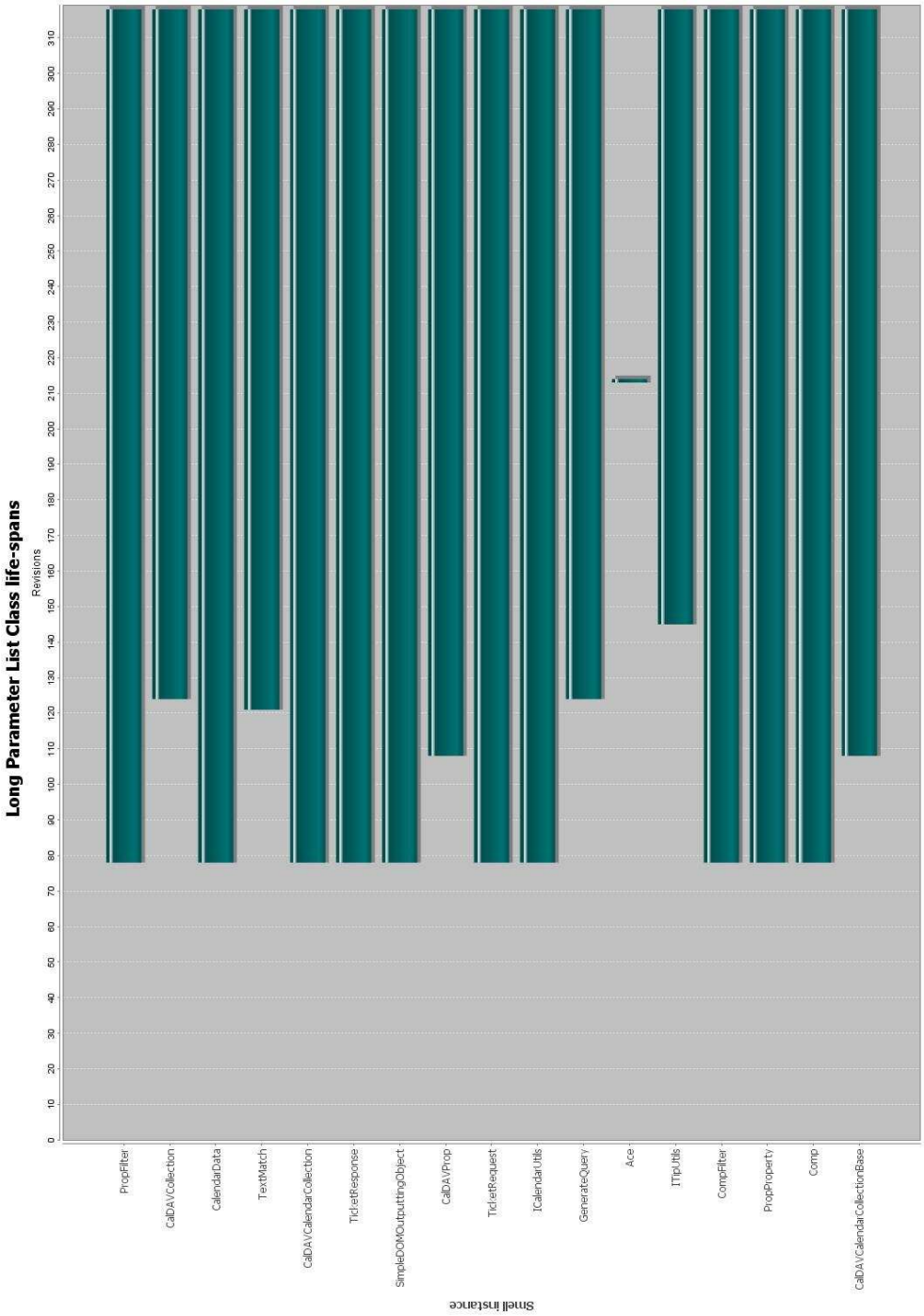


Figure C.5: Lifespans of Long Parameter List Classes in CalDAV4j.

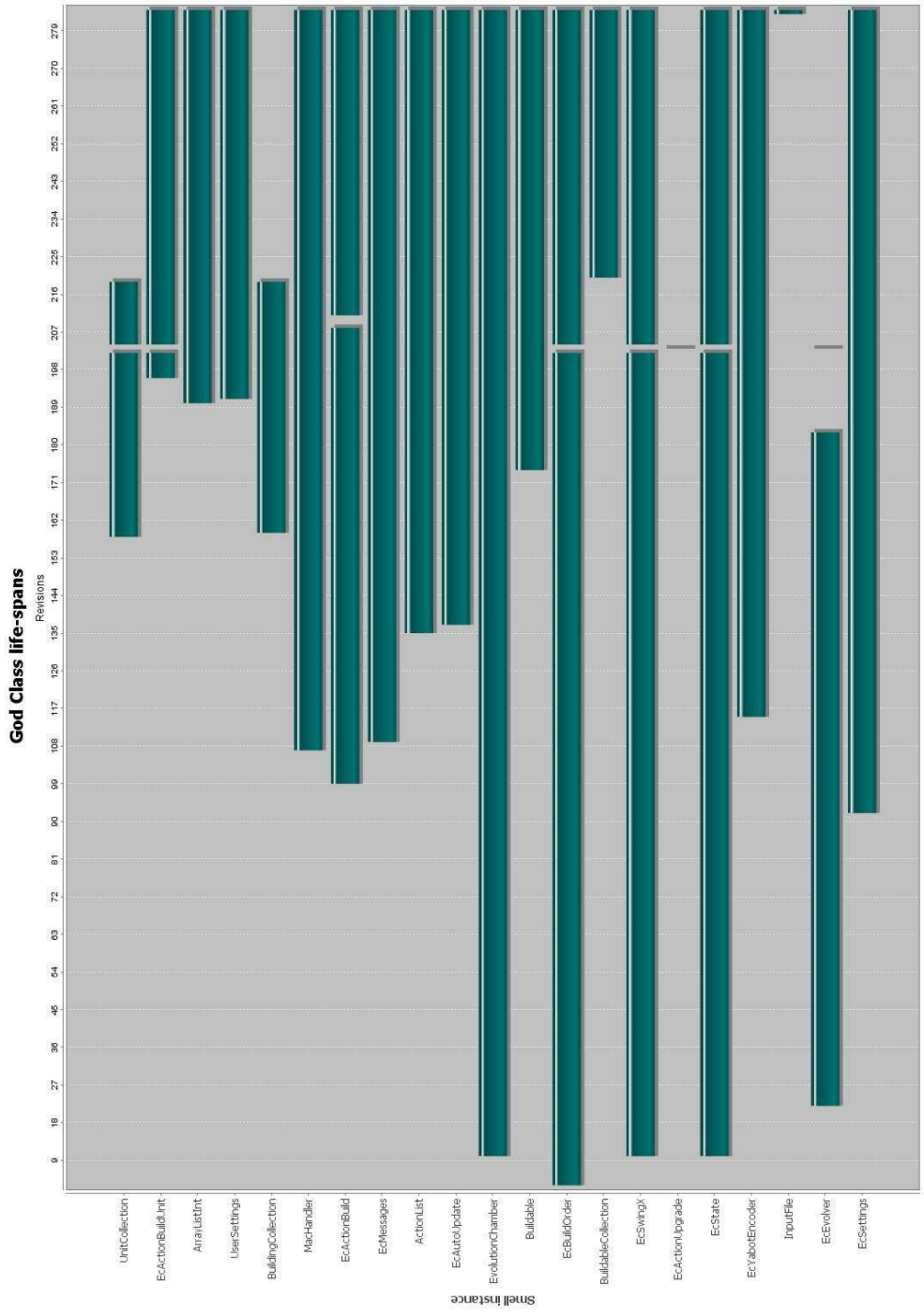


Figure C.6: Lifespans of God Classes in Evolution Chamber.

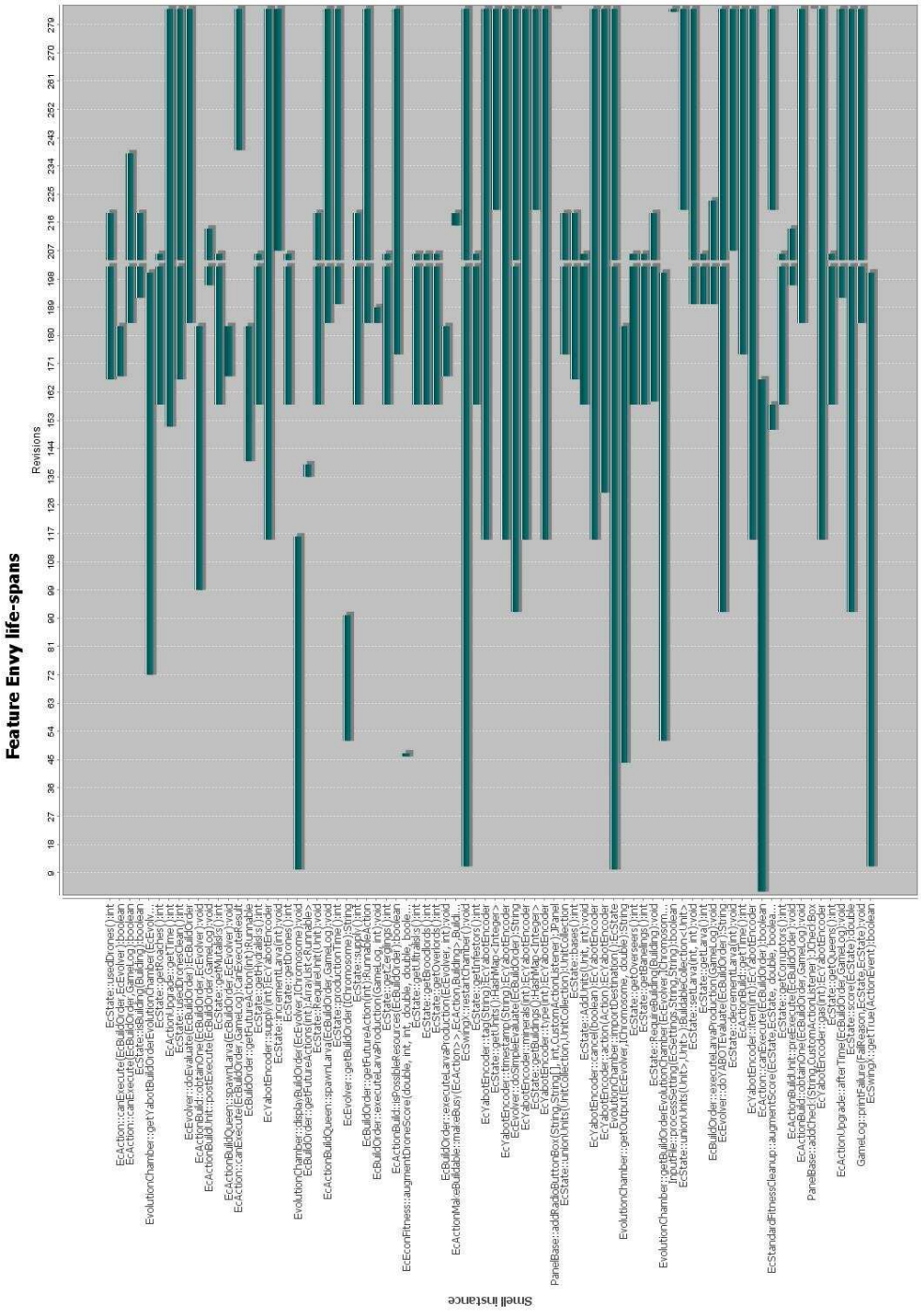


Figure C.7: Lifespans of Feature Envy Methods in Evolution Chamber.

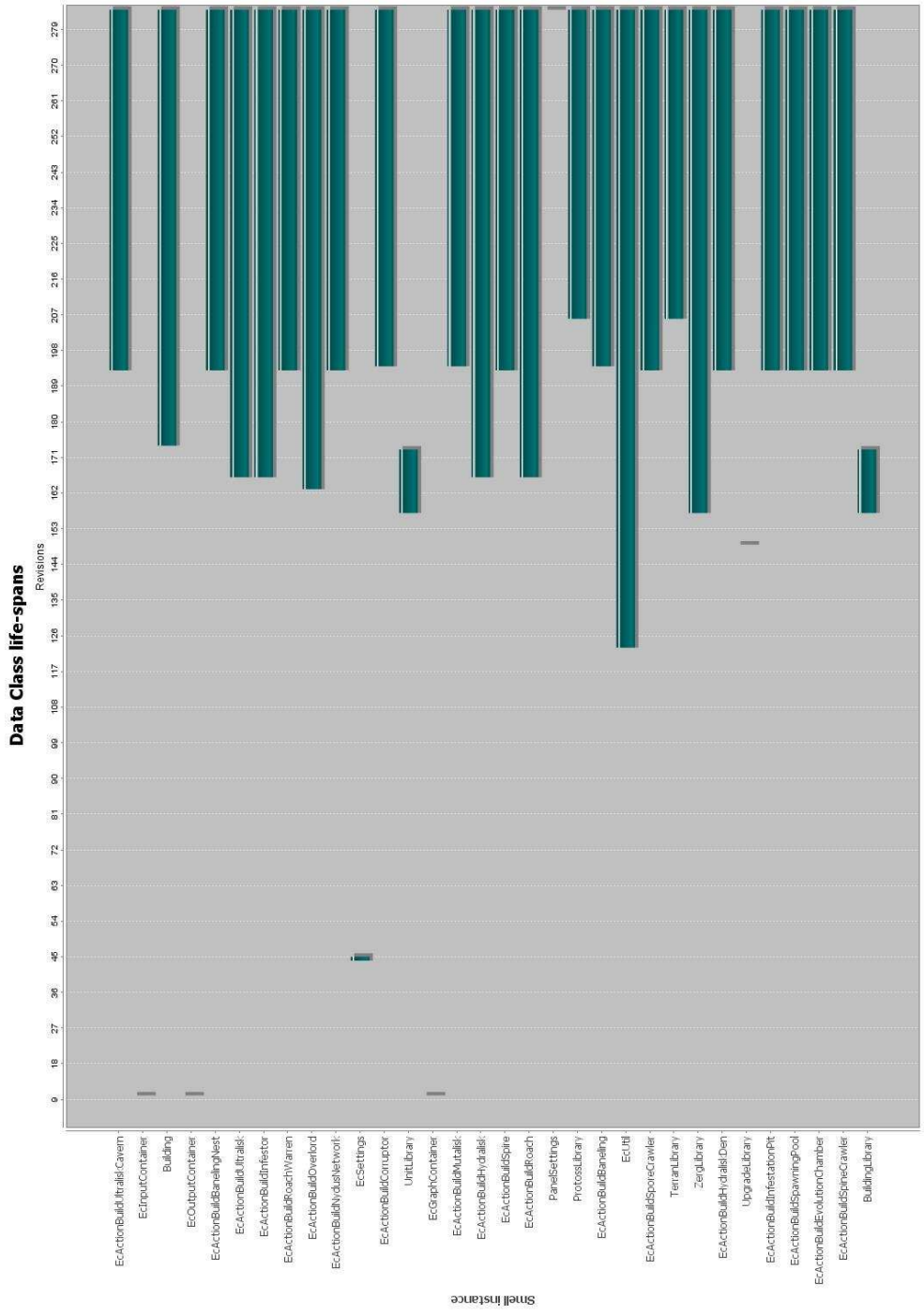


Figure C.8: Lifespans of Data Classes in Evolution Chamber.

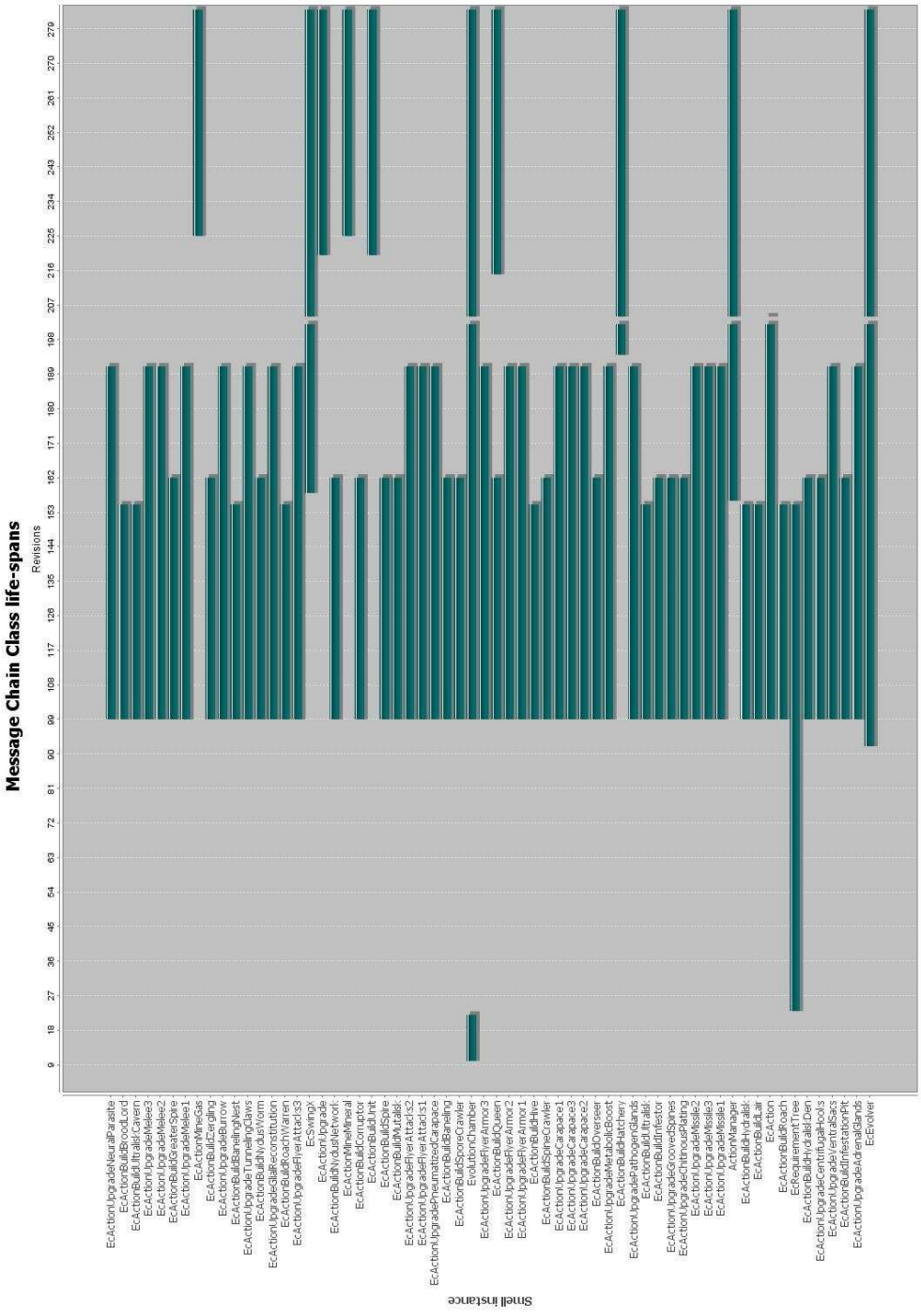


Figure C.9: Lifespans of Message Chain Classes in Evolution Chamber.

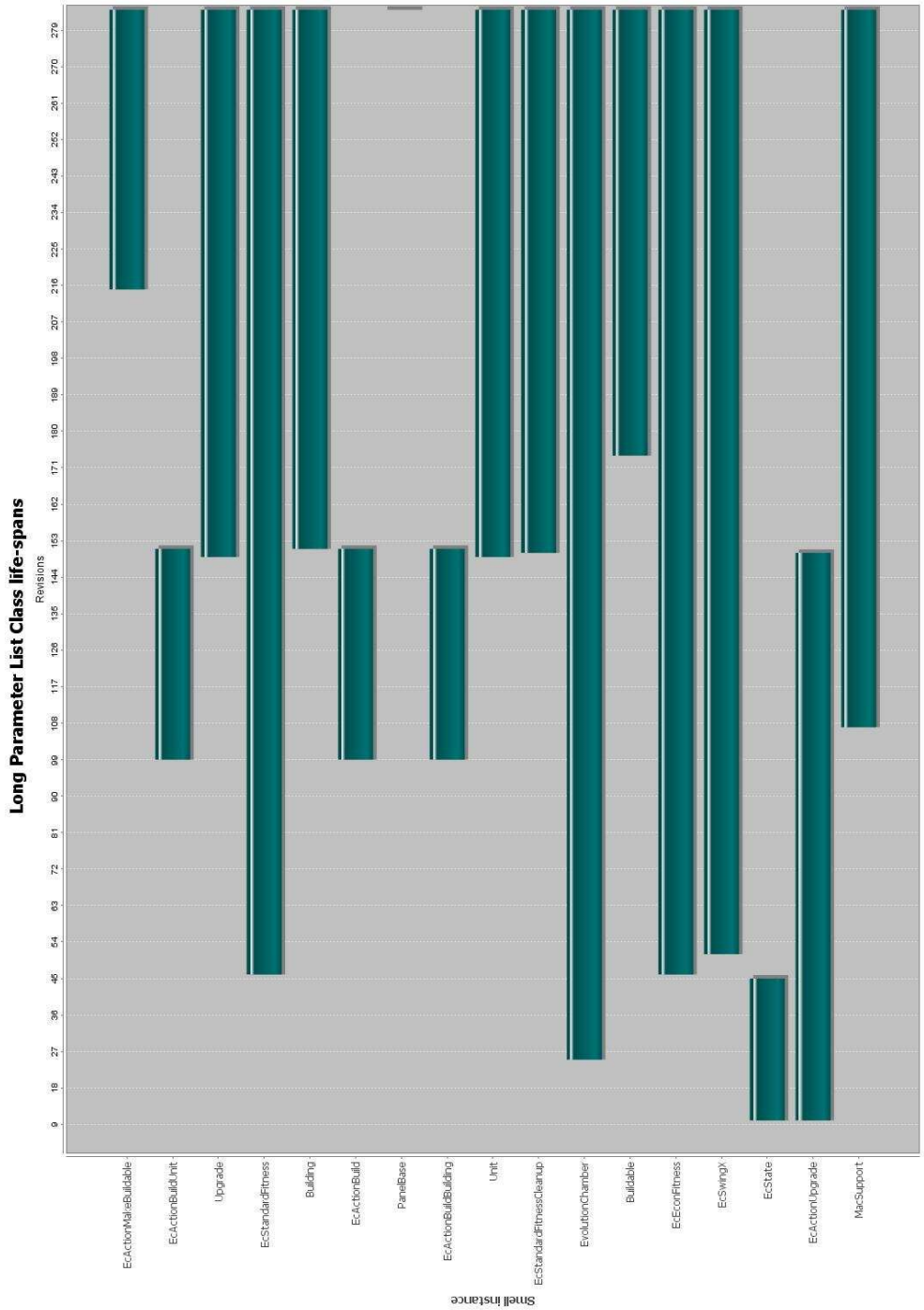


Figure C.10: Lifespans of Long Parameter List Classes in Evolution Chamber.

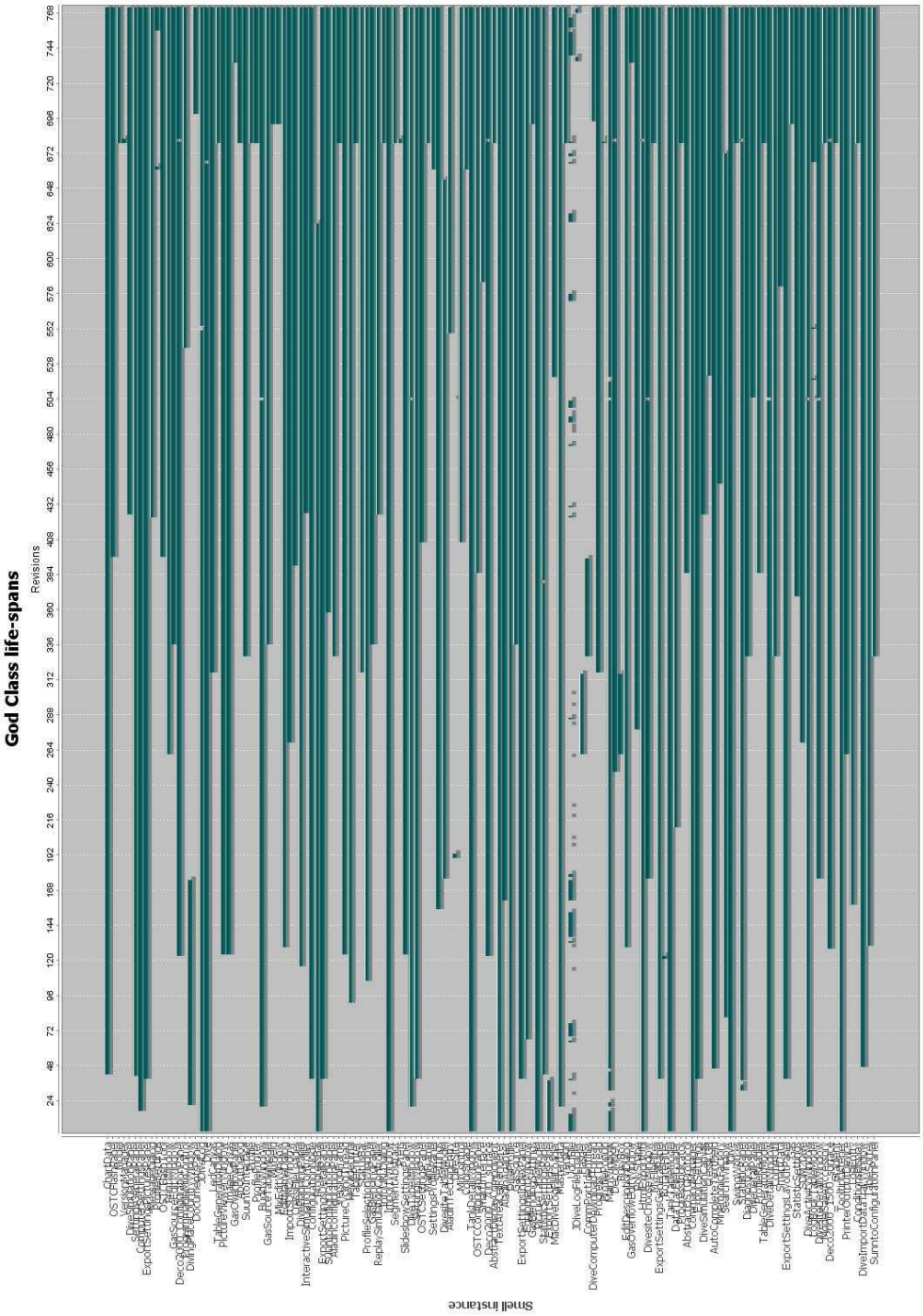


Figure C.11: Lifespans of God Classes in JDiveLog.

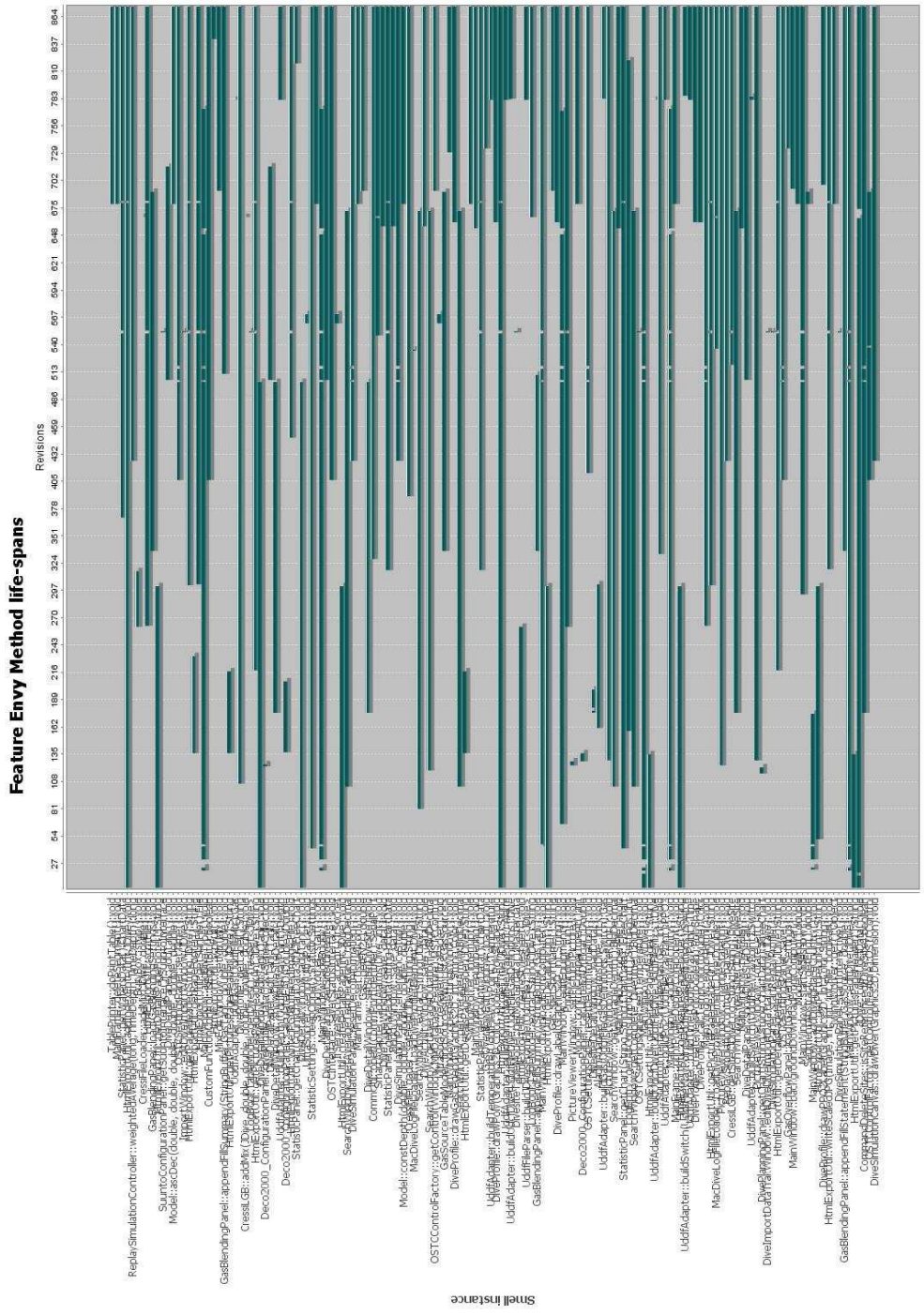


Figure C.12: Lifespans of Feature Envy Methods in JDiveLog.

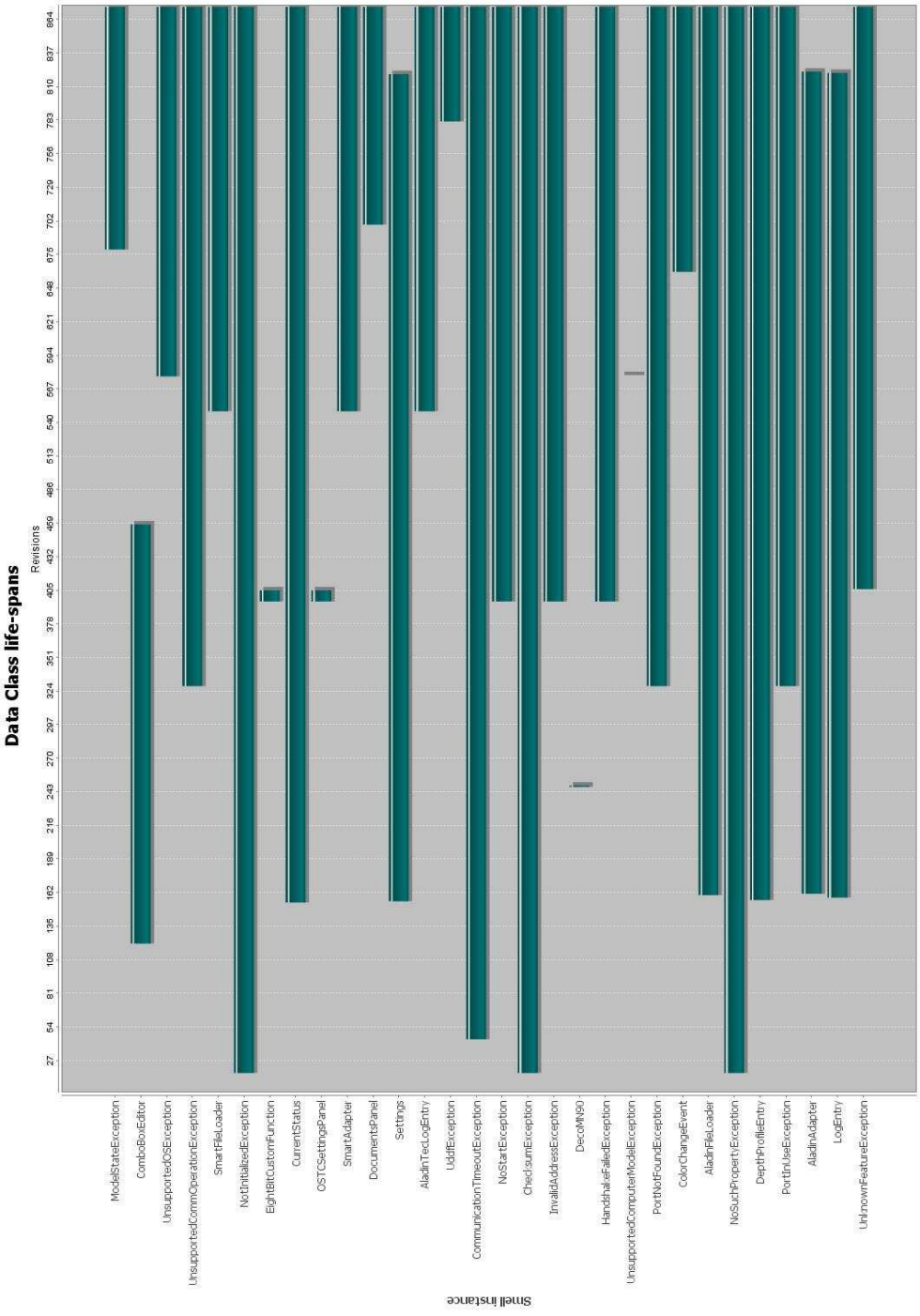


Figure C.13: Lifespans of Data Classes in JDiveLog.

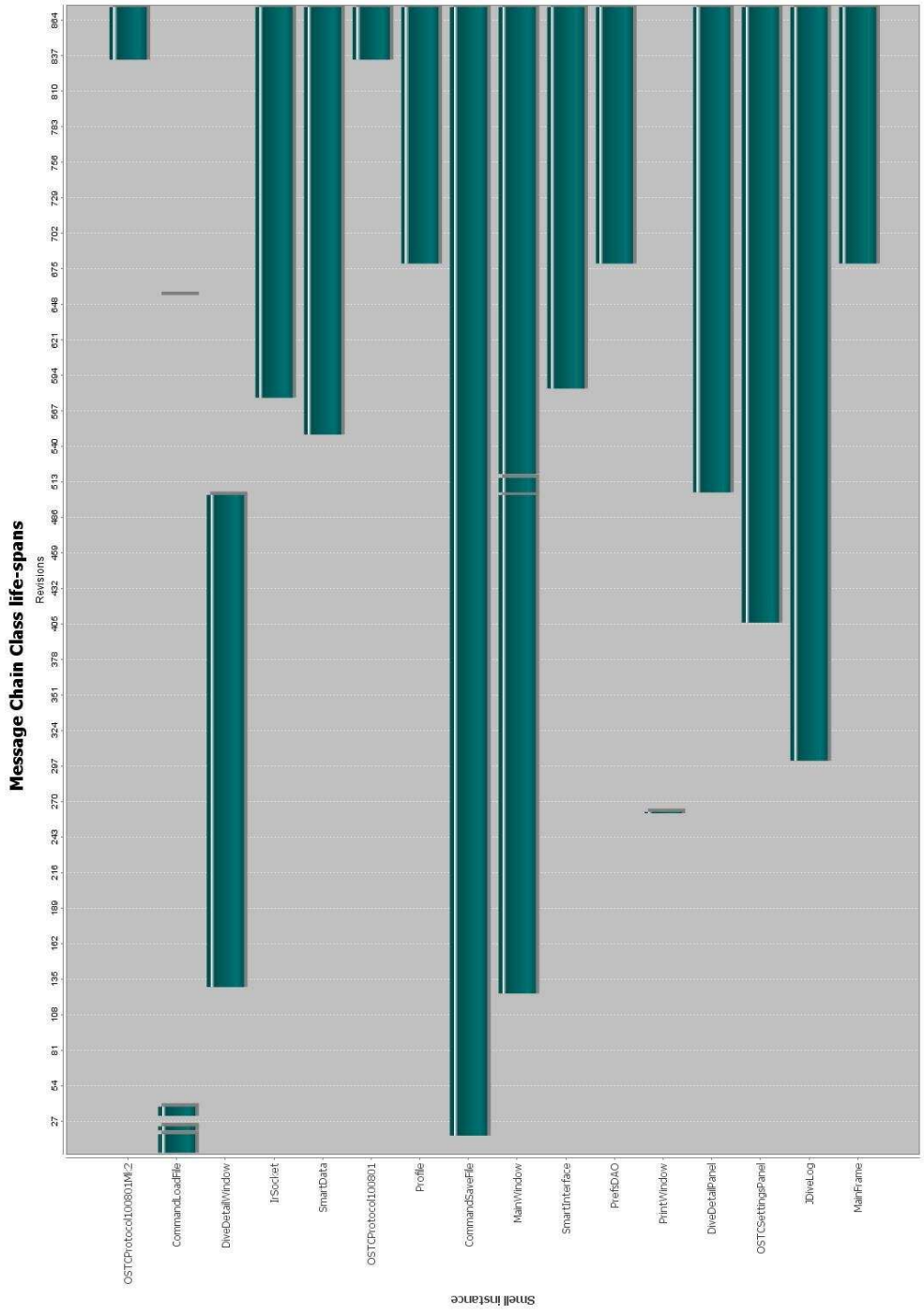


Figure C.14: Lifespans of Message Chain Classes in JDiveLog.

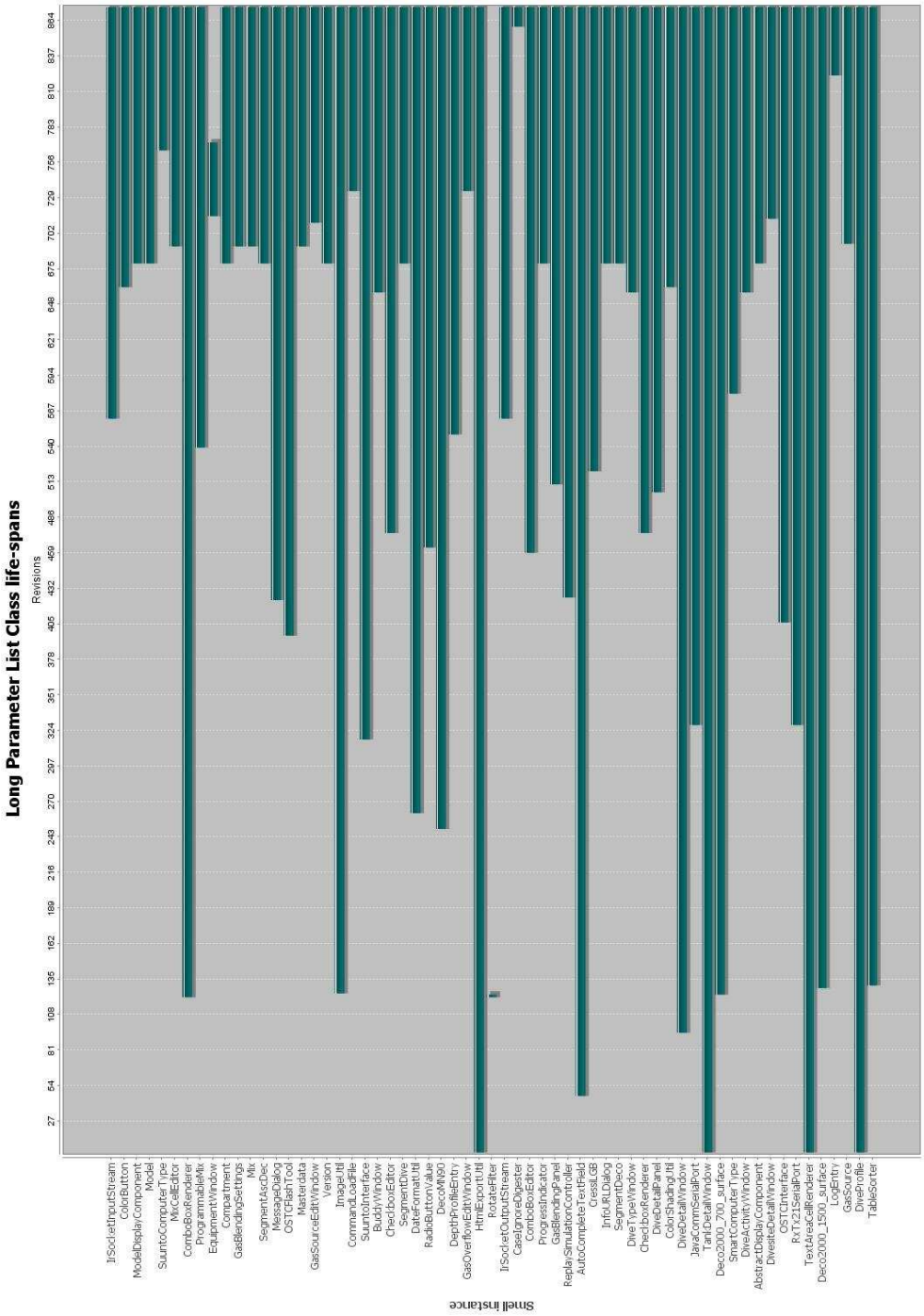


Figure C.15: Lifespans of Long Parameter List Classes in JDiveLog.

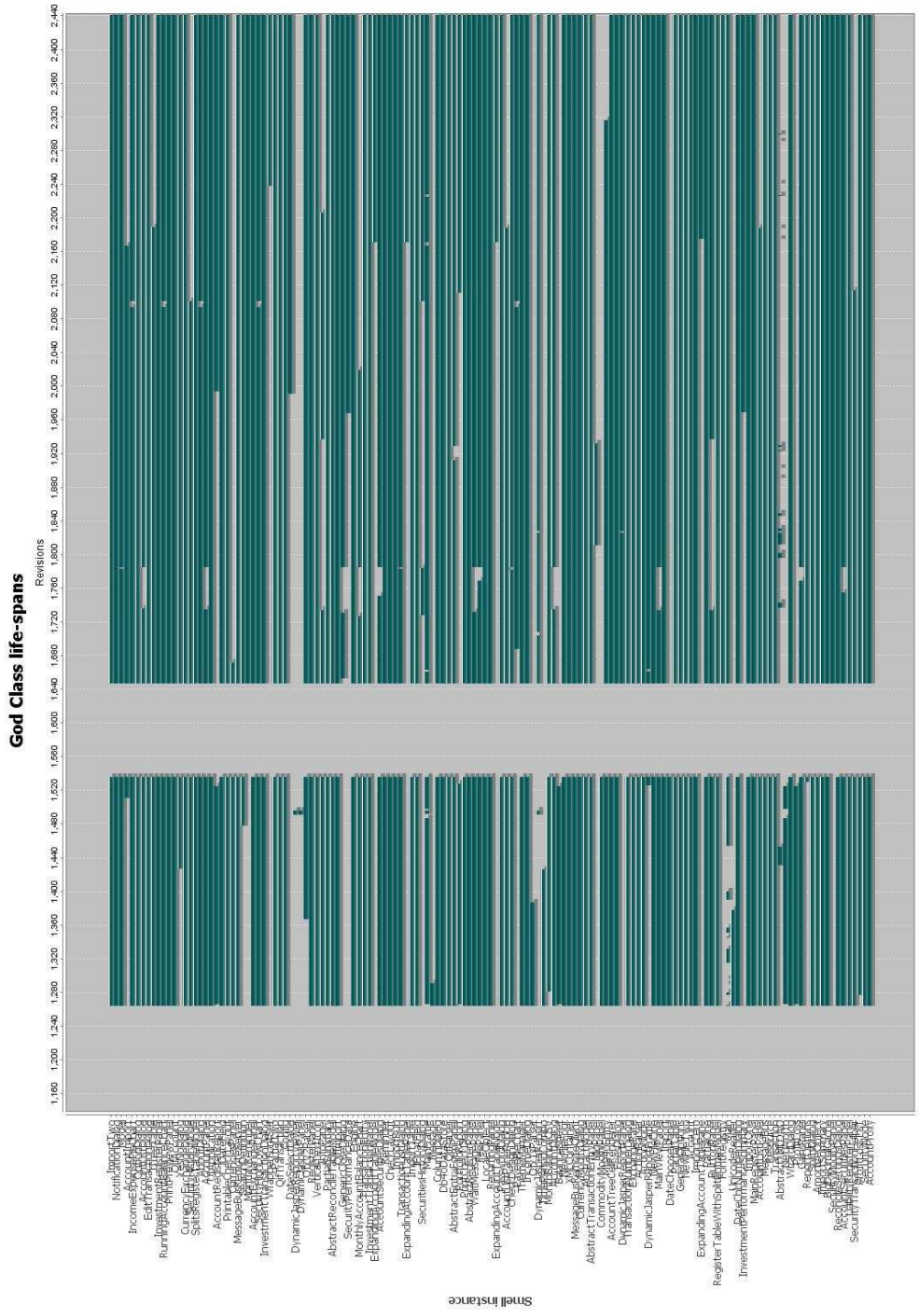


Figure C.16: Lifespans of God Classes in jGnash.

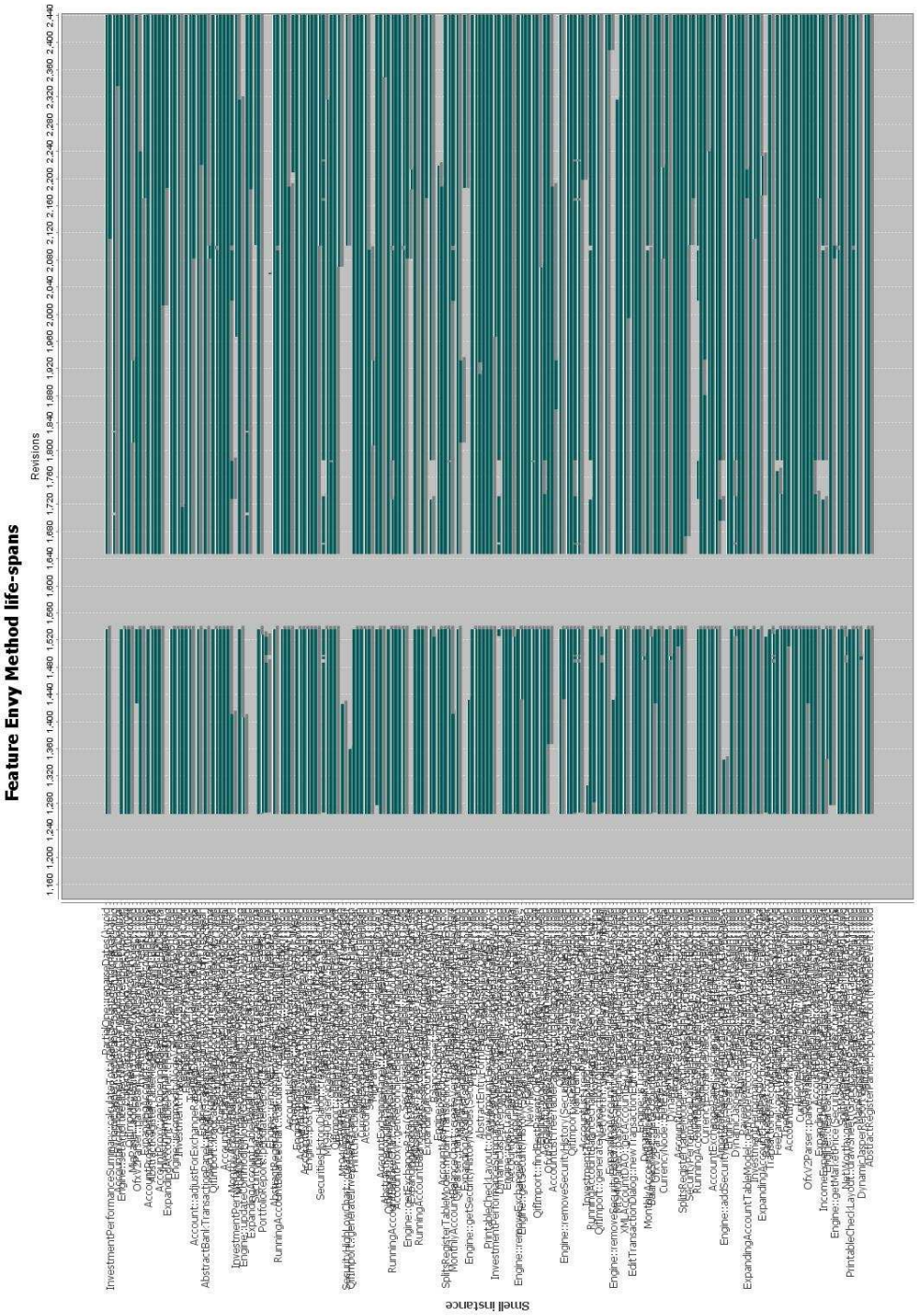


Figure C.17: Lifespans of Feature Envy Methods in jGnash.

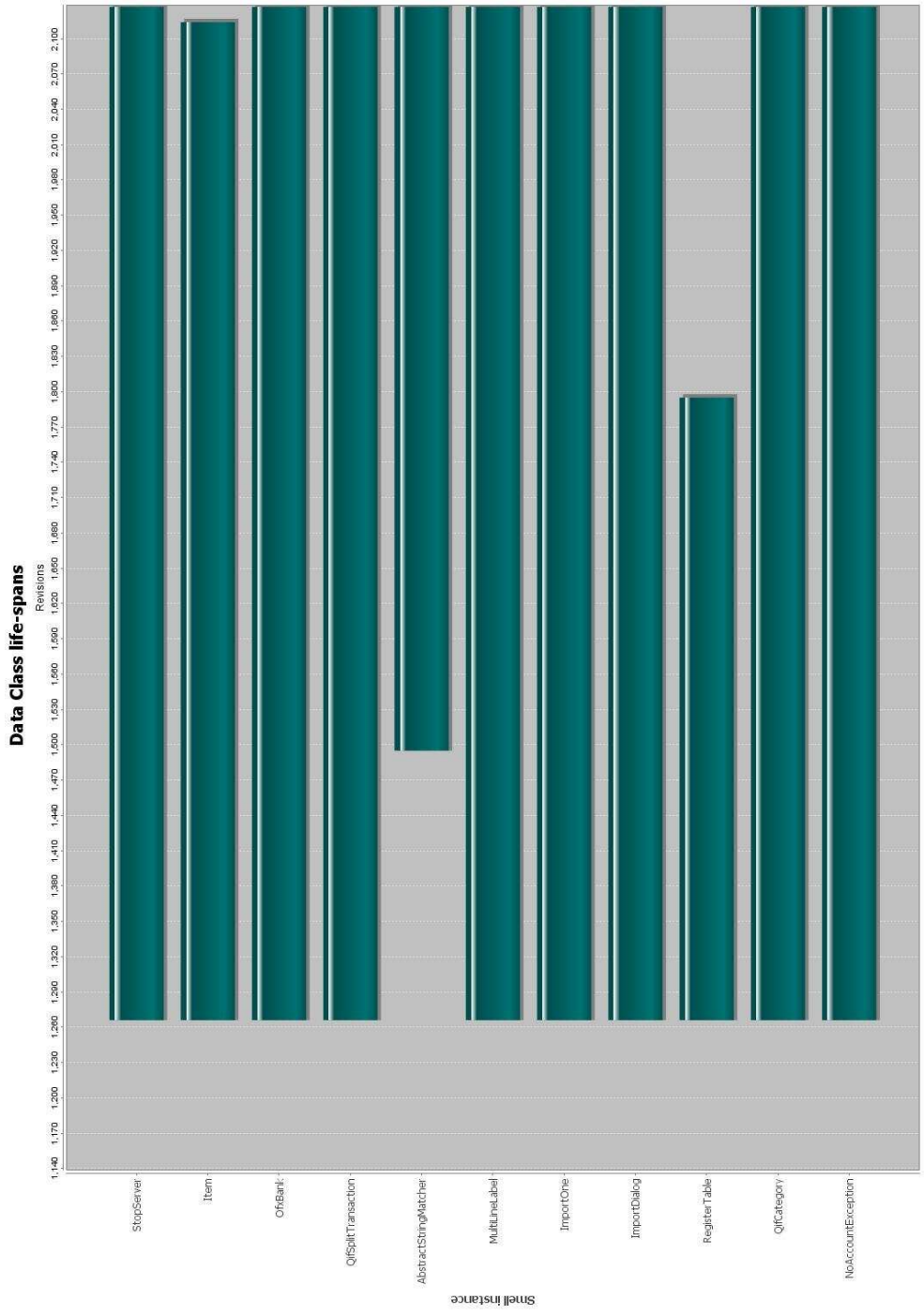


Figure C.18: Lifespans of Data Classes in jGnash.

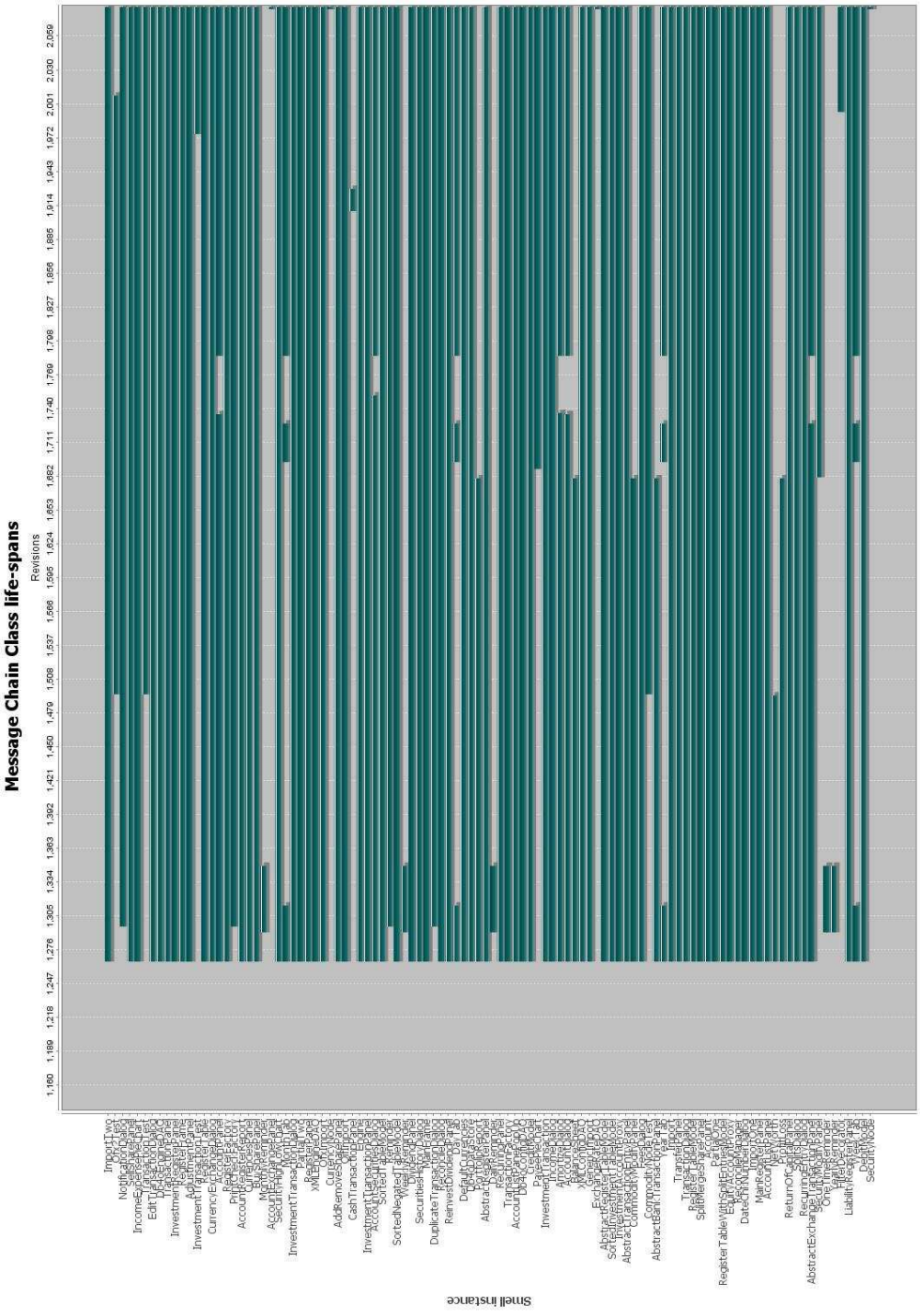


Figure C.19: Lifespans of Message Chain Classes in jGnash.

Figure C.20: Lifespans of Long Parameter List Classes in jGnash.

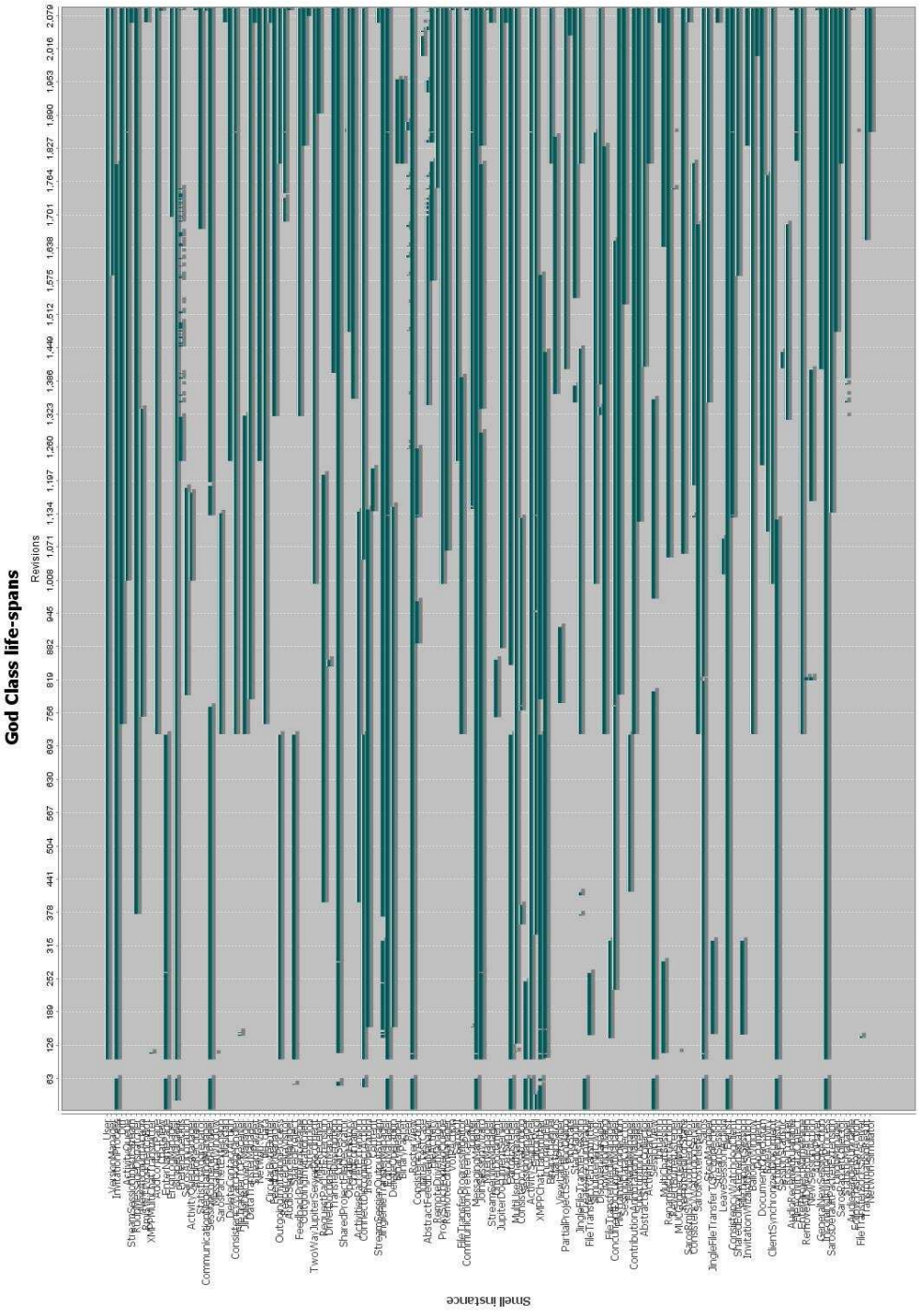


Figure C.21: Lifespans of God Classes in Saros.

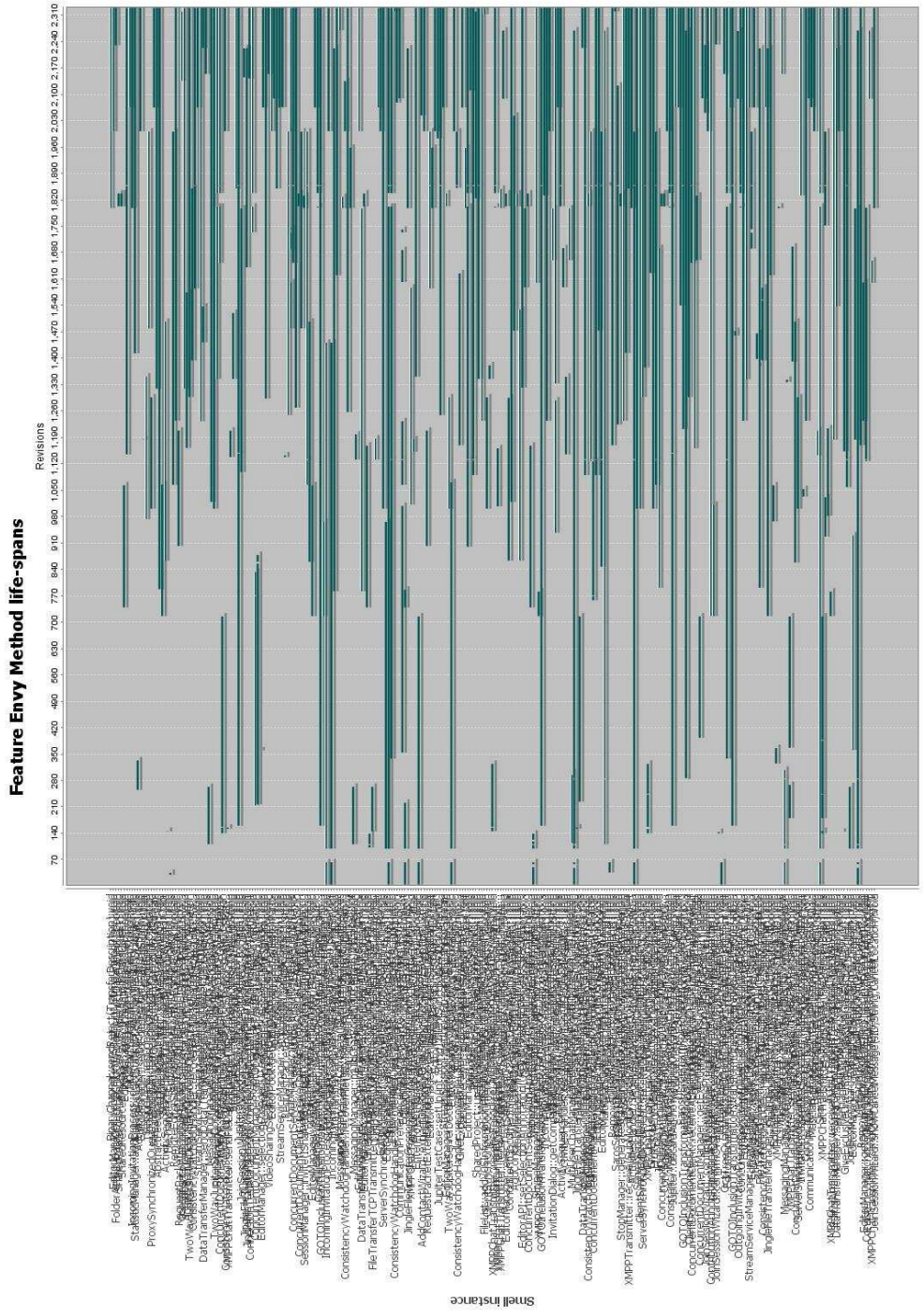


Figure C.22: Lifespans of Feature Envy Methods in Saros.

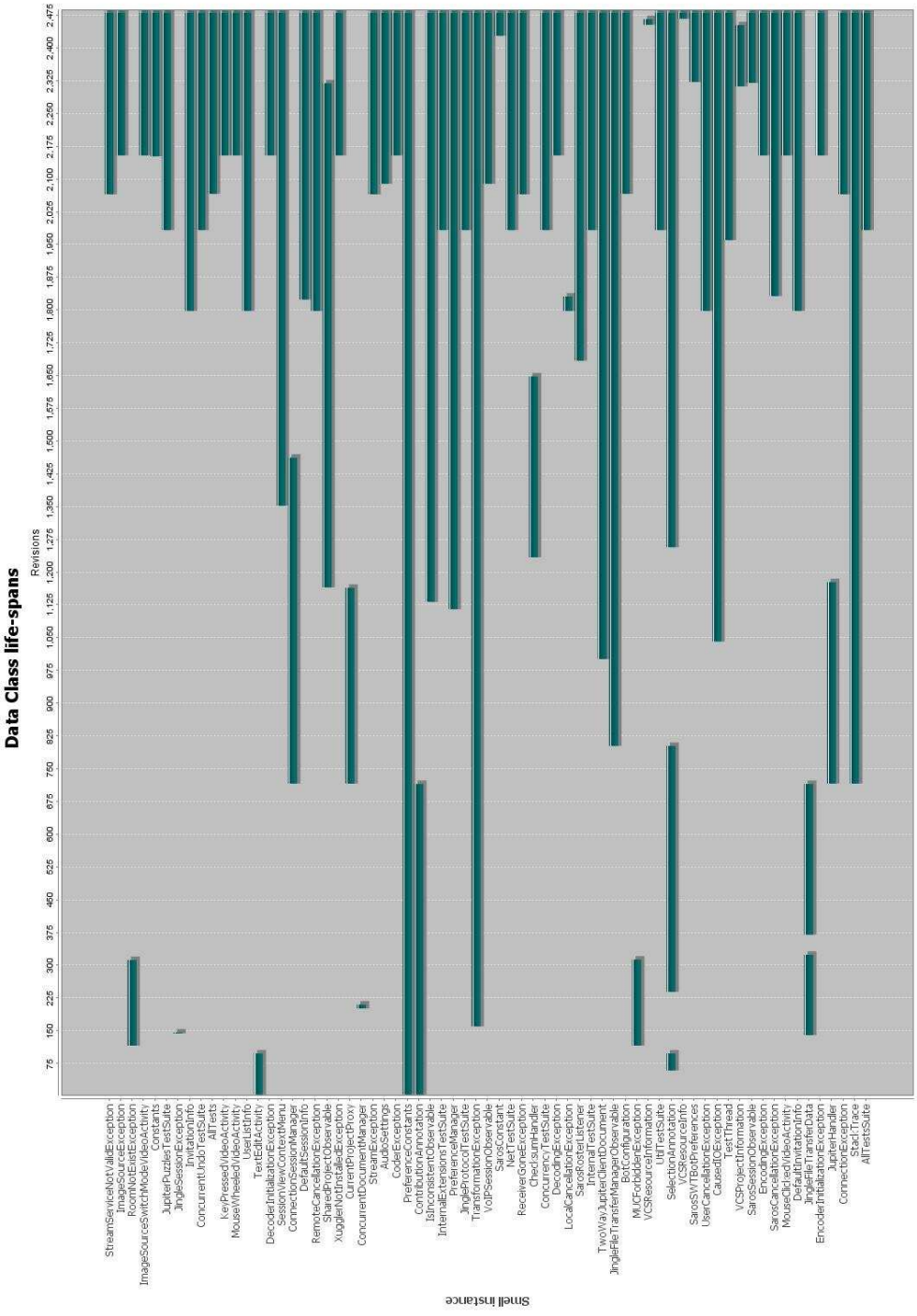


Figure C.23: Lifespans of Data Classes in Saros.

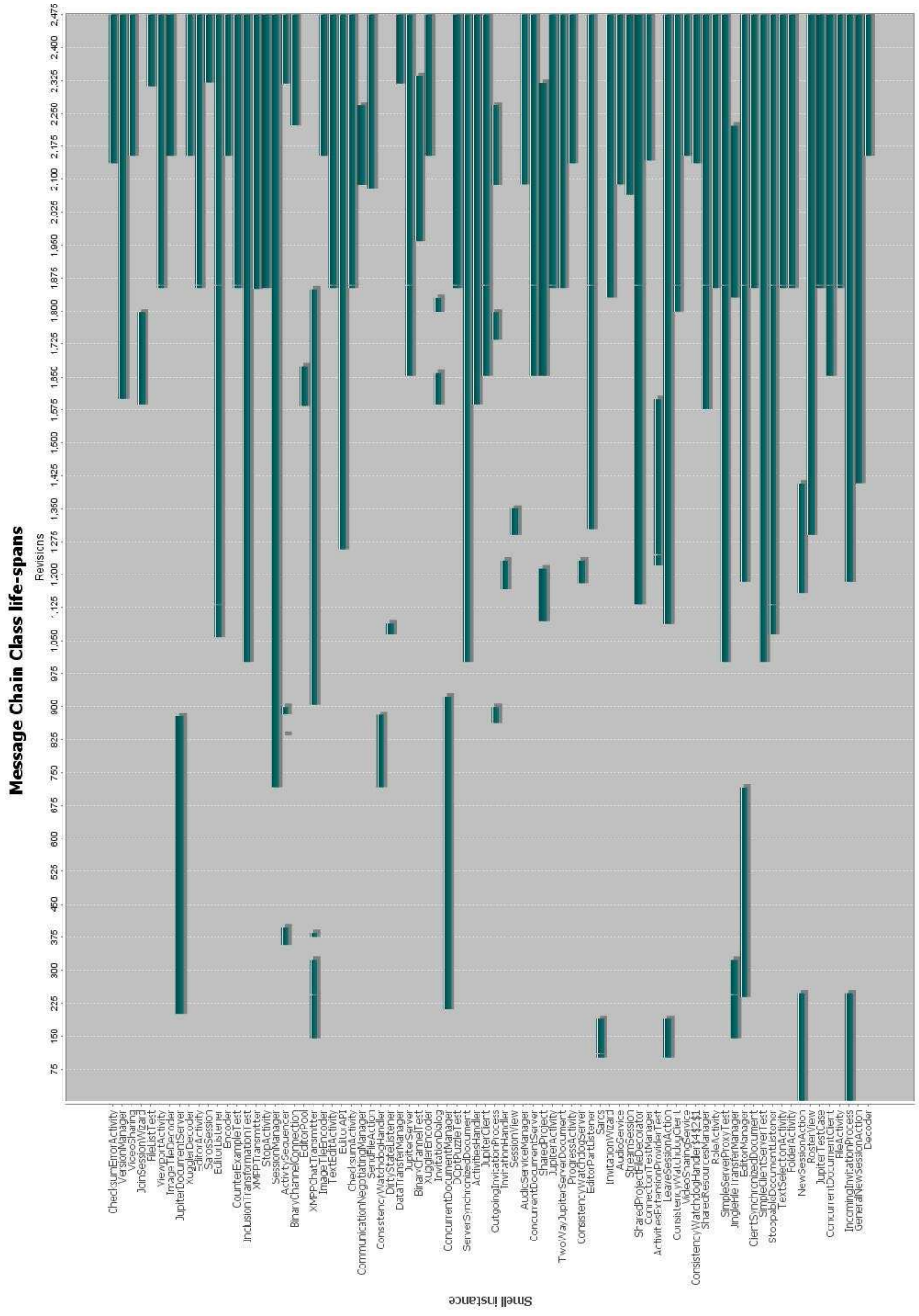


Figure C.24: Lifespans of Message Chain Classes in Saros.

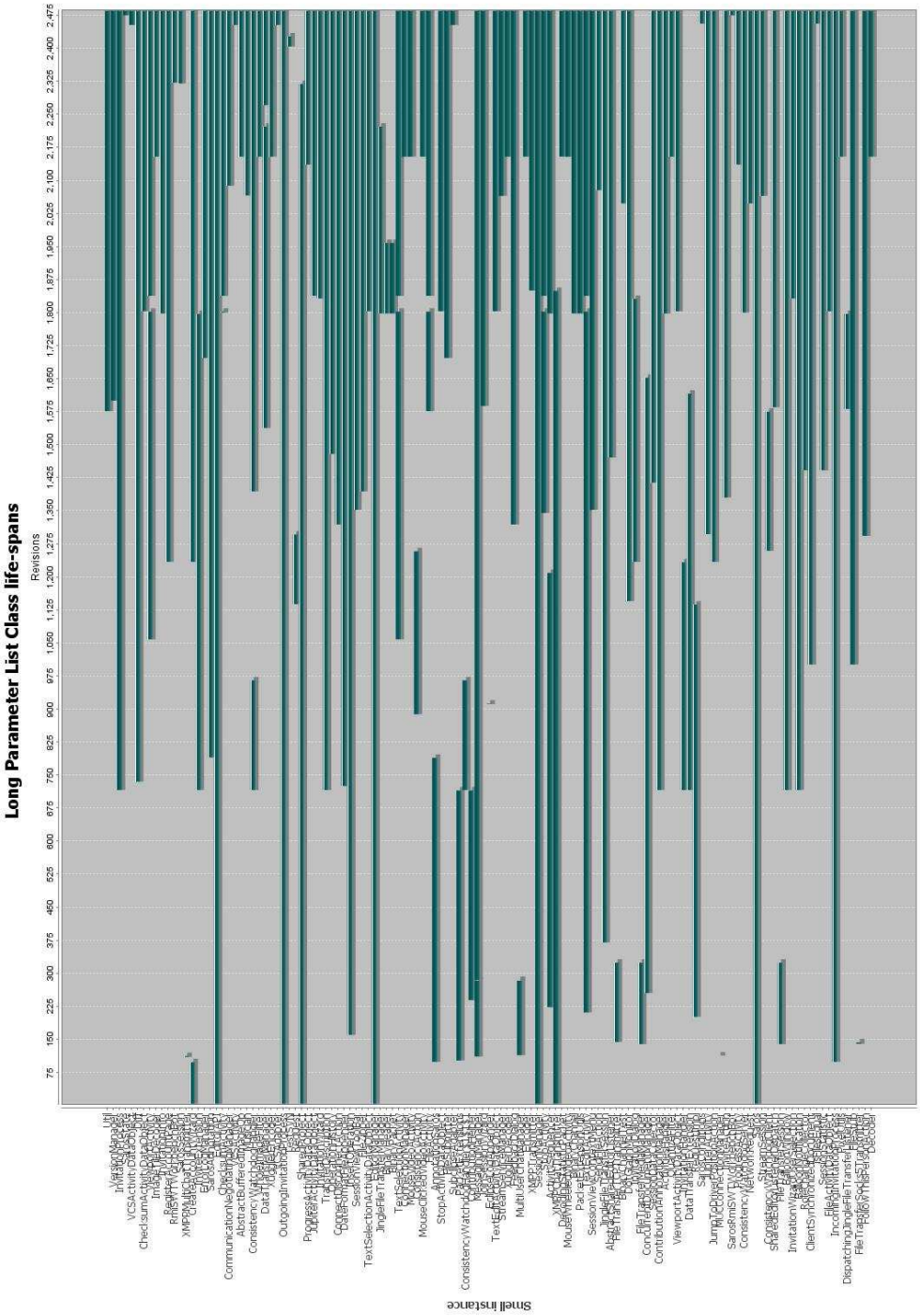


Figure C.25: Lifespans of Long Parameter List Classes in Saros.

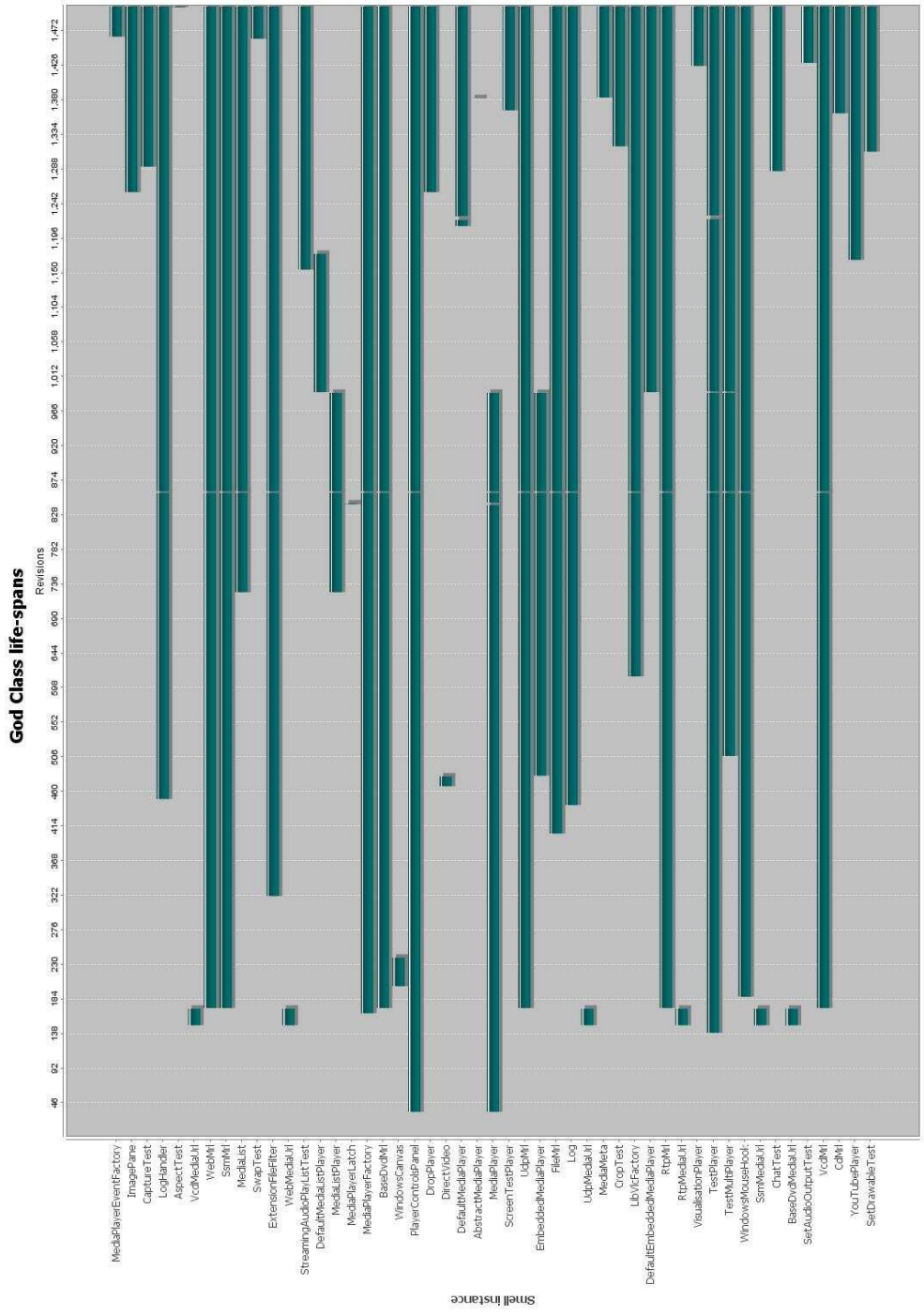


Figure C.26: Lifespans of God Classes in VLCJ.

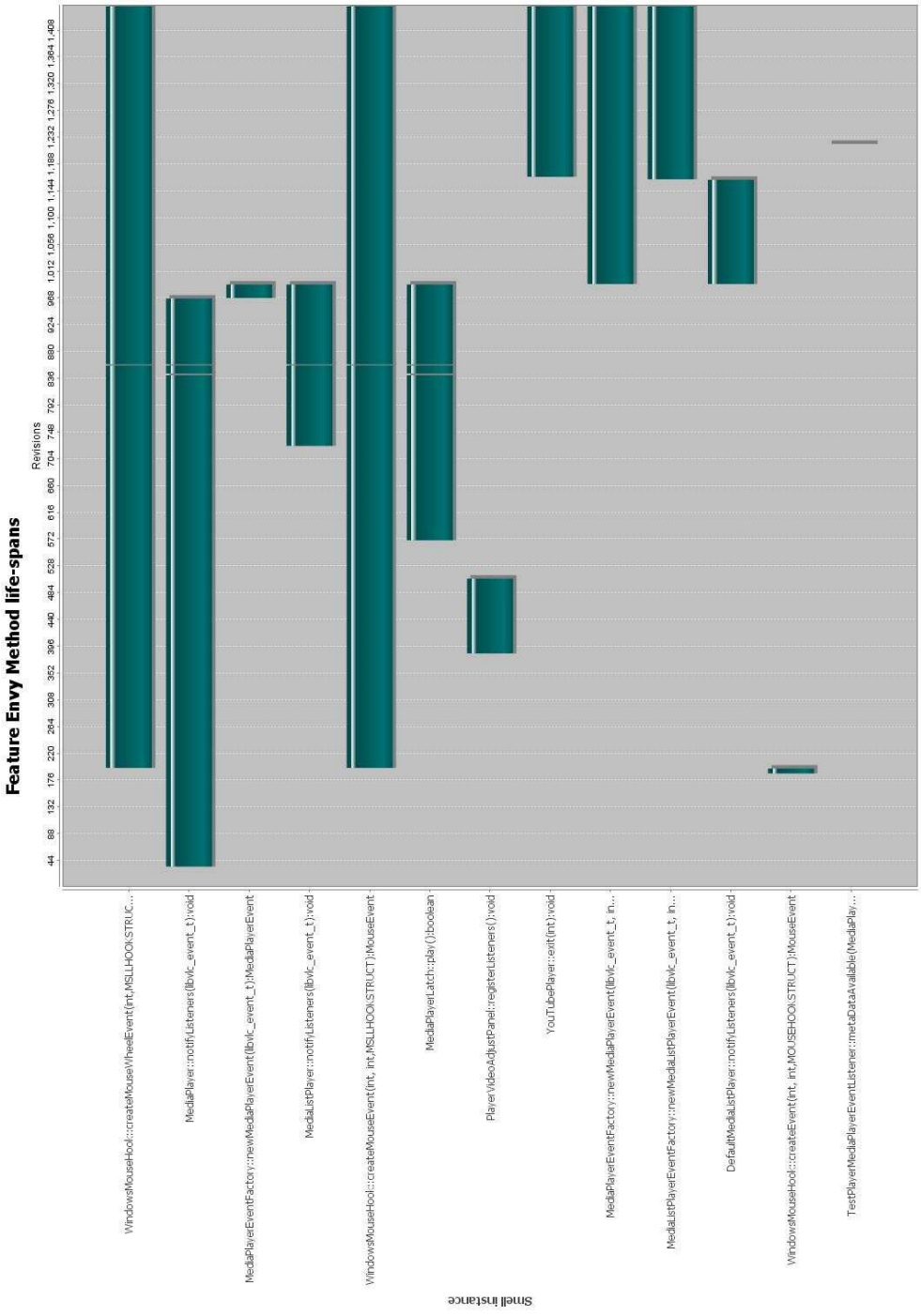


Figure C.27: Lifespans of Feature Envy Methods in VLCJ.

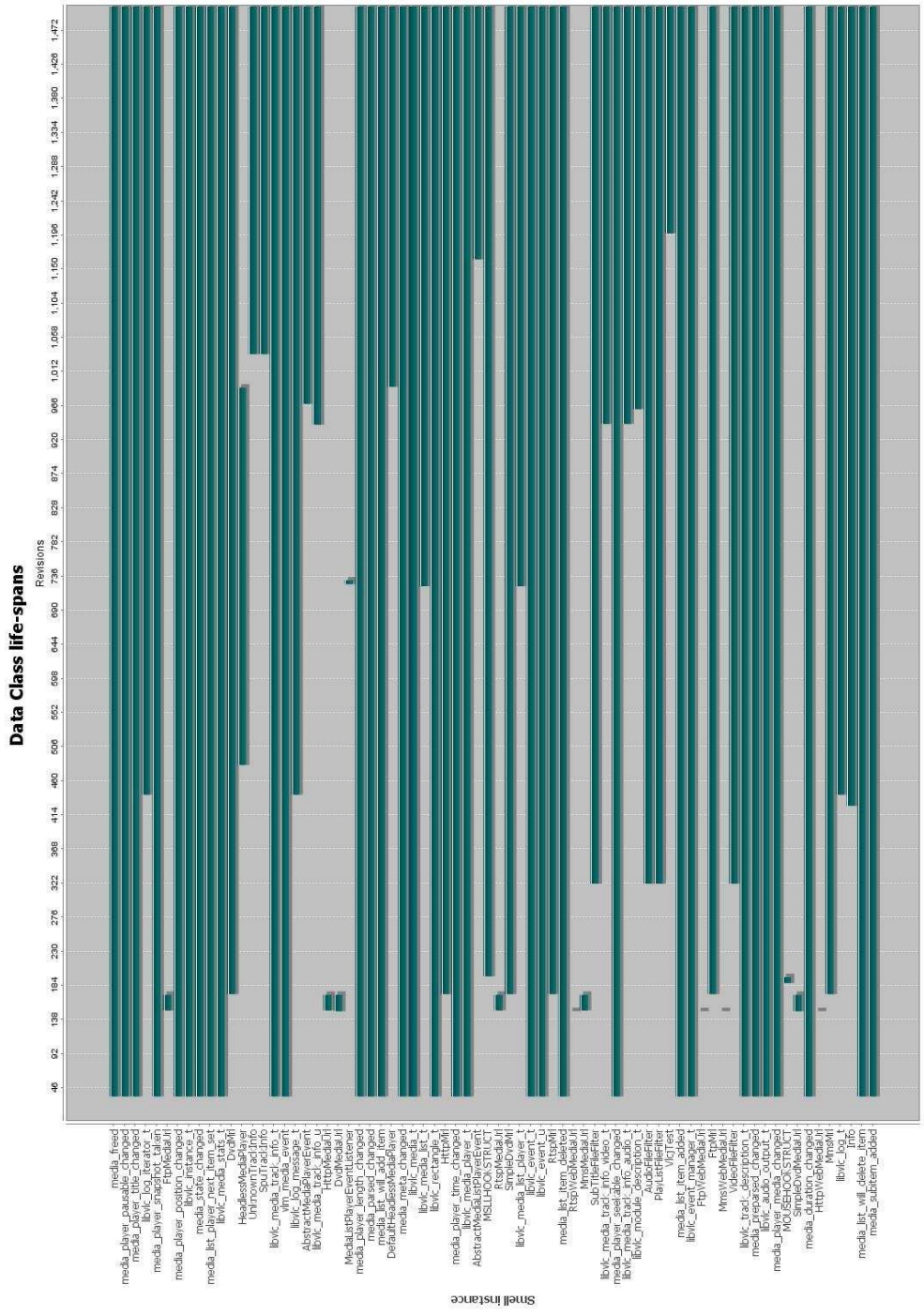


Figure C.28: Lifespans of Data Classes in VLCJ.

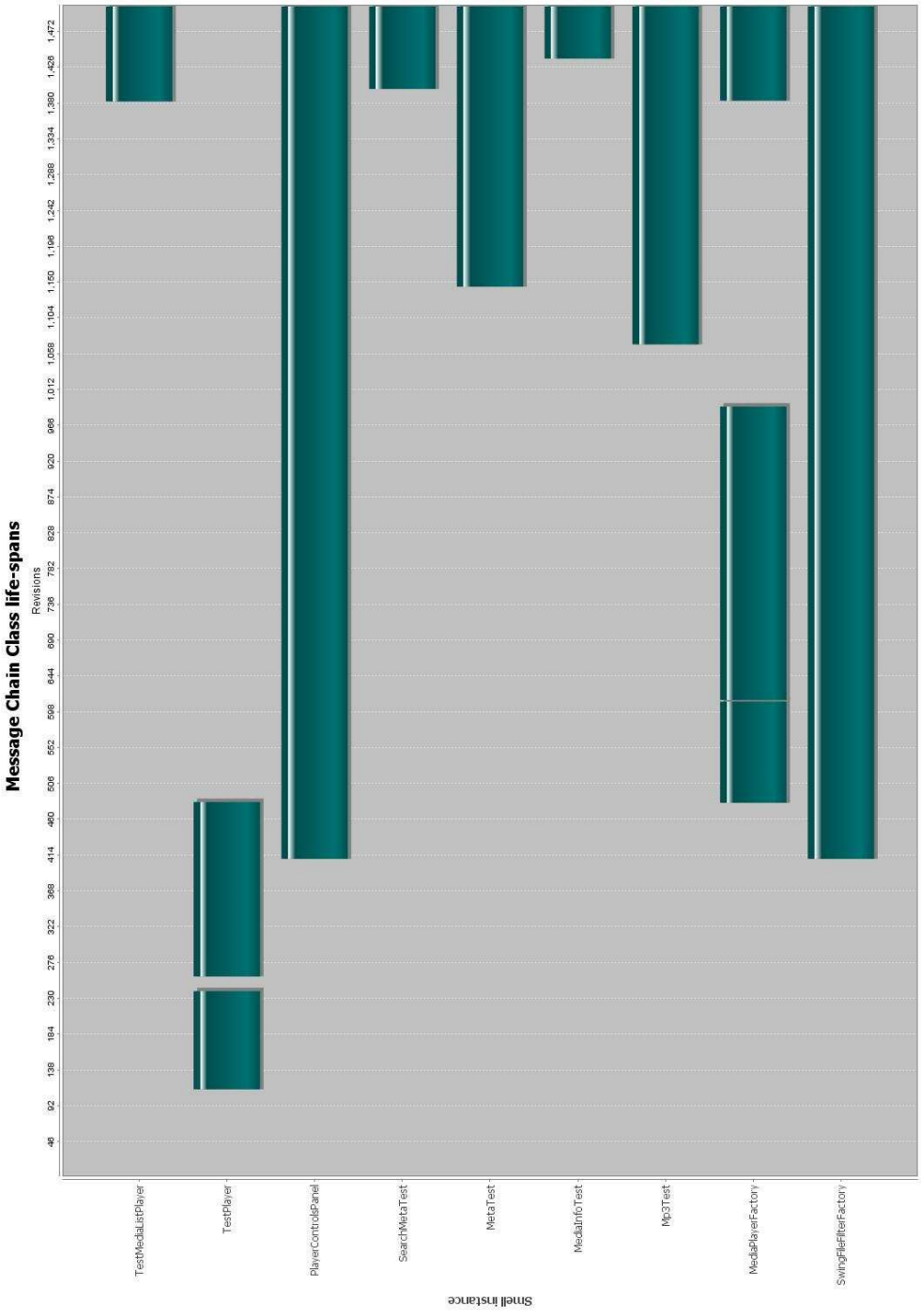


Figure C.29: Lifespans of Message Chain Classes in VLCJ.

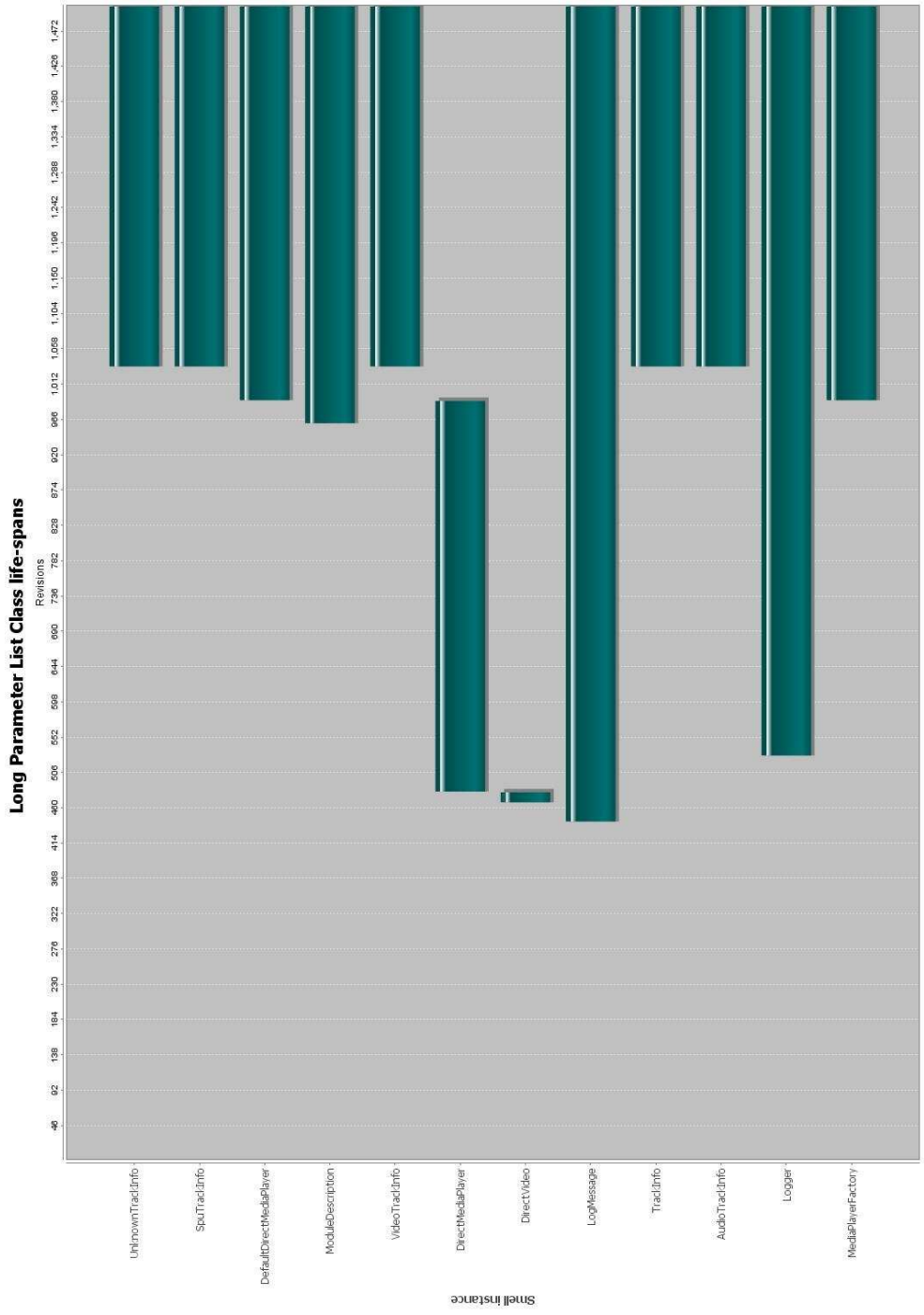


Figure C.30: Lifespans of Long Parameter List Classes in VLCJ.

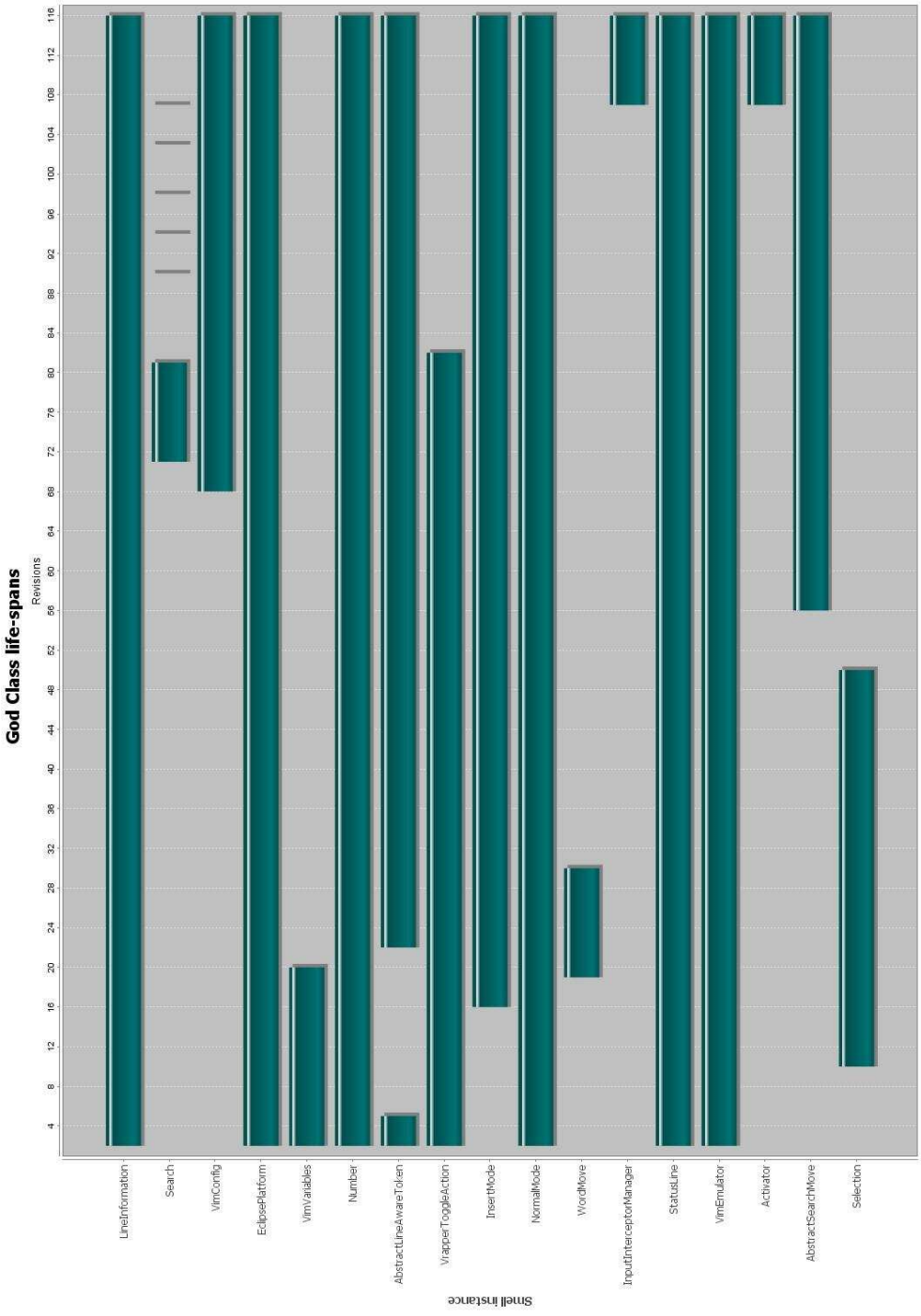


Figure C.31: Lifespans of God Classes in Vrapper (base).

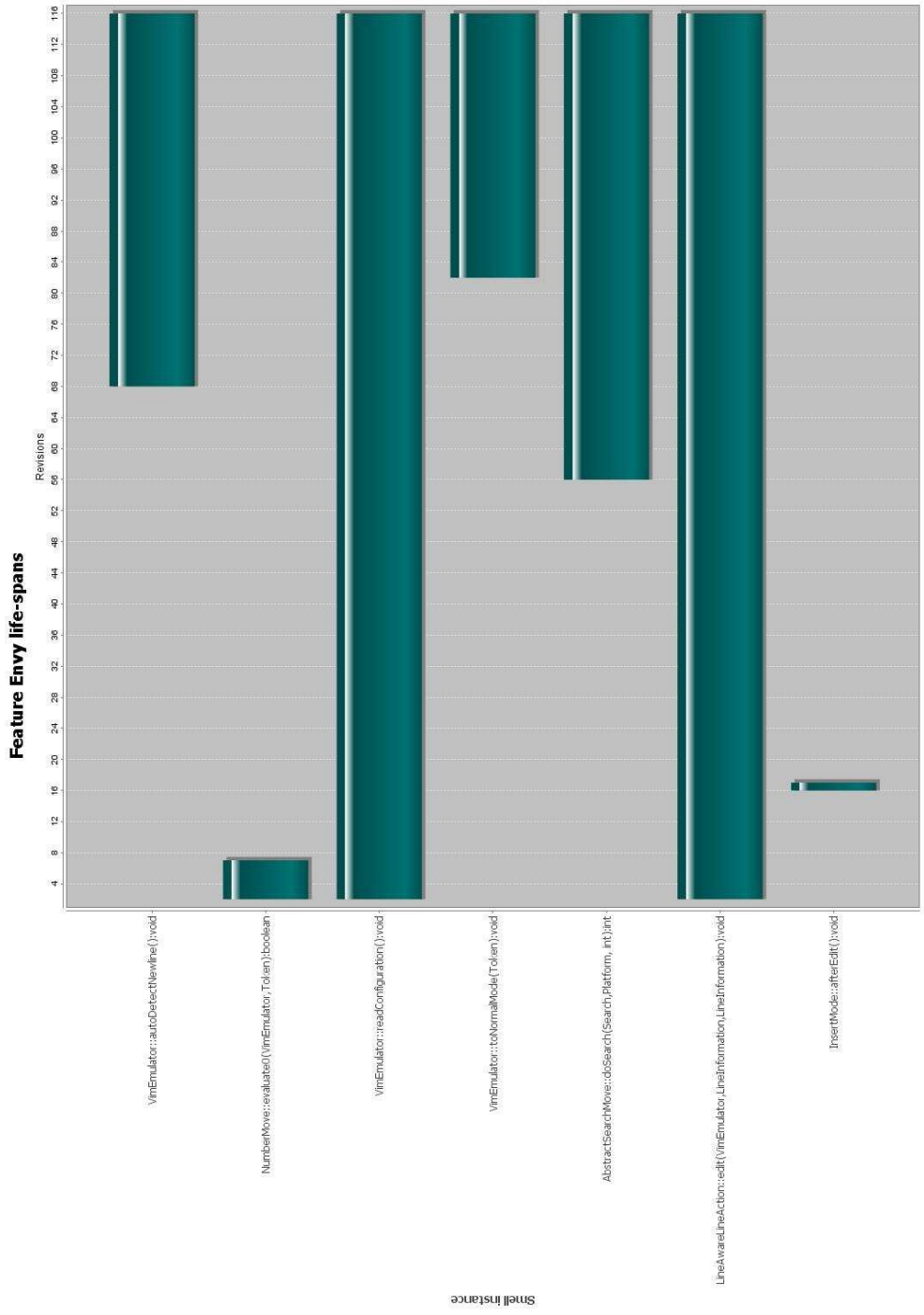


Figure C.32: Lifespans of Feature Envy Methods in Vrapper (base).

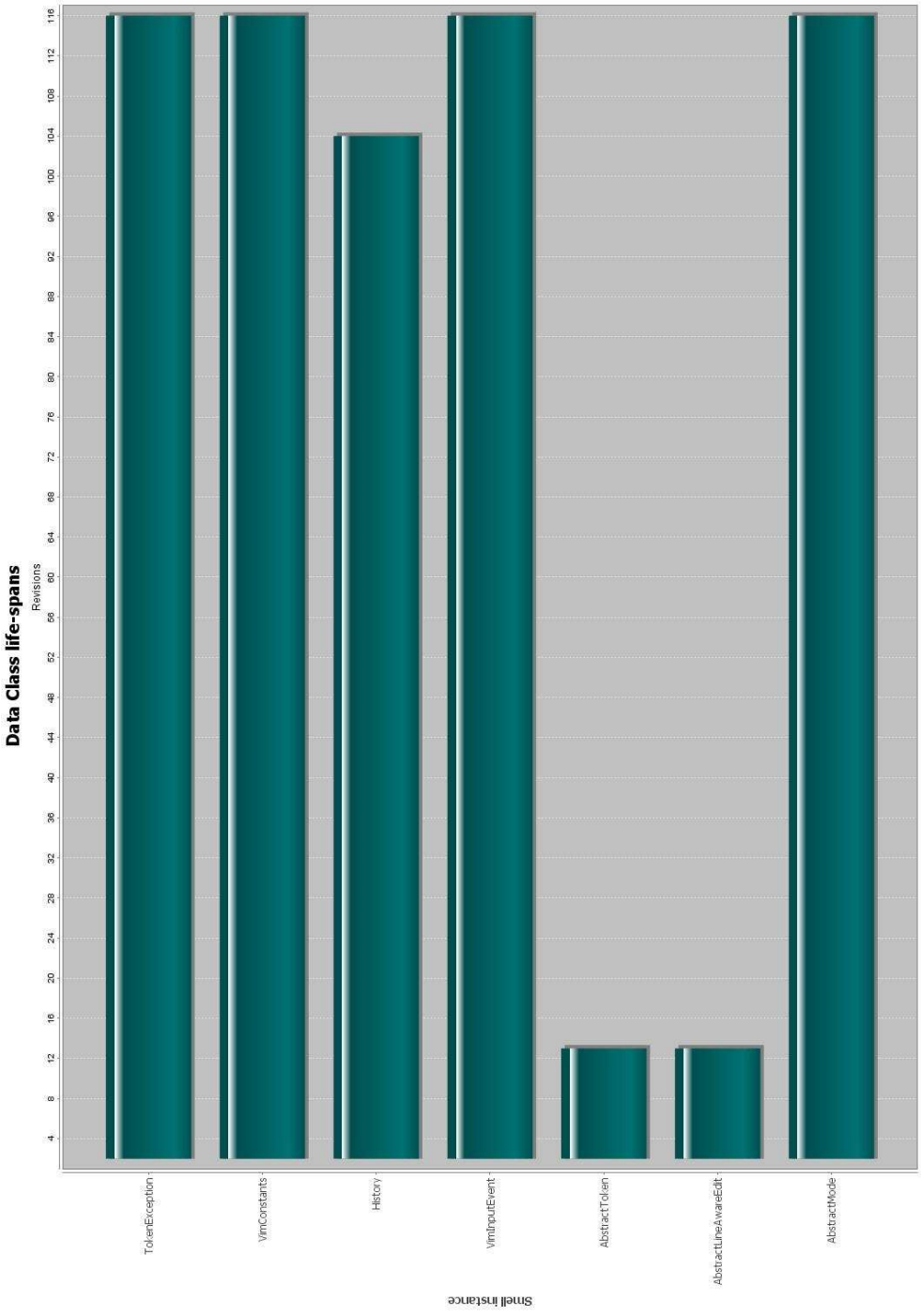


Figure C.33: Lifespans of Data Classes in Vrapper (base).

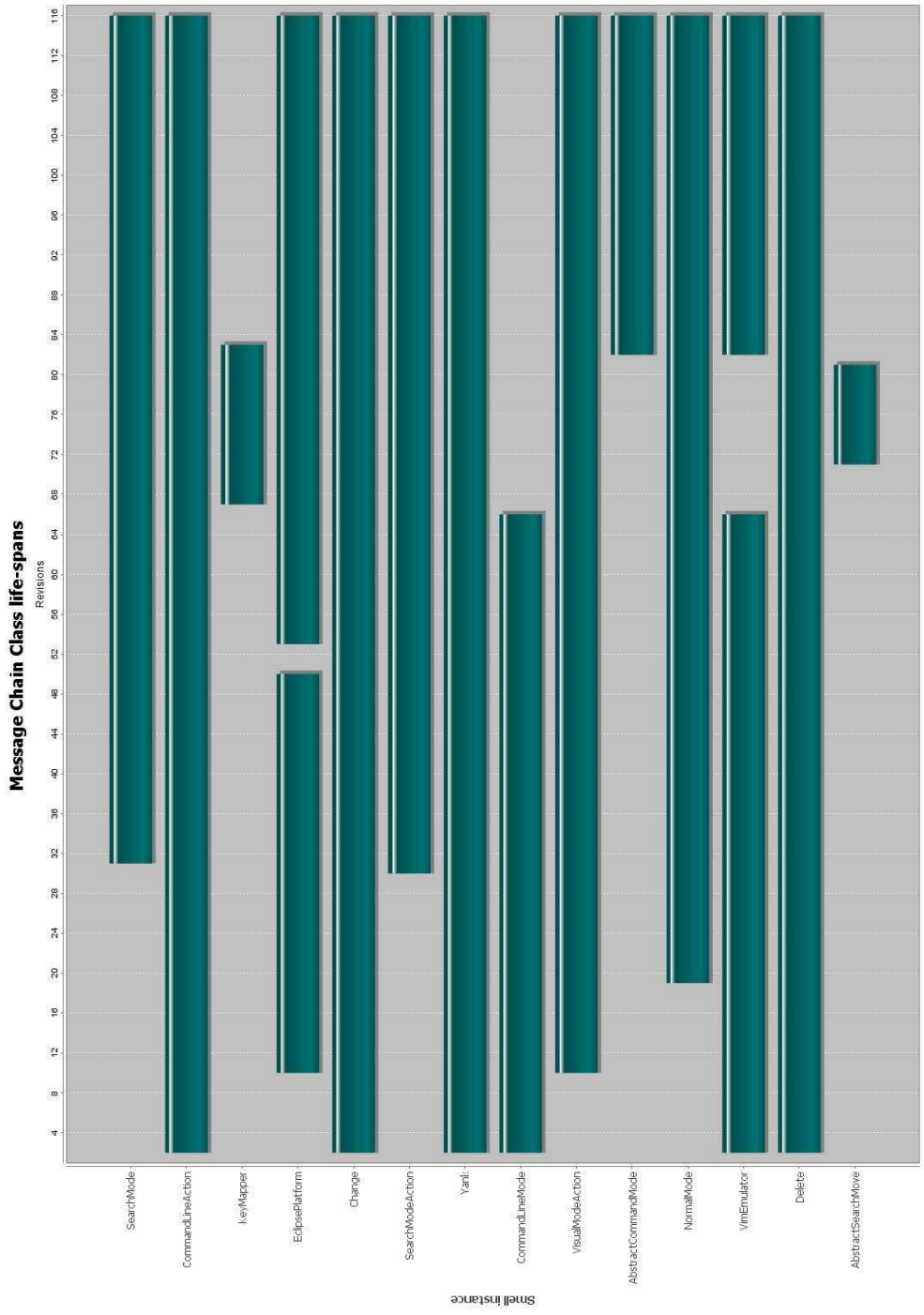


Figure C.34: Lifespans of Message Chain Classes in Vrapper (base).

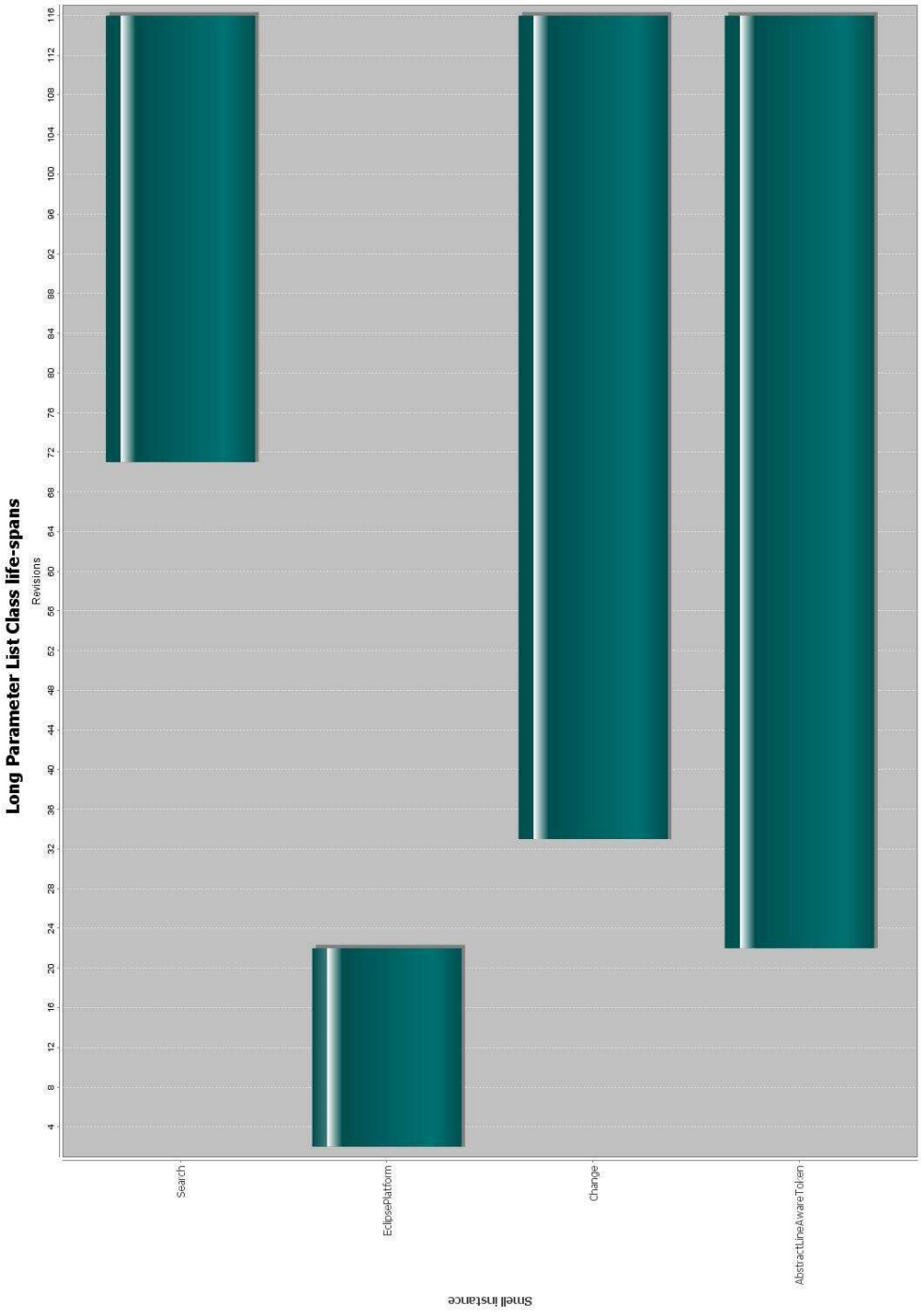


Figure C.35: Lifespans of Long Parameter List Classes in Vrapper (base).

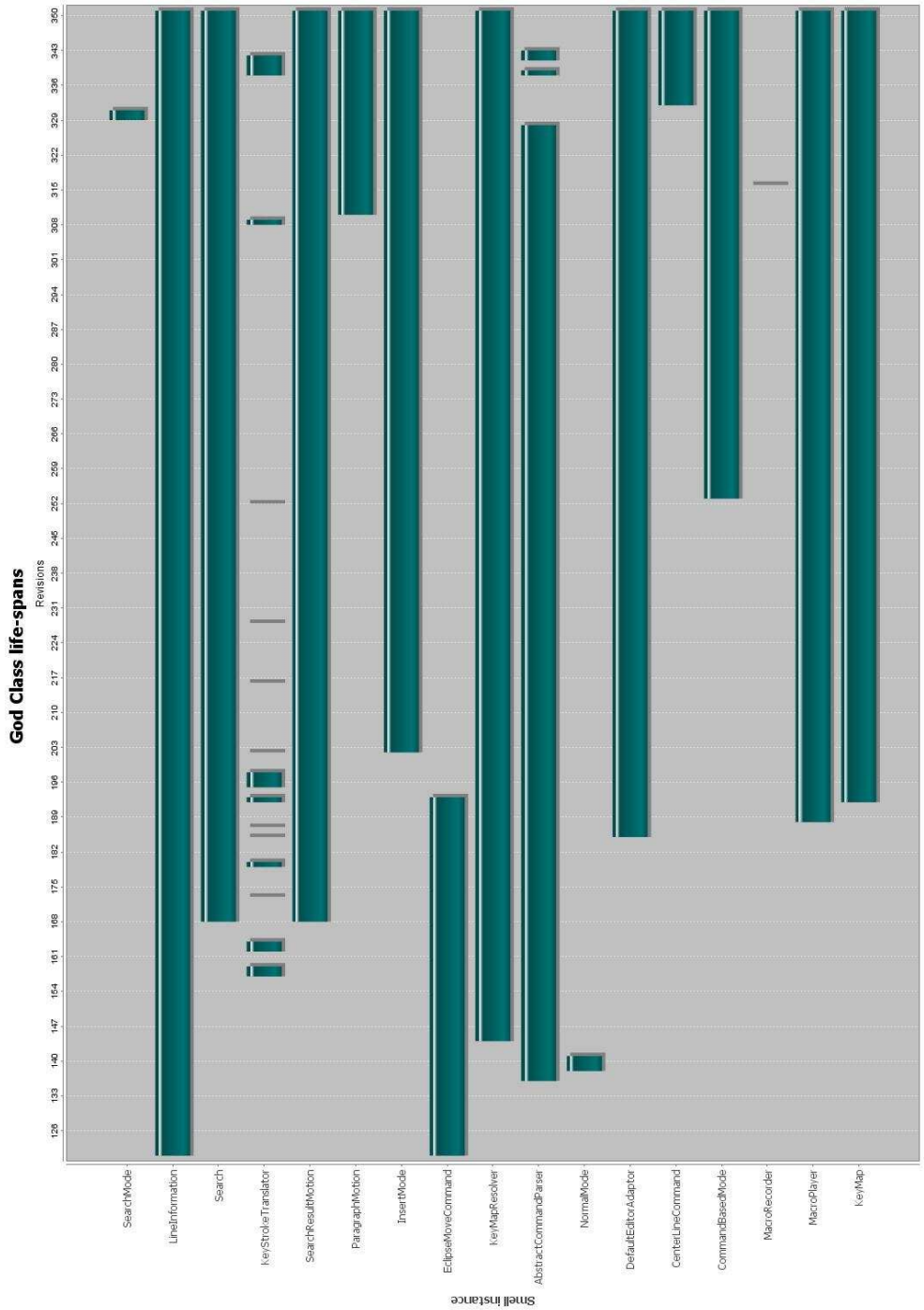


Figure C.36: Lifespans of God Classes in Vrapper (core).

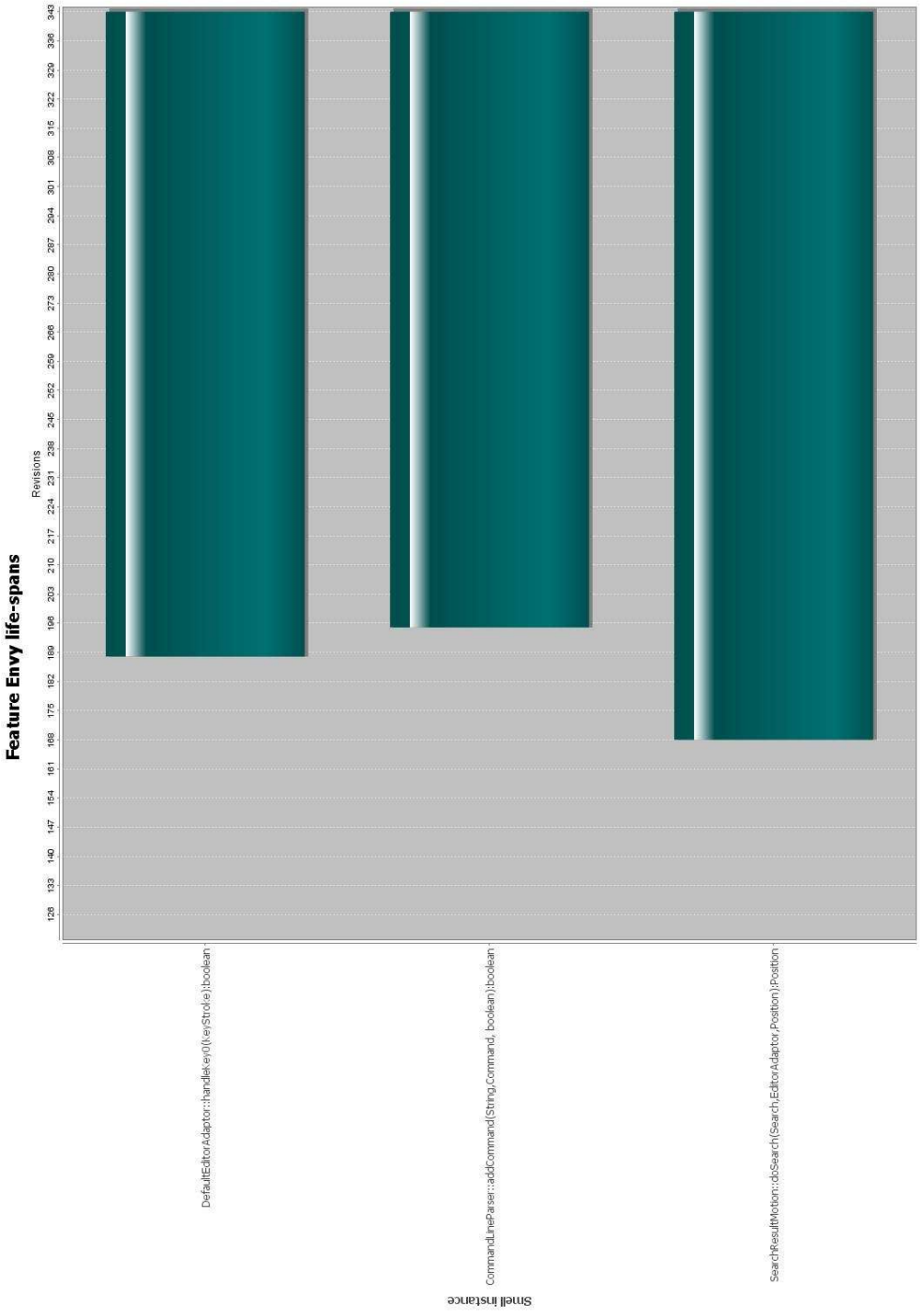


Figure C.37: Lifespans of Feature Envy Methods in Vrapper (core).

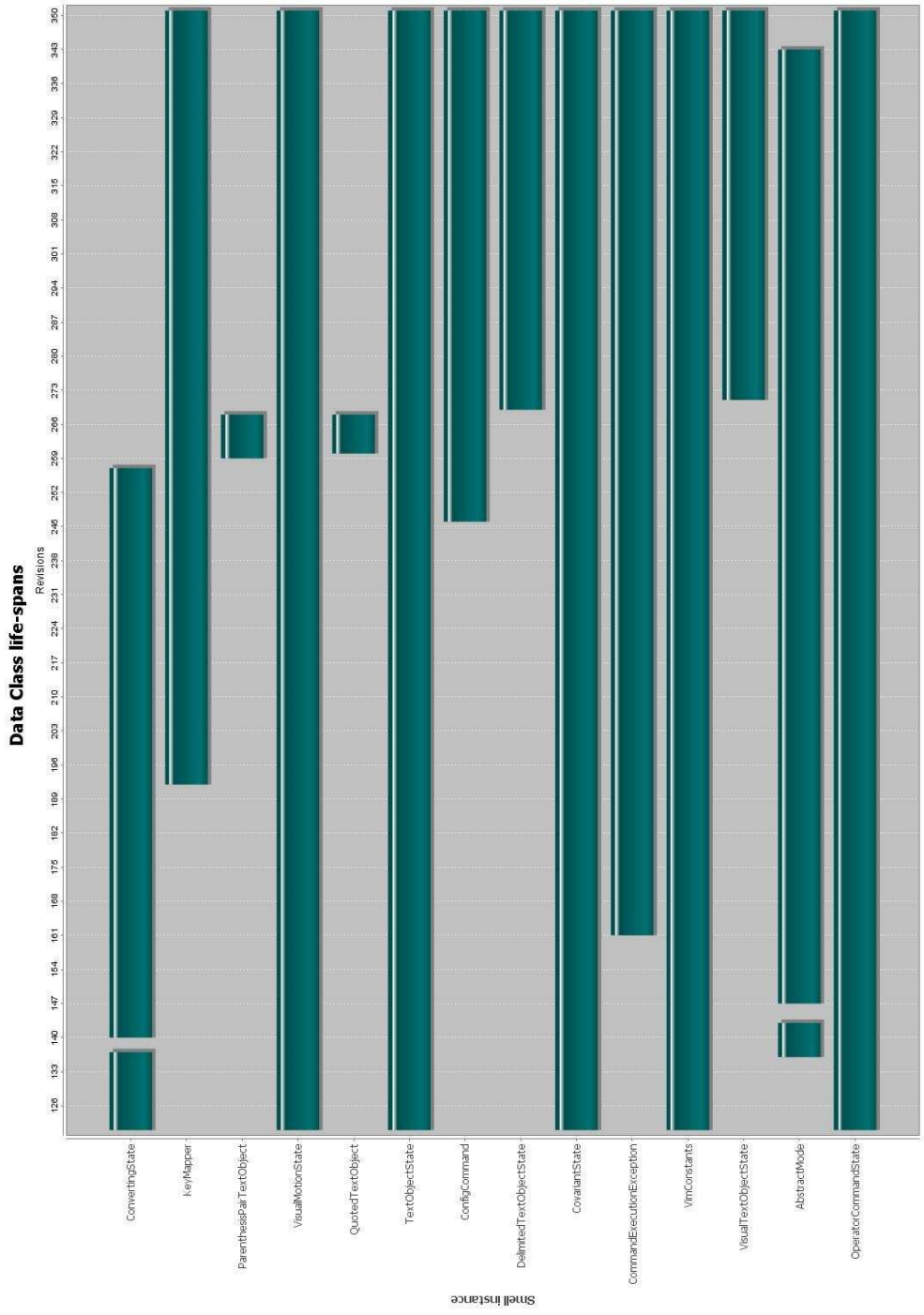


Figure C.38: Lifespans of Data Classes in Vrapper (core).

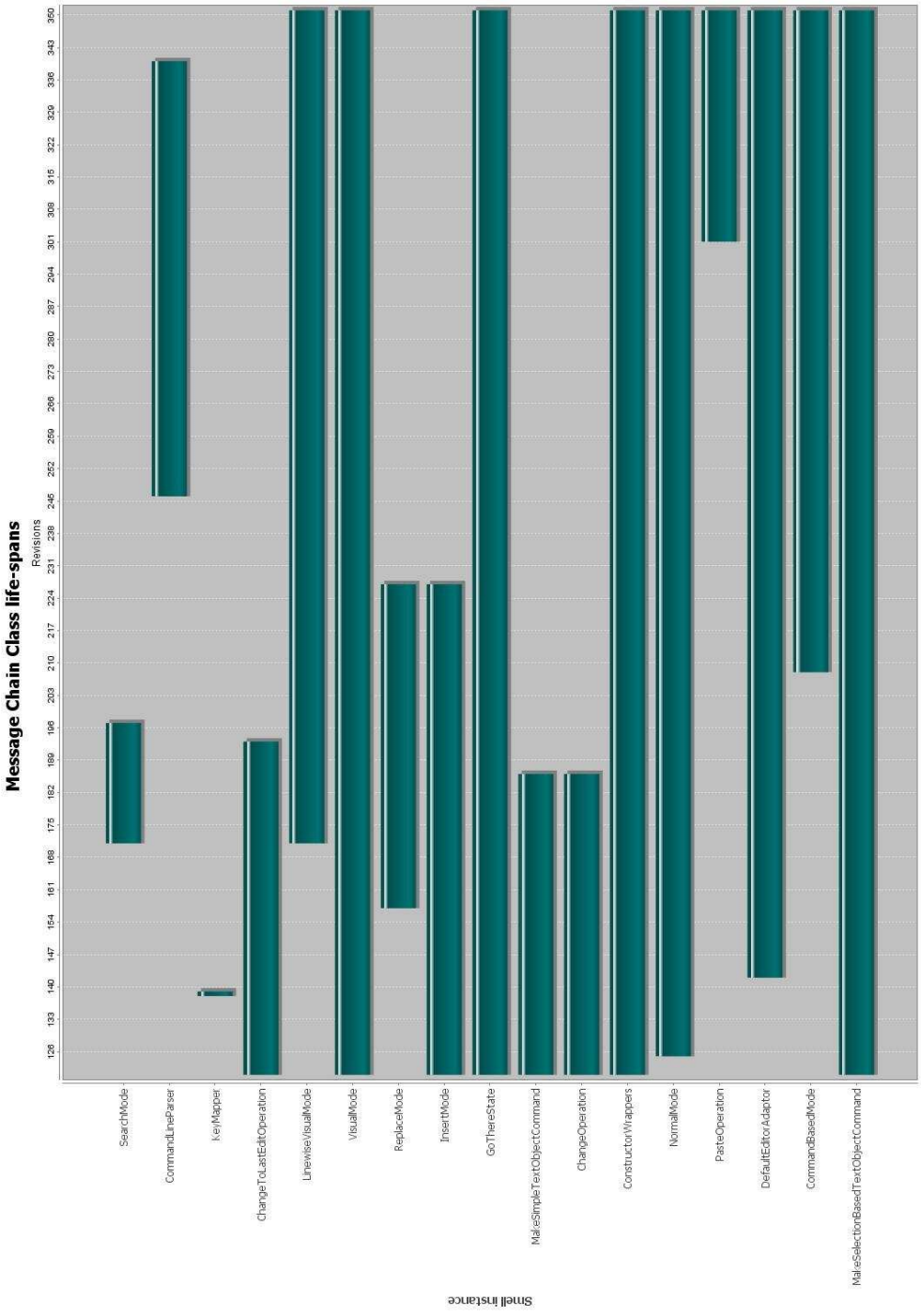


Figure C.39: Lifespans of Message Chain Classes in Vrapper (core).

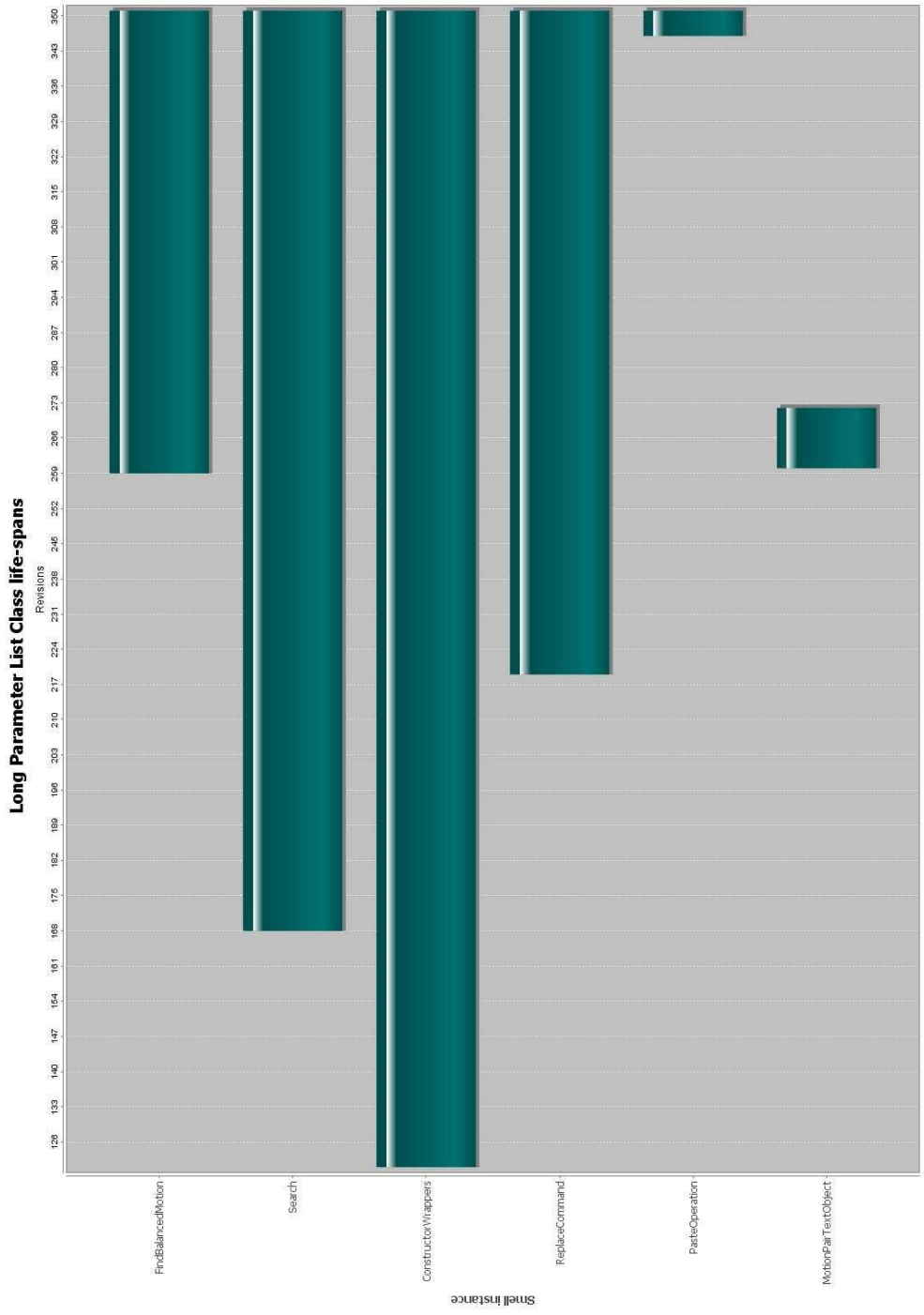


Figure C.40: Lifespans of Long Parameter List Classes in Vrapper (core).

Appendix D

Boxplots

The boxplots displayed below represent the distribution of individual lifespans of code smell instances per subject system over a range of revisions.

D.1 Overall Lifespans (RQ1)

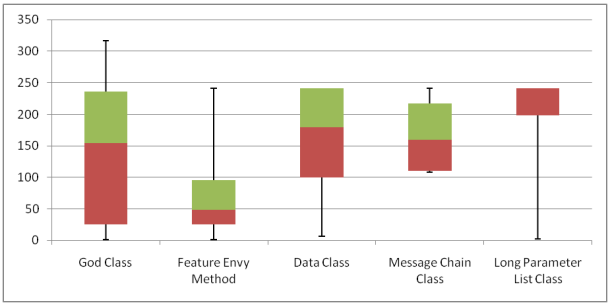


Figure D.1: Boxplot of lifespan distribution in CalDAV4j.

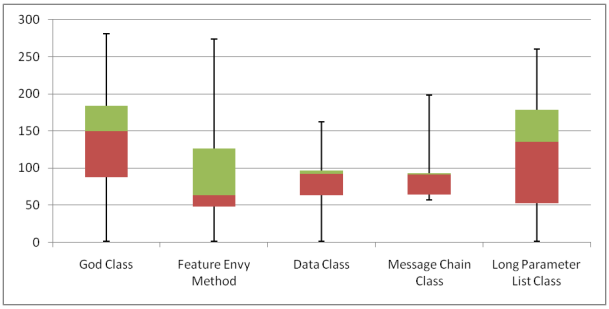


Figure D.2: Boxplot of lifespan distribution in Evolution Chamber.

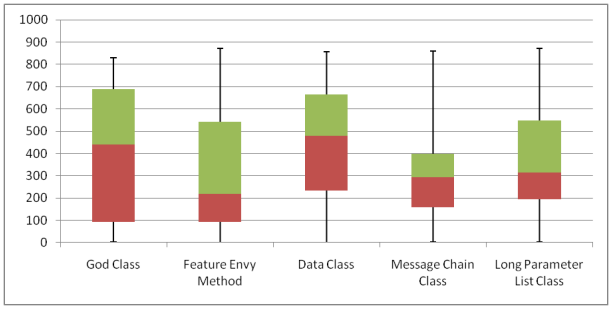


Figure D.3: Boxplot of lifespan distribution in JDiveLog.

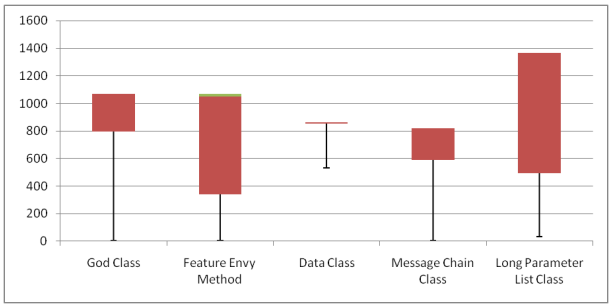


Figure D.4: Boxplot of lifespan distribution in jGnash.

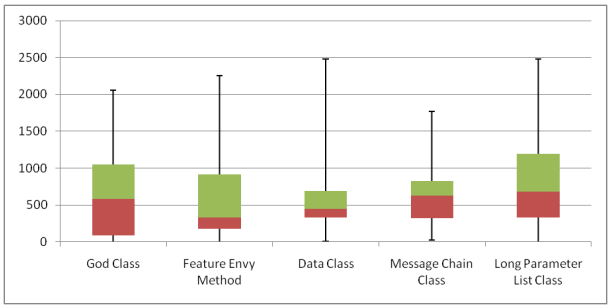


Figure D.5: Boxplot of lifespan distribution in Saros.

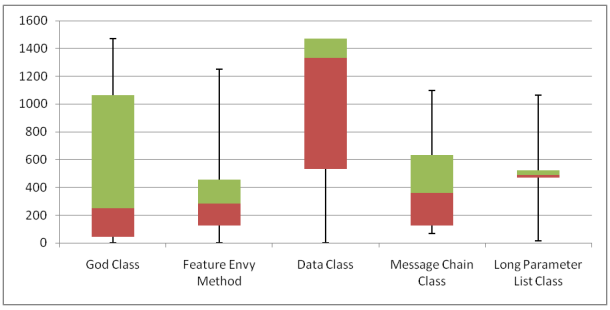


Figure D.6: Boxplot of lifespan distribution in VLCJ.

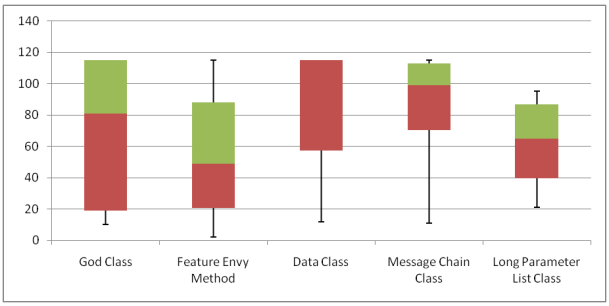


Figure D.7: Boxplot of lifespan distribution in Vrapper (base).

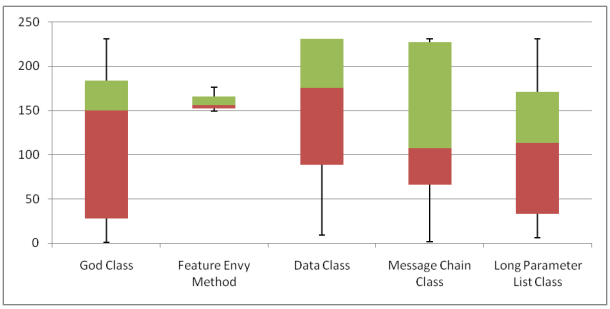


Figure D.8: Boxplot of lifespan distribution in Vrapper (core).

D.2 20% Lifespans (RQ2)

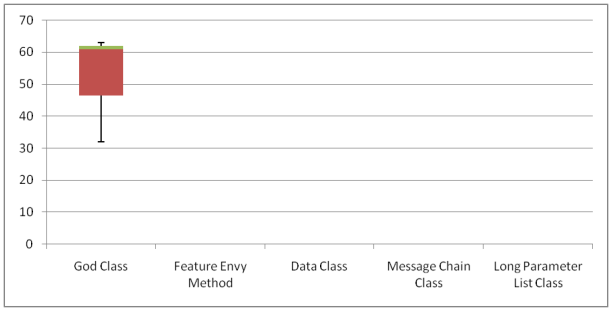


Figure D.9: Boxplot of lifespan distribution in the first 20% of CalDAV4j.

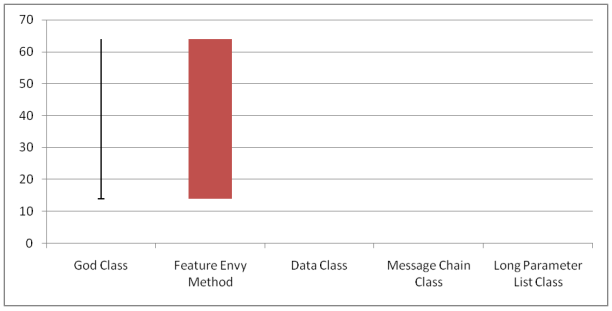


Figure D.10: Boxplot of lifespan distribution in the last 20% of CalDAV4j.

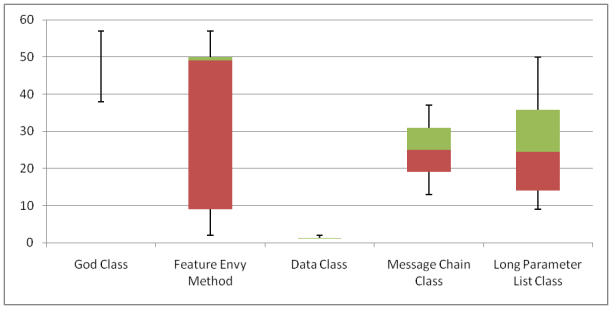


Figure D.11: Boxplot of lifespan distribution in the first 20% of Evolution Chamber.

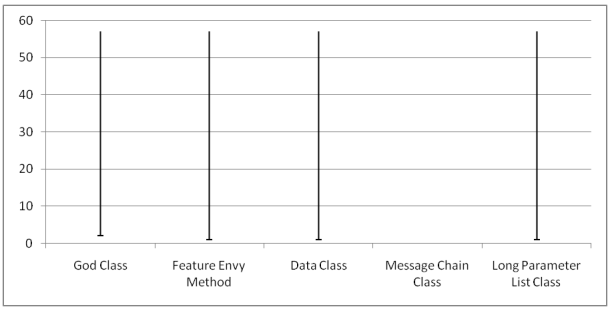


Figure D.12: Boxplot of lifespan distribution in the last 20% of Evolution Chamber.

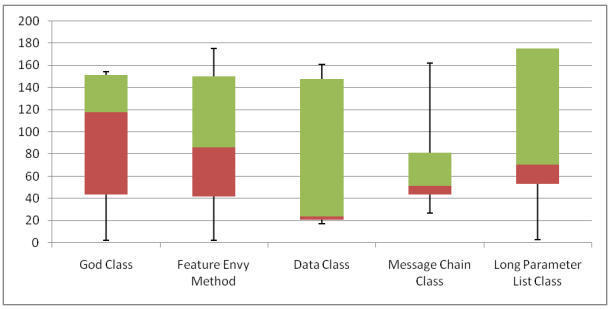


Figure D.13: Boxplot of lifespan distribution in the first 20% of JDiveLog.

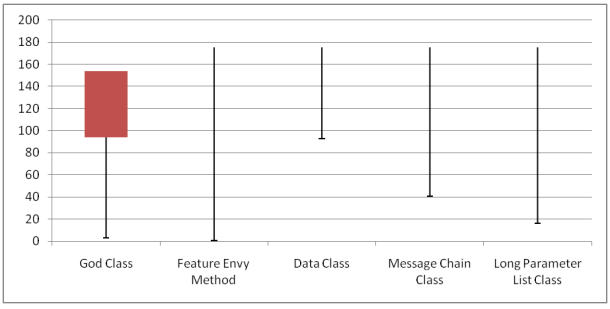


Figure D.14: Boxplot of lifespan distribution in the last 20% of JDiveLog.

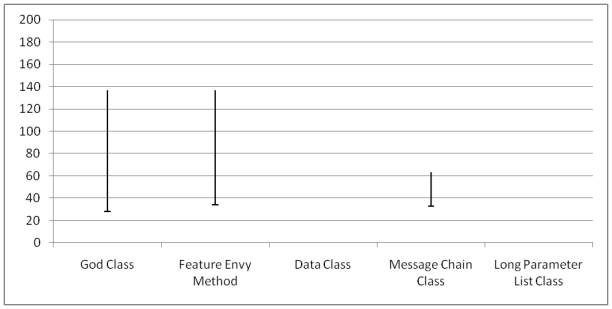


Figure D.15: Boxplot of lifespan distribution in the first 20% of jGnash.

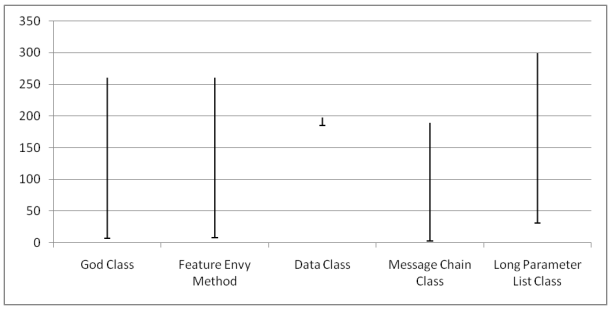


Figure D.16: Boxplot of lifespan distribution in the last 20% of jGnash.

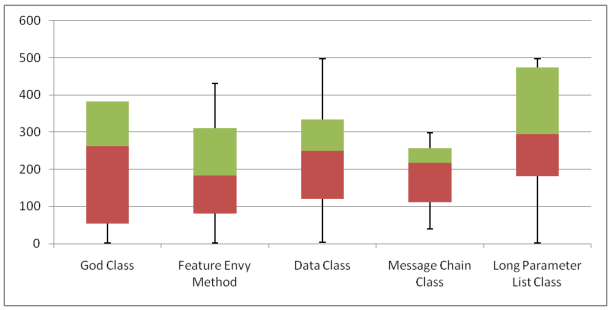


Figure D.17: Boxplot of lifespan distribution in the first 20% of Saros.

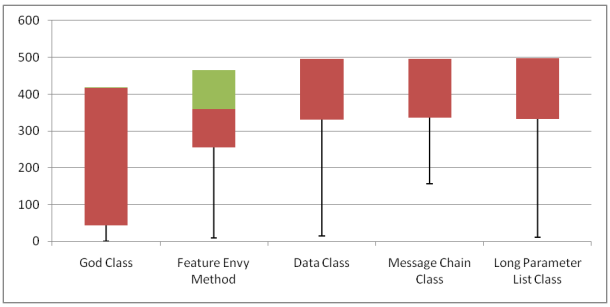


Figure D.18: Boxplot of lifespan distribution in the last 20% of Saros.

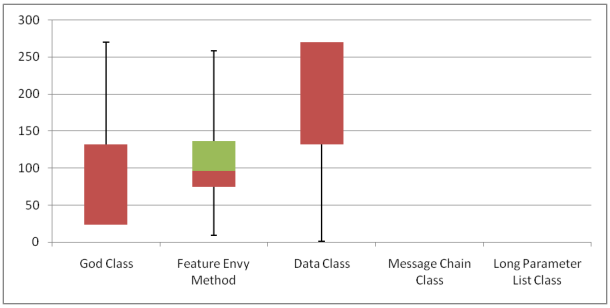


Figure D.19: Boxplot of lifespan distribution in the first 20% of VLCJ.

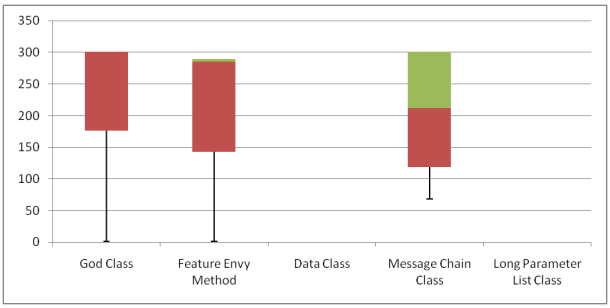


Figure D.20: Boxplot of lifespan distribution in the last 20% of VLCJ.

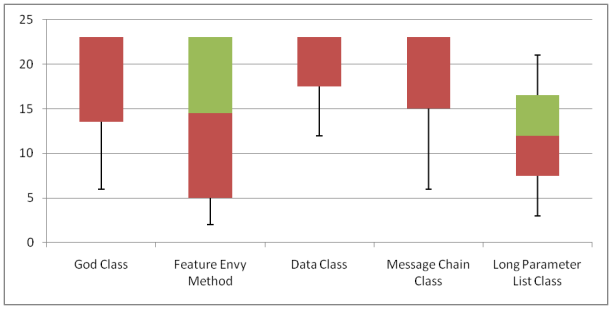


Figure D.21: Boxplot of lifespan distribution in the first 20% of Vrapper (base).

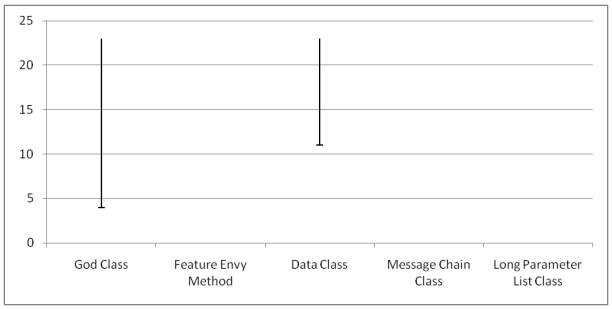


Figure D.22: Boxplot of lifespan distribution in the last 20% of Vrapper (base).

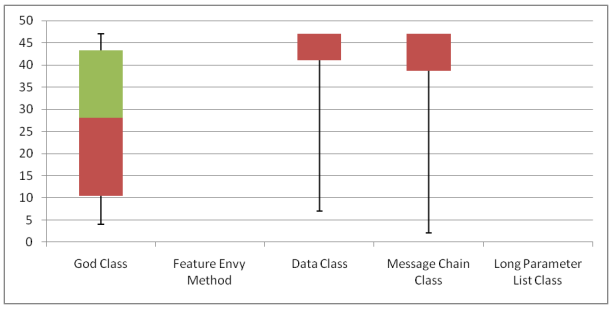


Figure D.23: Boxplot of lifespan distribution in the first 20% of Vrapper (core).

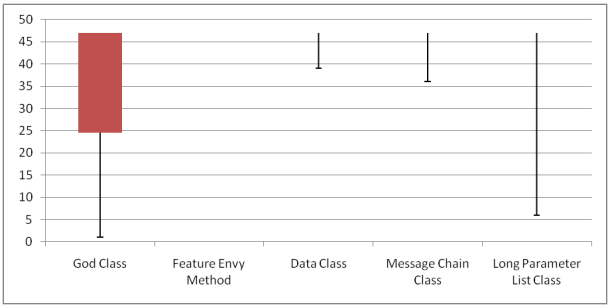


Figure D.24: Boxplot of lifespan distribution in the last 20% of Vrapper (core).