INTERNATIONAL HELLENIC UNIVERSITY

Evolution Analysis of Technical Debt: Monitoring the Breaking Point

Alexandros Michailidis

SID: 3306160006

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Mobile and Web computing

DECEMBER 2017

THESSALONIKI – GREECE

INTERNATIONAL HELLENIC UNIVERSITY

Evolution Analysis of Technical Debt: Monitoring the Breaking Point

Alexandros Michailidis

SID: 3306160006

Supervisor:	Dr. Apostolos Ampatzoglou
Supervising Committee Members:	Assoc. Prof. Name Surname
	Assist. Prof. Name Surname

SCHOOL OF SCIENCE & TECHNOLOGY

A thesis submitted for the degree of

Master of Science (MSc) in Mobile and Web computing

DECEMBER 2017

THESSALONIKI – GREECE

Abstract

Technical Debt (TD) is a metaphor to financial debt, used both by management and technical stakeholders, which denotes that the development of poor code quality software can be perceived as a debt, that will be repaid in the future as higher maintenance effort. In this paper we build upon the FITTED framework, which proposes a methodology for monitoring the point, that the company has spent the whole amount of funds on maintenance activities, the breaking point. To achieve this goal, we have created a desktop application, which incarnates the FITTED theory, and illustrate its applicability through a case study on open-source software projects.

Keywords: software quality, technical debt, principal, interest, breaking point

Alexandros Michailidis

29/12/2017

Contents

Table of Pictures

Table of tables

1 Introduction

The concept of "cost of poor quality" or "poor quality costs" has been popularized in quality management literature by Harrington, and refers to costs derived by the production of low quality or defective products. Cost of poor quality includes both the cost of bringing the product to optimum quality level and the cost to repair any defects in delivered products. Respectively to the term of poor quality costs, the software engineering community has adopted the concept of technical debt, which refers to the aforementioned cost in software development. The term Technical Debt (TD) was introduced by Ward Cunningham in 1992, as a metaphor to financial debt, in the sense that the development of not-quite-right code can be perceived as a debt that will have to be repaid in the future as higher maintenance effort. Technical debt can be defined as "...a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible..." [1].

Technical debt and its management have gained increasing interest by the community during the last years, as 90% of papers on this subject have been published after 2010. According to Li et al [2]. 43% of this research is performed in academia, 40% in industry and 17% in both fields. Research is focusing on issues related to cost and benefit analysis of software quality management. Although research on TDM has been intense in the last years, the community still faces major challenges, such as: (a) the estimation of the amount of TD (i.e., the quantification of principal and interest) [4], and (b) the methodologies that can be applied for managing the increase in TD amount (due to the accumulation of interest) [5].

To address these challenges, at their previous work, Ampatzoglou et al. [4] [5] have defined a theoretical framework (FITTED – Framework for managing Interest in

Technical Debt) that can be used for the long-term management of Technical Debt. According to FITTED, as interest accumulates during software evolution, it can potentially reach an amount larger than the amount required for repaying the principal of technical debt at earlier phases of the development. Therefore, it is critical for project managers to be able to estimate the instance at which the accumulated interest will reach the amount of technical debt principal. The time point at which the accumulated interest equals the amount of principal is called the 'breaking point', in the sense that any benefit deriving from the decision to take on technical debt is being neutralized by the need for extra maintenance effort as software evolves [5]. Another interesting point in the evolution of software (inflection point) is the time at which maintenance effort exceeds effort of implementing new functionality.

In this study we build upon the FITTED framework [5] to explore the sustainability of industrial software applications. In particular, we investigate if the technical debt breaking point is shifted to the future or comes closer to present, as the software evolves. To achieve this goal, we study the full evolution of five industrial applications and we estimate: (a) the amount of technical debt, (b) the interest to incur in future maintenance activities, and (c) the breaking point for all versions of the system. Using this information, we assess the trend in the cornerstones of technical debt managements (i.e., principal and interest). As analyzed by Ampatzoglou et al. [4], the breaking point can be calculated by the ratio of technical debt principal over the average interest per software version. More specifically, for any software system, a set of refactoring suggestions, which can be applied in order to increase quality, can be extracted. Applying these refactorings will lead to an optimal design, as defined by a fitness function. The difference in effort between the optimum and the actual system can be used as an indicator of the technical debt principal. The ratio of the optimum system's quality over the actual system's quality is perceived as an estimator of the ratio of increased effort over the original effort (interest) for maintenance purposes. Finally, in order to assess the reliability of the proposed approach we perform the aforementioned analysis using four different fitness functions, and explore to what extent the outcome of instantiating FITTED is influenced by the subjective selection of metrics.

The study is organized as follows: In Section 2 we provide some background information for technical debt along with the definition of interest in economics. In section three we present in detail, the FITTED framework, as it is defined by Ampatzoglou et. al. Next, in section 4, we analyze in depth the functionality of the software created for validating the FITTED framework, along with the technologies used. In section 5 we present the case study of our research, whereas in section 6 we discuss the results and answer on the research questions. Finally, the conclusions of our work and our future plans on the topic are presented in sections 7 and 8 respectively.

2 Background Information

This chapter provides the reader with the basic knowledge required to comprehend the topic and the terminology behind the main idea of this dissertation, the management of the technical debt.

2.1 Literature Review on Technical Debt

Software companies often decide to rush the development of a software product rather than implementing with top quality along all the development phases. This shift-towards approach might be a time saving solution, however it results into poorly specified, designed and implemented software, with increased future maintenance costs. This kind of financial consequence, in software engineering, is defined as the technical debt.

Due to its multidisciplinary nature, technical debt combines elements from both economics and information technology scientific sectors, which could result into a communicational gap between the stakeholders involved in the development of a software product. Consider a development team which consists of project managers and technical staff such as designers, developers and testers. On the one hand the managers are not directly associated with the implementation of the product during the various software lifecycles. Contrariwise they are mostly occupied with time management and financial tasks, as they are primarily interested in increasing the benefit of the company along with decreasing the production costs by publishing the product into the market the soonest possible. On the other hand, the hands-on stakeholders are keen on making extra effort on designing and implementing the artifact, as the enhanced quality will ease post-production activities such and reusability and maintenance. Moreover, identical or similar projects will reach the market with less effort needed, therefore increased profit for the organization. Nevertheless, the final decisions are always taken by the project managers, thus the technical staff should be able to argument on the importance of the architectural quality of the product. In case that a decision of emerging the development process of the software is taken, the possibility of skipping important steps of the software lifecycle increases. Consequently, the terminology of the technical debt is a “bridge” that connects the two clashing scientific sectors.

In their previous work Li et al. [7] classified technical debt into ten major types, based on the reason that occurred its appearance:

1. **Requirements TD**: Represents the gap between the optimal (probably desired by the client) requirements and those implemented at the release of the artifact.
2. **Architectural TD**: Caused by immature or hurried architectural decisions which result into quality defects and reduces maintainability and sustainability of the product.
3. **Design TD**: Refers to the shortcuts that are taken in detailed design.
4. **Code TD**: Refers to the poorly written code that violates best coding practices and rules.
5. **Test TD**: Refers to the shortcuts or even to the lack of testing.
6. **Build TD**: Flaws in the build system, or in the build process of the software that makes the build procedure overly complex and difficult.
7. **Documentation TD**: Insufficient, incomplete, or outdated documentation in any aspect of software development.
8. **Infrastructure TD**: Refers to a sub-optimal configuration of development-related processes technologies, supporting tools, etc.
9. **Versioning TD**: Problems in source code versioning, such as unnecessary code forks.
10. **Defect TD**: Refers to defects, bugs, or failures found in software systems.

2.2 Financial Approach on Technical Debt

In the following paragraphs we are going to analyze the financial aspects of technical debt, trying to familiarize people with a software engineering background with terms, definitions and approaches which belong to the economics sector, but are fundamental for the measurement of technical debt. The term *debt* describes an amount of money that has been lent by one individual/team to another individual/team temporarily. The debtor, the person that was borrowed the money, is obliged to *repay* a larger sum which accrues as the aggregation of *principal* and *interest.* The principal denotes the original amount of money lent while the interest represents the additional fee that the debtor must pay to the lender, as a "compensation" for the undertaken risk of their agreement (the risk includes either the loss of money due to a non-repayment or the loss of benefit from a possible investment). The interest is calculated as a percentage, *interest rate*, over the principal and it can be classified as:

- **Simple or Compound:** When calculated by multiplying the interest rate with principal for each period of loan it is characterized as simple, however when adding the previously accumulated interest it is called compound. Simple interest increases linearly, while compound on an exponential factor.
- **Fixed or Floating:** Fixed interest is stable over period, whereas floating is characterized by fluctuations from period to period.

Interest is the term that mostly concerns us, in the estimation of the breaking point, thus it is worth foreseeing a bit. We should mention that, in technical debt, interest rate is not directly defined, so we are unable to apply the classification between fixed and floating. However, we can surely assume that technical debt is compound, since the effort to repay the technical debt when maintaining or expanding a non-optimal version of the software increases as the artifact evolves.

Based on the aforementioned statement, that technical debt is a multidisciplinary domain, Ampatzoglou et al. [8] conducted a research trying, to identify which perspective of research is the most overwhelming and concluded on the following classification depending on the aspect chosen:

1. **Emphasis on the software engineering**: Studies based solely on software engineering terms, such as code smells, to calculate the debt, setting aside any financial approach.
2. **Emphasis on the finance model**: Studies that touched only the surface of software engineering but deeply adopted financial methods such as Cost-Benefit analysis, Portfolio Management and Real Options which will be covered on the next sections.
3. **Emphasis on both aspects**: Studies that combine elements from both scientific fields in order to reach a rational and robust result.

Actually, a fourth category does exist, which includes studies that refer only on the basic financial aspect without occupying with software engineering. However, at this point of research we are not further interested in taking those into account, because mostly they argue with the existence of technical debt on systems which were built using agile technologies. Finally, the approach we adopted follows the 3rd model as we are using software evaluation metrics as well as financial standards to calculate the technical debt, but this methodology will be discussed in the following sections.

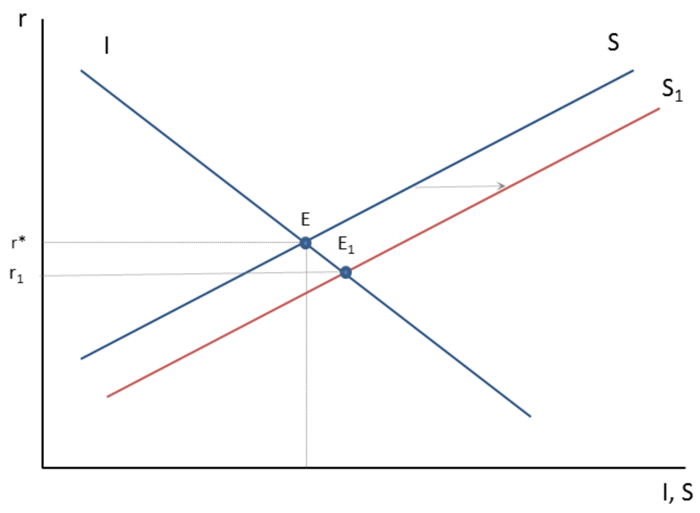
2.2.1 Interest Theory in Economics

In economics theory, different schools of economics have suggested different models on the definition of interest rate in the market. The Loanable Funds Theory, proposed by the Neoclassical School, and the Liquidity Preference Theory, introduced by the Keynesian theory, are the mainstream theories on the issue [11].

Interest rate can be considered as the price of money, in the sense that it is the price paid for borrowing money, or, on the other hand, the payment received when lending money. As any other price in economics, interest rate can be defined in the market by the equilibrium point, where market supply equals market demand. Loanable Funds Theory suggests that interest rate is defined in the market of loanable funds. On the one hand, demand for loanable funds is formed by individuals or enterprises that are eager to invest and ask for loans in order to fund their investments. In this case, the higher the interest rate gets, the more expensive borrowing money becomes. Consequently, as interest rate rises, demand for loanable funds falls. On the other hand, supply of loanable funds is generated by individuals or enterprises who want to save their money, using the loanable funds market. In order to save their money for later, instead of spending their entire income, they decide to put part of it into the loanable funds market. Consequently, as interest rate gets higher, return on savings also rises. As a result, supply of loanable funds increases as interest rate increases.

Picture 2.1 presents the equilibrium in loanable funds market; with the mention that, in economic theory, in supply – demand diagrams, the dependent variable is represented on the horizontal axis, whereas the independent variable on the vertical axis. Accordingly, in the diagram of picture 2.1, interest rate (r) is depicted on the vertical axis, whereas supply and demand for loanable funds are denoted on the horizontal axis. Line S is the supply curve of loanable funds, which represents the quantity of loanable funds supplied in the market at any interest rate. The positive correlation between interest rate and loanable funds supply is indicated by the positive slope of line S. Besides, line I is the demand curve for loanable funds and it represents the quantity of loanable funds demanded in the market at any level of interest rate. The negative slope of line I shows the negative correlation between loanable funds demand and the level of interest rate.

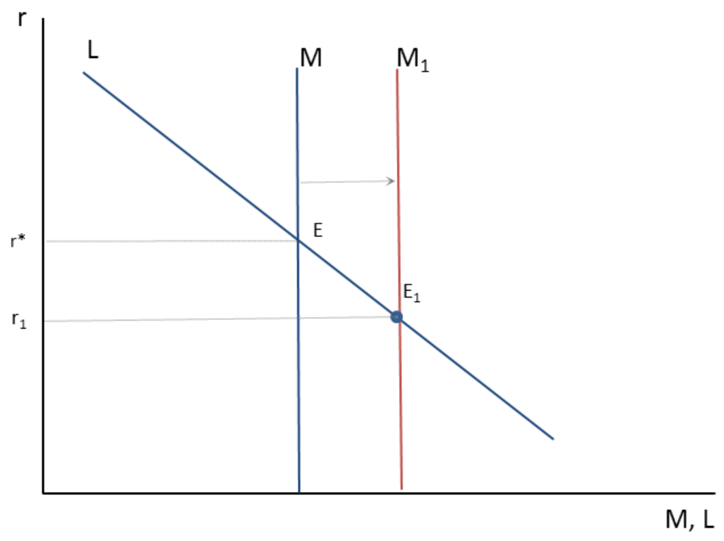
For interest rate levels higher than r*, quantity supplied is greater than quantity demanded because lending money is more profitable than borrowing, due to the high interest rate. On the contrary, for interest rate levels lower than r*, quantity supplied is less than quantity demanded because, due to the high interest rate, borrowing money is more profitable than lending, i.e., investing is more profitable. At interest rate level r=r*, loanable funds supply is equal to loanable funds demand. Consequently, neither the investors nor the savers have any motivation to alter their position in the market, thus equilibrium is achieved, and interest rate is specified at the level of r=r*.



Picture 2.1: Loanable Funds Theory

The equilibrium point does not change, when any other variable in the market, that could influence savings or investment, stays constant. In economics, this situation is described by the Latin phrase ceteris paribus, meaning that all other factors are stable. Ceteris paribus is set as a condition in order to investigate the relationship between two variables. Therefore, interest rate r* is the equilibrium interest rate only when any other factor is stable. It can move upwards or downwards if savings or investments alter due to an exogenous factor, i.e., income. If income increases, the amount of savings would also increase. As a result, in picture 2.1, the savings curve (S) would move to the right and the new loanable funds supply curve would be represented by the new line S1. Accordingly, a new equilibrium would be achieved in the market, depicted by the point E1, and the new equilibrium interest rate would be defined lower than r*, at the level of r1.

According to the Liquidity Preference Theory, interest rate level is determined in the money market, through the mechanism of supply and demand for money (i.e., cash). Under this perspective, it is assumed that, the quantity of money (cash) that people choose to hold for causes of transactions, precaution or speculation consist the demand for money (L). If interest rate increases, people find investing more appealing (profitable) than holding their money (because interest rate is considered as the cost of holding cash against to investing). Hence, there is a negative relationship between interest rate level and demand for money; as interest rate increases, demand for money decreases and vice versa. On the other hand, supply of money (M) is determined by the central bank, depending on the requirements of the economy. Thus, supply of money does not depend on the level of interest rate; it is given at any point of time (this hypothesis expresses the main contradiction with the Loanable Funds Theory). Correspondingly to the Loanable Funds Theory, the point where supply equals demand is the equilibrium point and determines the level of the interest rate in the market.



Picture 2.2: Liquidity Preference Theory

The equilibrium in the market of money is shown in the diagram of picture 2.2. The vertical axis depicts the interest rate, while the horizontal axis represents money supply and demand. As mentioned above, money supply is independent of the interest rate and it is determined by the central bank. Therefore, money supply curve is represented as line M, vertical to the horizontal axis. Besides, the negative correlation between demand for money and interest rate is depicted by the negative slope of line L which represents the money demand curve. Line L shows the quantity of money that people demand at any level of interest rate. The point where the two curves (M and L) intersect is regarded as the equilibrium of the market and determines r* as the equilibrium interest rate, ceteris paribus.

In case another determining factor (e.g. income) causes demand for money to change, or in case the central bank decides to increase or decrease money supply in the economy, equilibrium point will shift, and interest rate will change. For example, if monetary authorities decide to increase the quantity of money supplied, then the supply curve will move to the right, to the new line M1. As a result, E1 will be the new equilibrium point and interest rate will be now determined at the level of r1, lower than r*.

3 Proposed Interest Theory

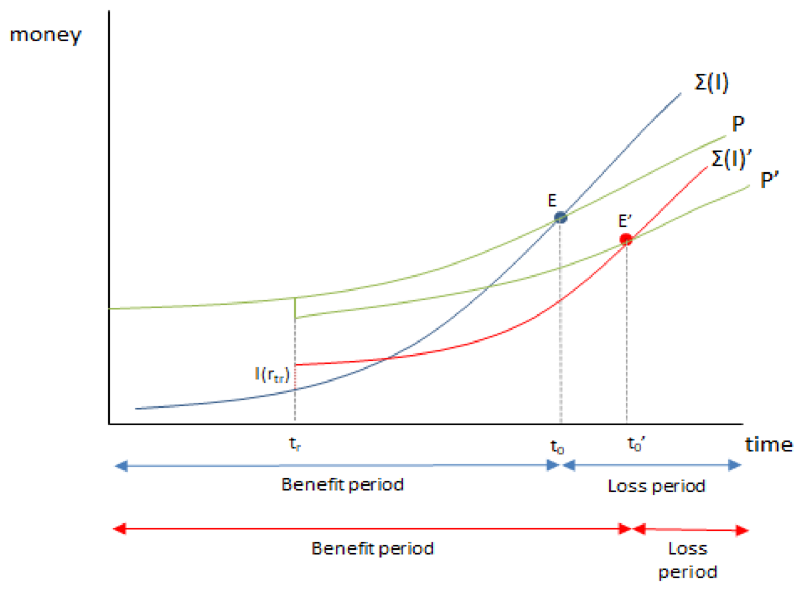
In this section we present the rationale of FITTED framework, as imposed by Ampatzoglou et Al. [4] [5], the framework upon which we based our calculations. We also provide our methodology for converting those theoretical ideas, into functions and calculations which will be performed by the tool that we are going to present on the following chapters.

3.1 Technical debt sustainability approach

Based on the rationale of the existing economic interest theories, described on section 2.2.1, Ampatzoglou et al. [10] developed a theory for managing technical debt interest Specifically, they adopted the Liquidity Preference Theory. The reason for selecting this concept in favor of the Loanable Funds Theory is that, in technical debt, the amount of money that is available to the software development company for managing the technical debt is stable, i.e., the principal. Under this perspective the money supply is paired to the principal, since principal represents the amount of money that the company can manage, after technical debt occurs., while the money demand is paired to the accumulated interest, in the sense that interest embodies the extra amount of money that will be necessary for future maintenance actions, caused by the incurring technical debt. As long as accumulated interest is lower than the principal, the benefit derived by the initial decision to save effort is more than the cost generated by the extra effort to maintain the software. On the other hand, when the accumulation of interest overcomes the principal, then the cost needed for the system's maintenance becomes more than the money saved when technical debt incurred. Consequently, the point where the accumulated interest is equal to the principal (breaking point) is very critical in technical debt management and can help project managers in their decision making.

Picture 3.1 represents the FITTED Interest Theory. The horizontal axis depicts time, while the vertical axis stands for money amount. The blue curve Σ(I) denotes the increasing cumulative interest, as the software develops. We consider cumulative interest as continuously increasing, because it consists of the accumulation of interest in every successive version of the project and because technical debt interest is compound [8]. Principal, on the other hand, is represented by the green line P and is also considered as increasing, in the sense that the introduction of new functionality to the software generates new technical debt items.

As described by the figure, until time stamp t_0, interest is lower than the principal and taking on technical debt can be perceived as beneficial for the project. After time point t_0, interest accumulates and becomes larger than the principal and extra maintenance costs exceed the initial money saving. Therefore, the equilibrium point E_0 denotes the time stamp, at which the company has spent the complete amount of money, hence the principal, in maintenance actions because of the technical debt.



Picture 3.1: FITTED framework

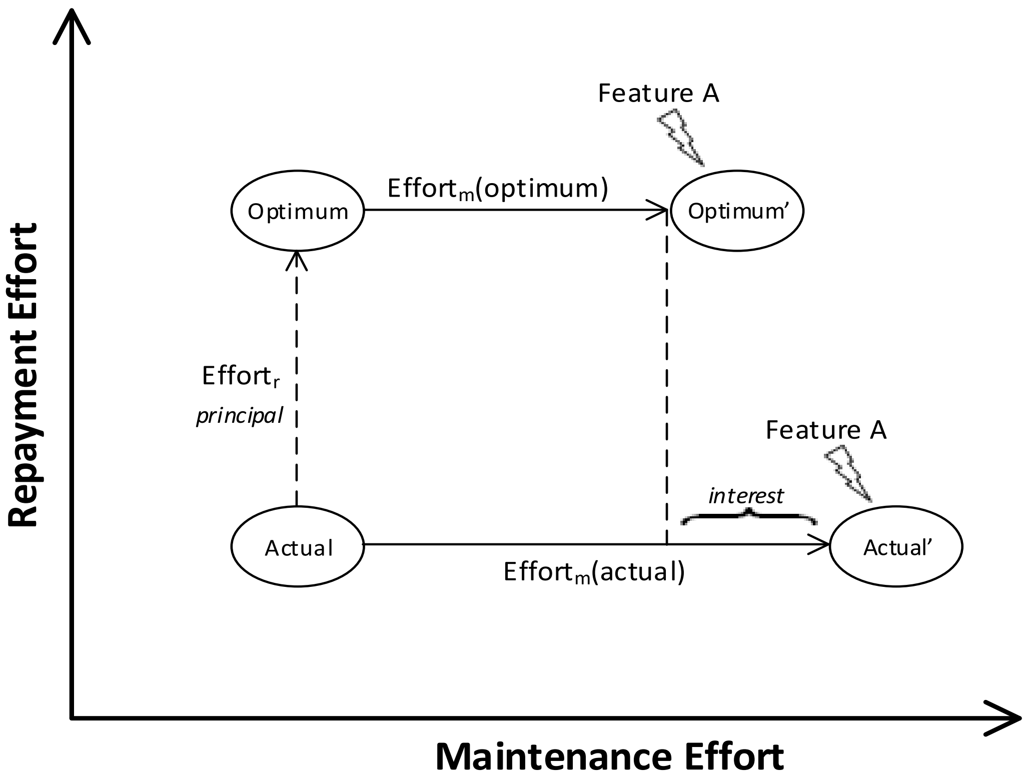
If the development team proceeds with some repayment activity, e.g. at timestamp t_r, accumulated interest increases, because of the addition of the repayment effort, and line Σ(I) shifts upwards to Σ(I)΄. However, the slope of the line decreases, since lower maintenance activities are expected in the future due to the repayment. On the other hand, principal decreases at time stamp t_r, because of the repayment activity, and then follows the course of line P΄. Consequently, the equilibrium point moves to the right (E'), and the benefit period increases to time stamp t_0' [5]. Hence, in the case that no repayment is made, if the life cycle of the software is expected to be shorter than time stamp t_0, then technical debt is beneficial for the company. If the software's lifespan expands after t_0, then taking on technical debt is harmful decision.

It has to be mentioned that we define the equilibrium point based only on the estimation of effort, i.e., the effort saved when technical debt occurs, and the effort required for extra maintenance due to technical debt accumulation. For the shake of simplicity, our analysis excludes other costs or benefits caused by undertaking technical debt, for example any financial benefits gained due to early product release. The proposed interest theory can help practitioners in their decision making by:

- Identifying the timestamp in which incurring TD, becomes harmful for the company. Thus, they can decide if they should undertake the debt.
- Supporting the continuous monitoring of the interest that has been paid so far.
- Evaluating the repayment activity, based on the time-shift of the equilibrium point that it offers.

3.2 Application of FITTED Interest Theory

As an illustrative example of how the FITTED interest theory can be applied, Chatzigeorgiou et al. [10] proposed an empirical methodology for calculating the expected time when the equilibrium point will be reached. Based on this study, the time point at which the equilibrium is reached has been termed as the breaking point of technical debt. To formulate principal and interest, Chatzigeorgiou et al. used the rationale depicted in picture 3.2. In particular, it is assumed that for each object-oriented software an optimum design does exits, or at least it can be assumed that it exists. The quality of every system can be calculated with the use of a fitness function which actually denotes the *"distance"* between the current system and the optimal one. Also, using search-based optimization a design that optimizes the value of this function can be obtained. The effort needed to transform the actual system to the optimal one can be defined as the principal. Furthermore, it is reasonable to assume that extra effort needs to be performed in order to add a new feature on the actual system, rather than adding the same feature to the optimal system. This difference in effort represents the interest.



Picture 3.2: Increased maintenance effort for technical debt item

To instantiate this approach with a specific implementation, the following actions/assumptions can be made:

- Fitness value: Ratio between coupling and cohesion, two well-known software quality metrics.

- Principal: Number of simple refactorings needed to transform the actual design to the optimal one.
- Interest: The ratio of the levels of design quality (fitness function) is correlated to the ratio of the two models (the actual and the optimum).

3.3 Estimation of Principal

Technical debt principal can be calculated as a function of three variables. The first variable is the number of problems that must be fixed, the second one is the time required to fix each one of these problems, and the third one is the cost for fixing each problem. Regarding the number of must-fix problems, Ampatzoglou et al. [10] have presumed that for any object-oriented software an optimal design quality does exist, which can be estimated by a proper fitness function. Under this perspective, a measure of coupling and cohesion have been selected to serve as a fitness function, whose role is to assess the allocation of a system's entities (methods and attributes) to the classes. Then, local search optimization algorithms can be applied, in order to define the optimum design that consists of the same entities that optimize the aforementioned fitness function. Therefore, the 'spread' between the optimal and the actual design, as it can be derived by the difference in their fitness function values, can be mapped to the principal, i.e. the effort needed to convert the actual system to the correspondent optimum one. This notion of "distance" is depicted in picture 3.2, as Effort$_r$. For this study, we have decided to base our estimation of principal on the computation of a widely used platform, SonarQube. SonarQube extracts a metric called Sqale Index, which represent the time (in minutes) needed for the average developer to fix all the maintenance problems of the analyzed artifact. Therefore, by taking into account this metric and that the hourly rate of the average developer is 45.81$ [16] we can calculate the value of principal in dollars as:

$$Principal(\$) = Hourly\ Rate * Sqale\ Index\ (in\ hours)$$

Picture 3.3: Principal in dollars

3.4 Estimation of Interest

Assuming the optimal and actual systems of picture 3.2, under normal circumstances, maintaining the optimum system requires less effort than maintaining the actual system. As shown in the figure, adding a new feature A to the optimal system needs a certain effort, noted as $Effort_m$ (optimal), whereas adding the same feature to the actual system necessitates a larger effort, noted as $Effort_m$ (actual). The difference between these two efforts represents the interest that is accumulated during this maintenance activity, i.e., the addition of feature A. However, since the evolution of the software cannot be predicted, it is not possible to foresee what kind of modifications will be made in a software system during future releases. Hence, we found our assessment of future maintenance effort on historic data, by considering effort spending on maintenance activities.

More specifically, although many measures can be used for the calculation of past effort, we have selected to use the average number of lines of code added between sequential releases. Added lines of code indicate the amount of effort required for the addition of new functionality and in a way the effort needed for the change of existing modules. Supposing that an average of *k* lines of code is added in the transition from one version to another, it is reasonable to assume that adding k lines to a high design quality system (expressed by the value of the employed fitness function) is easier than making the same addition to a lower design quality system. At this point we suppose that maintenance effort is dependent on the design quality, as in picture 3.4, where c represents an arbitrary constant.

$$Effort_m = c \cdot FitnessValue$$

Picture 3.4: Maintenance effort

Based on the abovementioned approach, the fitness value for both the optimum and the actual systems can be calculated and therefore we can capture the proportion of the theoretical over the actual effort, as in:

$$\frac{Effort_m(optimal)}{effort_r(actual)} = \frac{c * FitnessValue(optimal)}{c * FitnessValue(actual)} \Rightarrow$$

$$Effort_m(optimal) = \frac{FitnessValue(optimal)}{FitnessValue(actual)} * Effort_r(actual)$$

Hence, if past maintenance effort (i.e., the number (k) of added lines) is used to define the actual maintenance effort, then the optimum effort can be directly obtained. Consequently, the amount of interest accumulated between any two sequential versions can be calculated as the distance between the optimum and the actual effort and is given by picture 3.6.

$$(Interest) = \Delta Effort = k * \left(1 - \frac{FitnessValue(optimal)}{FitnessValue(actual)}\right)$$

Picture 3.5: Calculation of interest

Given the fact that interest is closely related to the system's maintainability, our analysis concentrates on the estimation of interest, based on design time quality attributes, such as inheritance, cohesion, coupling, complexity and size. So, in order to define the fitness value used for the estimation of the interest, we use a set of source-code level metrics which will be presented in the following chapters. More specifically, in order to define the optimum design, our analysis consists of the following steps. Firstly, classes of high similarity level with the investigated class are found. Secondly, the optimum value for each one of the aforementioned metrics is identified and the difference between the actual and the optimum fitness values is calculated and the average distance between the actual and optimum design is assessed. Finally, by taking into account that the average developer adds 25 lines of code per hour [17] and the developer’s hourly rate [16], we can produce interest in dollars by calculating k in picture 3.7 as:

$$Interest(\$) = \left(\frac{Lines\ of\ Code * FitnessValue}{Hourly\ Lines\ of\ Code}\right) * Hourly\ Rate$$

Picture 3.6: Interest in dollars

3.5 Breaking Point

By having formulated both the principal needed to perform all the code refactorings for maintenance, as well as the interest that is being accumulated from version to version, we are able to calculate the number of versions when the breaking point occurs by the following function:

$$Breaking\ Point\ (Versions) = \frac{Principal\ (\$)}{Interest\ (\$)}$$

Picture 3.7: Breaking point in versions

4 Contribution

It follows from the previous chapters, basically from the literature review, that for generating the results, to successfully monitor the evaluation of the breaking point, it would need plenty of recursive mathematical calculations. Thus, we automated this procedure, in order to produce accurate results, by developing a desktop Java application, called Breaking Point Tool. Essentially, this chapter represents the “incarnation” of the aforementioned methodology. The upcoming sections, represent an in-depth analysis of the steps we followed, for transforming the theoretical ideas into practical actions, through the Breaking Point Tool.

4.1 Technologies Used

4.1.1 Java

Java was developed by Sun Microsystems, and in particular by James Gosling in 1995. It is a high-level and general-purpose programming language with plenty of libraries that support the implementation of applications of any technology. It is based on classes and interfaces, characterized for its object-oriented nature and supports any programming paradigm. The fact that Java is a complied programming language, increases the execution speed of applications and is one of its main advantages compared to interpretable script languages.

It is relatively easy to be learned for beginners or developers who are knowledgeable of C++, as Java has borrowed many of its features. Characterized for the clean and readable code, which consists of reserved keywords of English terminology. It has an extra-management system, automatic garbage collector and supports multi-threading and multiprocessing. It is extendable and scalable, which facilitates the maintenance of its programs. However, it is particularly sensitive to error handling, although it provides users with sufficient information to find and correct them. An additional key disadvantage is that it can bind multiple memory resources.

Basic uses of Java:

- Desktop Applications
- Android Development
- Web Services
- Web applications
- Video Games
- Network Applications

In recent years there has been a rapid increase in the use of Java on web applications industry, larger than in other scripting and programming languages. This is due to the turning of large corporations (Oracle, Yahoo, Microsoft and Apple) and developers to its use, given its multiple benefits. In order to implement web applications and web services, several frameworks do exist, with the most common of those being, Spring MVC, Spark, Struts and JSF. There are also several Integrated Development Environments (IDEs) for developing Java applications. Finally, it has programming communities that provide information and help for new developers.

4.1.2 Perl

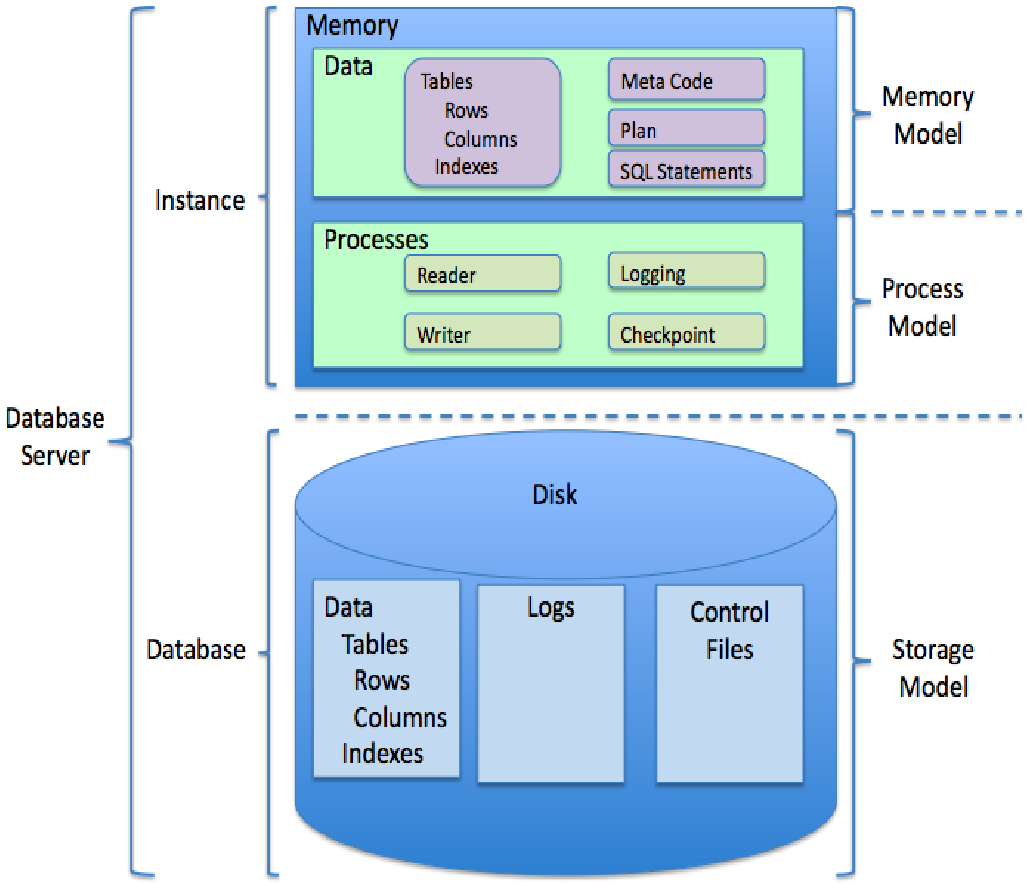
Perl was originally written by Larry Wall at NASA's workshops. Perl's original purpose was to function as a glue programming language for him and his colleagues. By "glue", we refer to a programming language specifically designed for creating applications that link different software components to each other. Perl incorporated the advantages from many programming languages. For example, he incorporated functions for regular expressions from sed (stream editor to UNIX), the awk template scanning function and many characteristics from other programming languages. Its syntax is derived from C, Pascal, Basic, UNIX shell languages and definitely the English language.

The first edition of Perl (Perl 1), appeared on December 18, 1987, and gradually evolves until nowadays with the contribution of countless people. The second version (Perl 2) extended the operation of regular expressions, while the third version (Perl 3) gave to the language, the ability to be able to handle binary data. Up to that time, no official documentation has been created for it, except for a simple man page. The fourth version (Perl 4) was released along with the book written about language management, which is called the "Camel book", without adding any significant functionality to that version. The substantial change in Perl was made with the release of the fifth edition (Perl 5), which had many changes to the syntax compared to the previous versions, but also fantastic extensions to its functionality. Up to now, upgrades have been made to the specific language and the latest version is Perl 5.26 released on 2017. It is worth noting that alongside Perl 5, Perl 6 was created. Perl 6 started to expand Perl 5, but ended up in a different language. Now there are different groups dealing with Perl 5 and Perl 6, by exchanging ideas with each other.

Perl was developed to be easy for people to write. The syntax of the language is almost like the speech of people, and not like other languages with a strange and rigorous syntax. Perl is pretty portable. The term portable as understood implies the ability to transfer a program written in Perl from one operating system to another and to be executed with absolute success. Perl is available in a variety of operating systems, resulting in the code being executed everywhere without any change to it. Perl can think of words or sentences, while other languages handle one character at a time. It is also characterized for the tremendous regular expression management system, as it can detect a template in a text in many ways, very easily and very quickly. Perl is a high-level, general purpose and dynamic language. Its most popular use is on CGI (Common Gateway Interface) scheduling. It is the main power of the well-known websites such as Amazon, Deja, and Slashdot. Finally, it is naturally used by many people for the original purpose it was created, to extract data from a source and translate them into another form.

4.1.3 MySQL

MySQL is a relational database management system. Its name is a combination of the name of its founder, (Michael Widenius) and the Structured Query Languages (SQL) abbreviation. Its first version was launched on 1995. MySQL supports a client-server architecture which means that a single database can serve multiple remote clients (applications). It organizes the data based on the relational model which is illustrated on picture 4.1.



Picture 4.1: Structure of the relational model

The major features of MySQL are:

- Open source software
- Can handle huge amounts of data without adversely affecting its functionality
- SQL syntax format

- Can be used on many operating systems (AIX, BSDi, FreeBSD, HP-UX, eComStation, i5 / OS, IRIX, Linux, macOS, Microsoft Windows, etc.)
- Can be combined with many programming languages (PHP, Perl, Java, C, C++, etc.).
- Supports large volumes of data (up to 50 million lines in a table). The default size for a table is 4GB, however the capabilities of the operating system can extend this up to terabytes (TB).

4.1.4 JDBC & DBI

As presented in the previous sections, MySQL can be combined with many languages. On our server-side implementation we used a script written in Perl, while the Breaking Point Tool has been developed in Java. In order to communicate with our MySQL database for the addition and retrieval of data, we used the corresponding libraries for each language, which are JDBC for Java, and DBI for Perl.

4.1.5 SonarQube

SonarQube is an open source platform for continuously monitoring the quality of the source code, which is of critical importance, for the developers. SonarQube was formerly known as Sonar, meaning that both terms are used to describe the same platform. It collects and analyzes information, relevant to the source code, measuring its quality and extracting conclusive data for it. It combines statistical and dynamic analysis tools, while it allows constant quality measurement. Every element that might have a negative effect on the source code (from minor user interface details to critical design flaws), are monitored and evaluated by SonarQube. As a result, the developers can view the source code analysis and have access to any debugging errors recorded by the platform. Sonar analyzes the source code from different angles, while checking in detail every single of its levels (project level, package level, class level). For every one of those levels, metrics and statistics are produced, revealing problematic points, in the code, that will need further improvement.

SonarQube, also, provides momentary perception of the code and finds spots either in which the code is relatively inferior or that may prove to be a fuss in a later stage. An indicator, that is able of tracking future problems, is Coverage. Coverage is used to measure the percentage of successful test scripts, which were applied in the code. For example, the Coverage indicator provided by Sonar is 50%, which does not seem that satisfying at first. However, what actually should be measured is the Coverage indicator of the former version. If the older percentage was 35%, then sufficient improvement was implemented in the code and therefore its quality. On the other hand, if the older percentage was 70%, then the changes applied in the code negatively affected its quality, which means that even more work is required to upgrade it. The platform does not simply note what is wrong in the code. It, also, offers quality management tools, that suggest how these flaws can be fixed. These tools are:

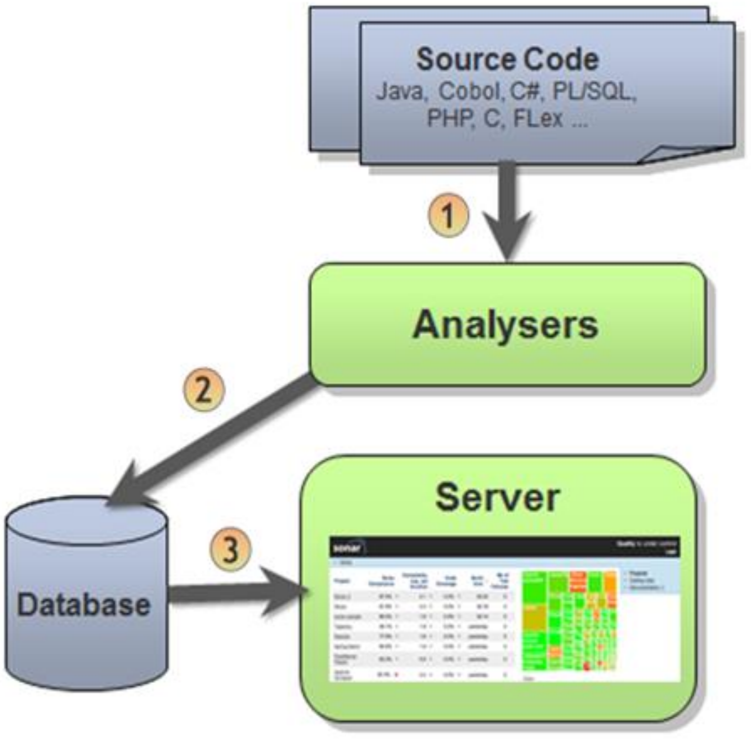
- IDE integration
- Integration for Jenskins
- Continuous Integration server
- Code-review tools

At this point, it is worth mentioning, that SonarQube is not the only software that provides information regarding the quality of the code, but any other software is focused in error finding, sonar has focused to what its creators call "seven axes of quality". These axes represent bugs and potential bugs, coding standards breach, duplications, lack of unit tests, bad distribution of complexity, spaghetti design and not enough or too many comments. Some software that function in the same way as Sonar does, do not just focus on bugs and potential bugs, but, also, on lack of unit tests and API documentation. So, as said, the reason Sonar is chosen among others, for the thesis, is the fact that it focuses on all seven axes.

SonarQube does not only help developers, but, also, testers, managers and businessmen as well. Regarding testers, the platform can indicate any spots in which the testing was too superficial or non-existent at all. Naturally, the gain for the developers is that it helps them pinpoint their mistakes in the code. Regarding businessmen and managers, statistics and metrics provided by the platform, with diagrams for the code which help them make any kind of necessary change or investments. SonarQube is written in Java and was initiated as a tool for analyzing Java programs. However, because of its vast development over the last few years, it can analyze even more programming languages including:

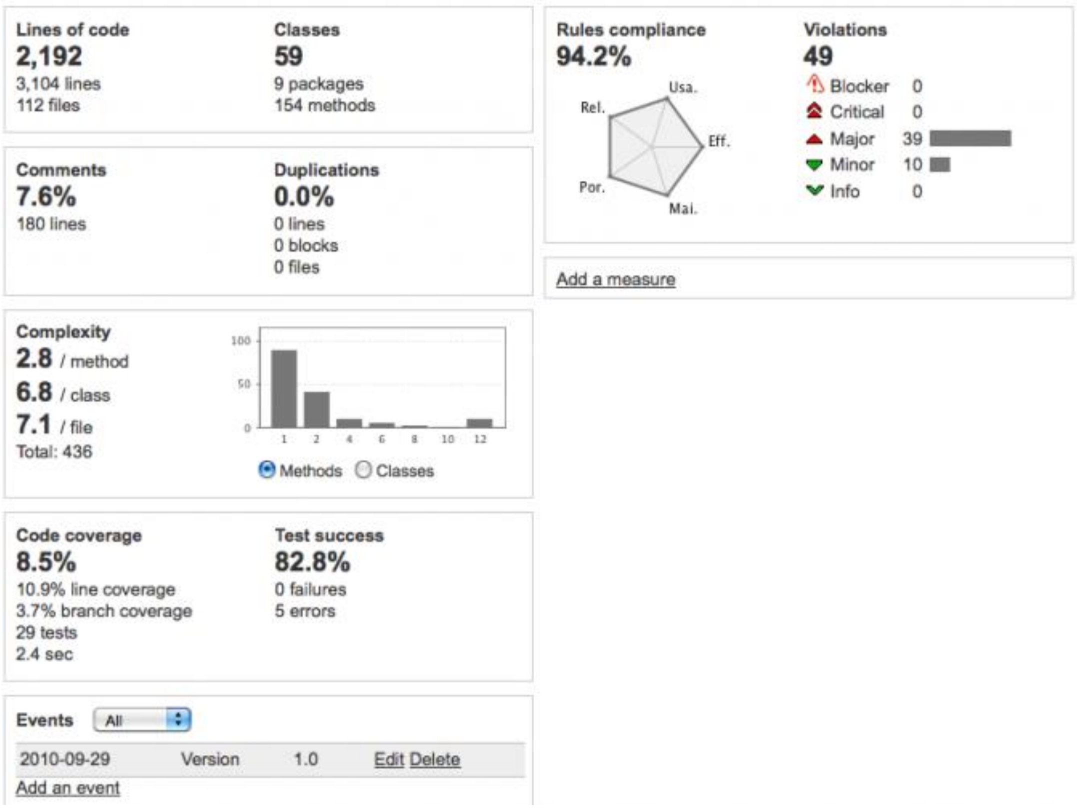
- C
- C++
- Javascript
- C#
- Java
- COBOL
- PL/SQL
- PL/I
- PHP
- ABAP
- VB.NET
- V86
- Python
- RPG
- Flex
- Objective-C
- Swift
- Web
- XML

As mentioned, a very important element of the platform is the creation of metrics and graphs based on the code. Available metrics exceed 140 in number and are stored in a database that has been modified accordingly. Picture 4.2 presents the structure of SonarQube. The first level depicts the source code to be analyzed, by using the relevant software (sonar-scanner or sonar-runner). The results of this analysis are then stored in the database and graphs are presented in the server by using a browser.



Picture 4.2: SonarQube structure

As seen in picture 4.2, the last level provides the information about the metrics in the browser, depicted by typing http://localhost:port_number. The default value for the port number is 9000, but this value can be updated manually. In picture 4.3, the analysis results of a project are presented. Initially, we can observe an overview of the project, which comprises of 2.192 lines of code on 59 classes, all includes in 9 packages, along with a coverage on the number of comments. A very important value is the complexity which is calculated on a method and a class level, as well as the success or failure of the tests written for the project.


Lines of code
2,192
3,104 lines
112 files

Classes
59
9 packages
154 methods

Comments
7.6%
180 lines

Duplications
0.0%
0 lines
0 blocks
0 files

Complexity
2.8 / method
6.8 / class
7.1 / file
Total: 436

Code coverage
8.5%
10.9% line coverage
3.7% branch coverage
29 tests
2.4 sec

Test success
82.8%
0 failures
5 errors

Events All

2010-09-29	Version	1.0	Edit Delete

Add an event

Rules compliance
94.2%

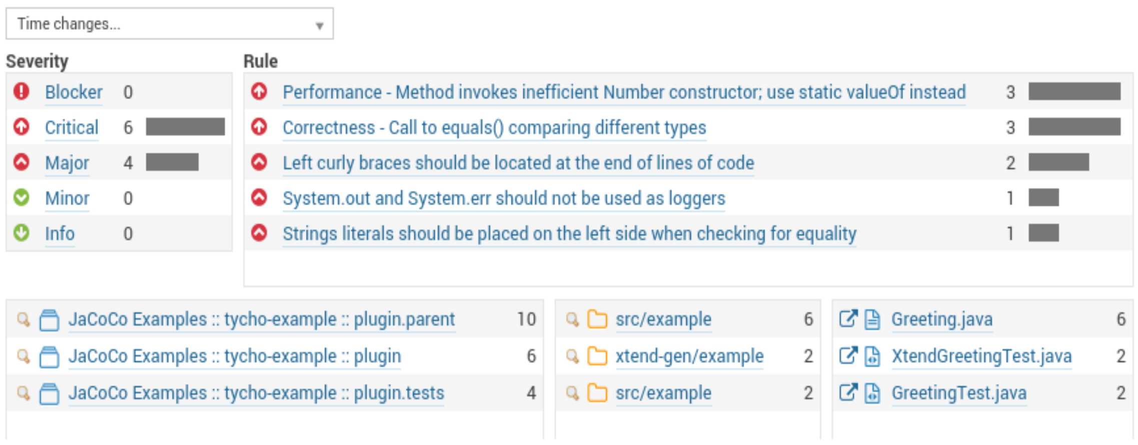
Violations
49

Blocker	0
Critical	0
Major	39
Minor	10
Info	0

Add a measure

Picture 4.3: SonarQube results

It is already highlighted, that the platform is capable of bug tracking. This functionality is depicted in picture 4.4, as all issues and their origin are presented along with a severity indicator for each error. For instance, 6 issues were detected as *“Critical”*, 3 of them of the category *“Performance”* and 3 on the category *“Correctness”*. The *“Rule”* board gives a detailed description on the violations and pinpoints the package or class, in which these flaws were detected.


Time changes...

Severity	
Blocker	0
Critical	6
Major	4
Minor	0
Info	0

Rule	
Performance - Method invokes inefficient Number constructor; use static valueOf instead	3
Correctness - Call to equals() comparing different types	3
Left curly braces should be located at the end of lines of code	2
System.out and System.err should not be used as loggers	1
Strings literals should be placed on the left side when checking for equality	1

JaCoCo Examples :: tycho-example :: plugin.parent	10	src/example	6	Greeting.java	6
JaCoCo Examples :: tycho-example :: plugin	6	xtend-gen/example	2	XtendGreetingTest.java	2
JaCoCo Examples :: tycho-example :: plugin.tests	4	src/example	2	GreetingTest.java	2

Picture 4.4: SonarQube violations

In conclusion, apart from the provision of metrics and statistics for the analyzed code, the most important feature of Sonar, is that it transforms these metrics into actual business values like technical debt and risk, which is the most intriguing part of this study.

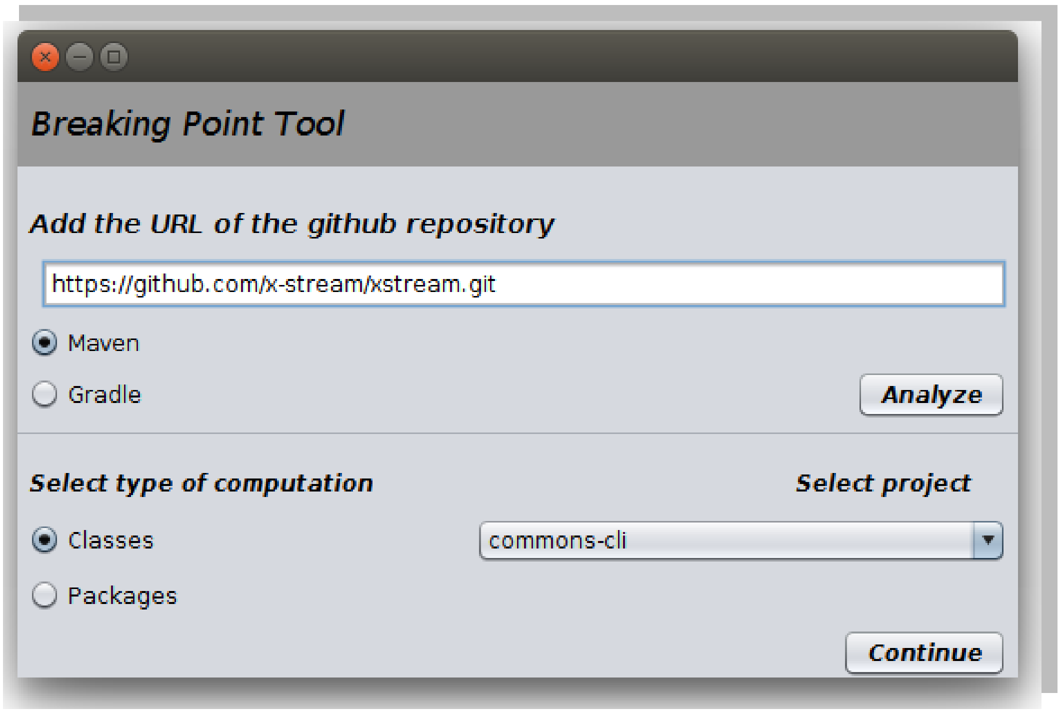
4.2 Breaking Point Tool

Breaking Point Tool (BPT) is a portable desktop application we created along with Christos Sarikiriakidis from University of Central Macedonia. As its name implies, the goal of this tool, is to effectively monitor the existence of technical debt and the evolution of breaking point, as the software matures in time.

The devastating majority of scientific software, accepts a specific input, processes the extracted data and produces an output, likewise BPT. Specifically, the unit of research is a single Git repository. The processing of the data, is performed into two major phases, the *analysis phase* and the *computation phase*, while the interpretation of results, thus the *presentation phase* can be done either from the user interface of the tool or by generating an extensive report into a .csv file. These three phases are presented through two separate screens as user interface.

4.2.1 Analysis Phase

In order to better interpret and increase the accuracy of the results, we needed a pool of several projects, each of them to be comprised of a large number of versions (ideally more than 20). To fulfil this precondition, we used GitHub, the well-known software platform, which hosts a vast number of open source projects available to the public. We should mention that our research will solely include with Java projects, which were developed using either Maven or Gradle as build tool. Picture 4.5 depicts the analysis screen of the tool.


Breaking Point Tool
Add the URL of the github repository
Analyze
Select type of computation
Select project
Continue

https://github.com/x-stream/xstream.git

[✓ Maven] Gradle

[✓ Classes] Packages

commons-cli

Picture 4.5: Analysis Screen

The primary action is to provide our software with a GitHub URL and select the build tool upon which the project was developed. The tool checks for the validity of the repository and in case it exists, it executes two asynchronous processes. This procedure takes place, for us to retrieve metrics needed for the calculation of the results. The first process in asynchronous and it concerns the analysis of the repository by SonarQube, while the second process takes place within the BTP by interacting with the user. The completion of both processes signals the completion of the analysis phase. However, this procedure may take sever hours, even days, so the computation of the results is not direct.

Analyzing a project with SonarQube

As already mentioned, SonarQube is an open source platform, dedicated on the analysis and quality measurement of software artifacts. It requires the installation and setup of an additional plugin, sonar-scanner, to inspect the code quality, detect code smells and bugs, and record quality. SonarQube, offers three different levels of analysis, project level, package level and class level, however currently, we are only interested in the last two scopes as shown in picture 4.6. SonarQube provides us with 5 metrics which we are using as a factor for similarity. Table 4.1 depicts these metrics, along with the description for each metric and the software quality attribute the represent.

Table 4.1: Similarity Metrics

Metric	Description	Quality Attribute
Classes	Number of classes (including nested classes, interfaces, enums and annotations).	General
Complexity	The complexity calculated based on the number of paths through the code.	Complexity
Functions	Number of functions.	General
NCLOC	Number of physical lines that contain at least one character which is neither a whitespace nor a tabulation nor part of a comment.	General
Sqale Index	Effort needed to fix all maintainability issues.	Maintainability
Statements	Number of statements.	General

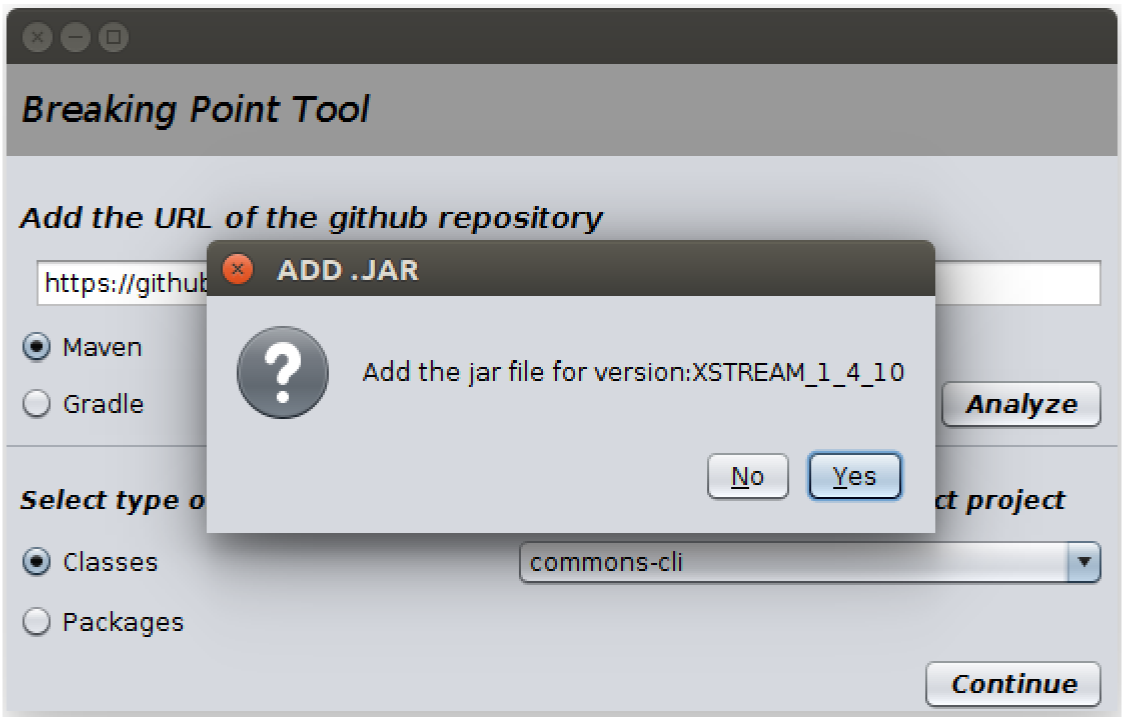
The software analysis by Sonar is a time-consuming procedure with the need for extended computational power, as it can last for several hours. Hence, this tool resides in our server on University of Macedonia and the communication between the BPT and Sonar is achieved by a mediator script, written in Perl.

By the time the user presses the analysis button, BPT connects to the server via SSH and executes the script, passing the URL of the repository as a runtime argument. The script communicates with the GitHub API to retrieve the history of the selected project. Specifically, the response contains a list with all the commits that the developer has “tagged” as releases, along with the release name and the commit hash for each version. Once the project’s history is retrieved successfully, the script clones the repository into the root directory of SonarQube’s installation. At that point the script acts as a manipulator of SonarQube. For each version retrieved, it uses the checkout command, to copy files from the history to the current working directory, hence the directory where the project is cloned and finally, it commands SonarQube to perform the analysis. The results of the analysis are stored in the database provided by SonarQube itself and can be accessed anytime.

Analyzing each version with Metrics Calculator

SonarQube does not produce all the metrics required to efficiently calculate the breaking point. Hence, we rely on a second tool in order to retrieve the remaining metrics. Metrics Calculator is an application which accepts .JAR file, analyzes the .class files that it includes and produces a .CSV file. The rows of the output file represent the analyzed classes, while each column a different metric for the analyzed artifacts.

The Metrics Calculator analysis takes places, instantly after the BPT has terminated the connection with the UOM server. The first action is to communicate with the GitHub API to retrieve the history of the project, likewise the Perl script, with the difference that BTP clones each version in a separate folder instead of checking out each version of the same directory. We have chosen this approach, in order to automatically package each version depending on the build tool that each project is built upon. The generated .JAR files are used as input from Metrics Calculator and the .CSV outcome is parsed from the BPT to retrieve and store the generated metrics into our database.


Breaking Point Tool

Add the URL of the github repository

https://githul

● Maven

○ Gradle

Analyze

Select type o

ct project

● Classes

○ Packages

commons-cli

Continue

ADD .JAR

Add the jar file for version:XSTREAM_1_4_10

No

Yes

Picture 4.6: Metrics Calculator Analysis

At this point we should mention a disadvantage of the BTP. Both Maven and Gradle specify the name of the .JAR file that will be generated in their configuration files, pom.xml and build.gradle respectively. In order to retrieve the name of each version, we had to parse those files and extract this specific variable. Not to fall behind with our research, we created a Dialog Popup as an alternative, for the user to manually add the .JAR of each version. Nevertheless, this is a feature that must be implemented in the near future as it will automate the whole procedure of analysis. The Dialog appears in picture 4.6 and by selecting the “Yes” option, the user is redirected to the root directory of each version and attaches the .JAR to be analyzed by the Metrics Calculator. It is obvious that the user will have to repeat this action as many times as the versions of the project, however he preserves the whole control of analysis as he gets notified about the number of classes that were analyzed along with their specific names.

As mentioned earlier we are interested in package and class analysis. Metrics calculator provides class level analysis, so in case a package level computation is selected, we have to aggregate the metrics for all classes that are contained in the same package. However, this issue will concern us in the following sections. Regarding Metrics Calculator we are interested in ten different metrics that we are using on the computation phase as quality indicators. Those metrics are presented on Table 4.2 along with a description, the quality attribute which they represent and the optimal value for each metric.

Table 4.2: Quality Metrics

Metric	Description	Quality Attribute	Best value
DIT	Depth of Inheritance Tree: Inheritance level number, 0 for the root class.	Inheritance	Highest
NOCC	Number of Children Classes: Number of direct sub-classes that the class has.	Inheritance	Highest
MPC	Message Passing Coupling: Number of send statements defined in the class.	Coupling	Lowest
RFC	Response for a Class: Number of local methods plus the number of methods called by local methods in the class.	Coupling	Lowest
DAC	Lack of Cohesion of Methods: Number of disjoint sets of methods (number of sets of methods that do not interact with each other), in the class.	Coupling	Lowest
LCOM	Data Abstraction Coupling: Number of abstract types defined in the class.	Cohesion	Lowest

WMP	Weighted Method per Class: Average cyclomatic complexity of all methods in the class.	Complexity	Lowest
NOM	Number of Methods: Number of methods in the class.	Size	Lowest
Size1	Lines of Code: Number of semicolons in the class	Size	Lowest
Size2	Number of Properties: Number of attributes and methods in the class	Size	Lowest

4.2.2 Computation Phase

Once the analysis phase has completed, we can proceed on computing the breaking point and producing the results. Primarily the user must select an already analyzed project from the dropdown menu, as well as a computation scope, either *package* or *class*. In order to provide a more abstract perspective of the methodology used, we define packages and classes as artifacts. Moreover, it is worth noting that the same algorithms are used independently of the computation scope that was selected by the user. The period of time that the computation phase lasts depends on two factors, the number of versions of the project history and the number of artifacts that each version consists of. Before analyzing in depth, the procedure, from the collection of the data to the extraction of the results, we provide a small overview of the basic steps:

1. Retrieval of versions and artifacts of each version based on the computation scope selected.
2. Retrieval of metrics both from SonarQube and Metrics Calculator databases.
3. Calculation of the resemblance for each artifact, between itself and the rest of the artifacts of its version.
4. Calculation of the optimal artifact based on the five most similar from the previous step.

5. Calculation of the Fitness Value, Principal and Interest based on the FITTED framework.
6. Calculation of the Breaking Point.

The first step of the computation phase is the retrieval of all the artifacts for each version of the project, both from SonarQube and from Metrics Calculator, from our database. However, we are processing those data in order to discard those artifacts, which have not been analyzed from both tools. Then, we are reaching anew our databases in order to retrieve the metrics needed in order to proceed. The metrics retrieved from SonarQube will be used on the third step, whereas the metrics from Metrics Calculator on the fourth step of the computation. The procedure, from step three to step six, takes place for each artifact of each version separately.

In order to calculate the interest for each artifact we must calculate the fitness value the function as described on section 3.2. The fitness value denotes the "distance" of the investigated artifact from the optimal artifact. Thus, we have to calculate the optimal values for each artifact investigated. To perform those calculations, we have to process the metrics retrieved on the second step.

Initially, we compute the five artifacts that mostly resemble the investigated artifact. To calculate this *similarity***,** we are using the similarity metrics retrieved from SonarQube and described on table 4.1. More precisely, we compare each artifact with every artifact included in the same version of the product. We apply the function at picture 4.7 for the six metrics from SonarQube and then we calculate the similarity as the average value of those calculations.

$$Similarity(\%) = \frac{abs\,(Metricvalu\ \ e_{class\ 1} - Metricvalu\ \ e_{class\ 2})}{\max(Metricvalu\ \ e_{class\ 1}, Metricvalu\ \ e_{class\ 2})} * 100$$

Picture 4.7: Similarity function

Once we have concluded with the calculation of similarity we must generate the optimal class. The optimal class is the result that accrues from selecting the best values of the ten-metrics described on table 4.2, among the investigated artifact and the most similar to it. At this point we are capable of calculating the fitness value from the following function:

$$fitness\ function = \frac{abs(metric_{investigated} - metric_{optimal})}{metric_{optimal}}$$

Picture 4.8: Fitness value function

The fitness function is applied on each one of the quality metrics which are aggregated, and the fitness value derives as the average of this calculation. At this point we should mention that we came up with an alternative approach for calculating the fitness value. The metric sqale index of SonarQube is described as *“the cost for fixing all the maintainability issues”*. The alternative way is to find the optimal Sqale Index among the investigated artifact and the five most resembling ones and apply the fitness function solely to this metric. Consequently, these calculations lead into two different values for the interest $i_{metrics}$ and i_{sqale}, therefore two separate values for the breaking point.

The next step is the calculation of principal and interest by applying the functions that we have been described on section 3.3 and 3.4 respectively. The principal emanates from the sqale index, which is described in the previous paragraph and the value of which is described in minutes. Therefore, we convert those minutes to hours and we multiply it by the average hourly salary of a developer. Both of the abovementioned types of interest are calculated by multiplying the average lines of code added per version with the fitness value.

At this point we should present an alternative rationale for calculating the breaking point. As the product evolves from version to version, some of the interest is being repaid. This accumulated interest must be subtracted from the total principal of the version that we are investigating in order to add more realism into our results. Hence, we conclude our calculations by producing four different types of breaking point, the effectiveness of which will be discussed at the results section. Those types are:

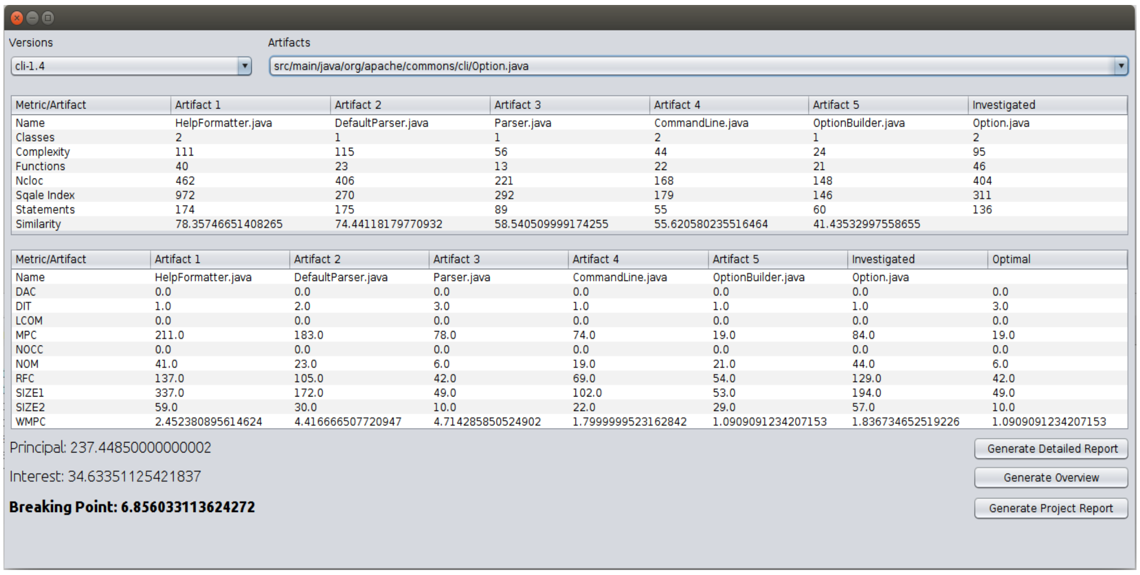
- Breaking Point$_{(metrics)}$: The breaking point in versions, which derives from the function at picture 3.6, with the fitness value being calculated based on the quality metrics presented on table 4.2.
- Breaking Point$_{(metrics\_minus\_sum\_interest)}$: The breaking point in versions, which derives from the function at picture 3.6, by subtracting from the principal, the interest that has been repayed on the previous versions of the software. The fitness value for both interest and past interest is calculated by the quality metrics.
- Breaking Point$_{(sqale)}$: The breaking point in versions, which derives from the function at picture 3.6, with the fitness value being calculated based only on Sqale Index which is presented on table 4.1
- Breaking Point$_{(sqale\_minus\_sum\_interest)}$: The breaking point in versions, which derives from the function at picture 3.6, by subtracting from the principal, the interest that has been repayed on the previous versions of the software. The fitness value for both interest and past interest is calculated based on Sqale Index.

4.2.3 Presentation Phase

The final phase of the breaking point tool is the presentation phase which is shown at picture 4.9. We should mention that this screenshot was taken after we concluded with the results of our case study, this it presents the final version of the product. At this screen the user can select any artifact from any version. For each artifact the user can observe the following results:

- The top 5 similar artifacts, along with the values for the similarity metrics, as well as the value of similarity between the investigated artifact and the most similar ones
- The values of the quality metrics, for the most similar artifacts and the investigated one
- The values of the quality metrics for the optimal class
- The principal of the investigated artifact in dollars
- The interest of the investigated artifact in dollars

- The breaking point of the investigated artifact in version



Metric/Artifact	Artifact 1	Artifact 2	Artifact 3	Artifact 4	Artifact 5	Investigated
Name	HelpFormatter.java	DefaultParser.java	Parser.java	CommandLine.java	OptionBuilder.java	Option.java
Classes	2	1	1	2	1	2
Complexity	111	115	56	44	24	95
Functions	40	23	13	22	21	46
Ncloc	462	406	221	168	148	404
Sqale Index	972	270	292	179	146	311
Statements	174	175	89	55	60	136
Similarity	78.35746651408265	74.44118179770932	58.540509999174255	55.620580235516464	41.43532997558655	

Metric/Artifact	Artifact 1	Artifact 2	Artifact 3	Artifact 4	Artifact 5	Investigated	Optimal
Name	HelpFormatter.java	DefaultParser.java	Parser.java	CommandLine.java	OptionBuilder.java	Option.java	
DAC	0.0	0.0	0.0	0.0	0.0	0.0	0.0
DIT	1.0	2.0	3.0	1.0	1.0	1.0	3.0
LCOM	0.0	0.0	0.0	0.0	0.0	0.0	0.0
MPC	211.0	183.0	78.0	74.0	19.0	84.0	19.0
NOCC	0.0	0.0	0.0	0.0	0.0	0.0	0.0
NOM	41.0	23.0	6.0	19.0	21.0	44.0	6.0
RFC	137.0	105.0	42.0	69.0	54.0	129.0	42.0
SIZE1	337.0	172.0	49.0	102.0	53.0	194.0	49.0
SIZE2	59.0	30.0	10.0	22.0	29.0	57.0	10.0
WMPC	2.452380895614624	4.416666507720947	4.714285850524902	1.7999999523162842	1.0909091234207153	1.836734652519226	1.0909091234207153

Principal: 237.44850000000002

Interest: 34.63351125421837

Breaking Point: 6.856033113624272

Picture 4.9: Presentation phase

Moreover, we provide the user with the ability to produce reports for the project that has been analyzed in order to get a more comprehensive picture of it. The buttons on the bottom right corner of the user interface on picture 4.9 produce .csv reports which can be further filtered by being converted into spreadsheets. The first button "Generate Detailed Report" produces a .csv file with the following details for each artifact, on each version of the project:

- Artifact name: The absolute path of the investigated artifact.
- Artifact type: Whether the selected unit is a concrete class or package
- Version ID: The version of the artifact to which the selected unit belongs
- Principal: The calculated principal
- AverageK: The average lines of code added in each version.
- RealK: The real numbers of lines of code added in the artifact from its previous version to the current.
- Fitness Value (metrics): The fitness value calculated by using the quality metrics
- Fitness Value (sqale): The fitness value calculated by using sqale index

- Interest (metrics): The value of the accumulated interest for the artifact calculated using the metrics approach
- Interest (sqale): The value of the accumulated interest for the artifact calculated using the sqale index approach
- The 4 kinds of breaking point that have been described on subsection 4.2.2

With this kind of report, we can observe the evolution of each artifact individually and we can investigate distinct kinds of behavior during the successive versions of the software. The *"Generate Overview"* button produces statistics regarding with the evolutionary factor of principal, interest and breaking point, by defining this behavior as *"Incremental"*, *"Decremental"* and *"Stable"*. Finally, the third button *"Generate Project Report"* created an output file with the value of breaking point generally for each version of the product.

5 Case Study

In order to evaluate and validate the FITTED framework, we performed a case study on open source software projects. The main purpose for selecting open source artifacts, is the vast amount of data available on those repositories, as well as the convenience on accessing successive versions of the software.

5.1 Goal and Research Questions

The goal of the case study, is to analyze the FITTED approach for the purpose of evaluation, concerning its ability to (a) accurately estimate occurrence of breaking point, and (b) predict the effect of repayment activities on breaking point move in time. According to the aforementioned goal we have derived three research questions (RQ) that will guide the case study design and the reporting of the results:

***RQ$_1$**: Which breaking point calculation approach produces the most reliable results?*

This research question aims at identifying, which of the four breaking point calculation approaches, better depicts the evolution of the breaking point among successive software versions. We will empirically evaluate the results, to find out which approach produces the most stable and accurate results and therefore will be used as the basic breaking point calculation method on the following research questions, as well as in future research.

***RQ$_2$**: How do TD principal and interest evolve during successive software versions?*

Research question 2 aims at monitoring technical debt, as a part of technical debt management. The evolution of principal and interest during the development and maintenance of software is of crucial importance in FITTED framework, as it can lead to the identification of the breaking point. Regarding interest, we expect it to grow exponentially, due to system's increasing size and complexity. Concerning principal, we have to point out that although in our analysis of the FITTED framework, we have defined it as an amount of effort saved once while taking on technical debt and considered it as increasing through software evolution, in this study we have the ability to estimate principal as software grows and therefore reach a more accurate assessment of the version where breaking point occurs.

RQ$_3$: *What is the sustainability of software assets along evolution and which factor (principal or interest) is more important?*

This research question is going to investigate whether software development and maintenance affect the version in which the breaking point occurs and, therefore, to evaluate the software's sustainability. The fundamental aim is to examine if the breaking point moves backwards or forward in time as the software project develops. More specifically, research question 3 will evaluate the effect of principal and interest evolution on the progress of the breaking point and examine if one of the two factors is more important for the system's sustainability.

5.2 Case Selection

To collect data for our case study, we retrieved and analyzed sets of successive versions of 5 open source projects. In particular, we selected projects for which development data for many releases were available, in order to better monitor the evolution of breaking point. Table 5.1 depicts these projects and the number of releases of which they constitute.

Table 5.1: Analyzed projects

Project Name	Number of Versions
Commons-cli	20
Commons-lang	32
Joda-time	54
Wro4j	50
Xstream	29

5.3 Data Collection

5.3.1 Collected Data

To answer the research questions mentioned in section 5.1, we performed analysis on each one of the aforementioned projects by generating the reports that have been described on subsection of 4.2.3 at the presentation phase of the tool.

5.3.2 Collection Process

To collect the data required for our study, we used the Breaking Point Tool with a class level computation for all projects. After extensive discussion upon the methodology that we should follow for collecting the required information, we separated the process into three phases.

- **Analysis Phase**: As already mentioned, in order to gather the data required for our study, we had to combine statistics accrued from a couple of analysis tools. This time-consuming and asynchronous task, prompted us on completing the analysis phase for the selected artifacts before proceeding further. The analysis phase was completed, upon collecting both the similarity and maintenance metrics, from Sonar and Metrics Calculator respectively, for every successive version, for each project listed on table 5.1.

- **Unit Selection Phase**: Each project is comprised of a plethora of artifacts (classes or packages) which we could choose, to monitor the breaking point. In order to provide accurate results, we had to choose between units that had a core role in the functionality of each project. Thus, we excluded classes or packages that consist the model layer of a project such as POJOs (Plain Old Java Object) or DAOs (Data Access Object), because mostly the maintenance effort required is trivial. For instance, a POJO is nothing more than a Class constituted of fields and accessor methods. The effort of adding or deleting code on those units is considered minimal. Hence, we tried to monitor units that consist the core of each project, such as units whose contents were being modified from version to version in order to be improved in terms of performance, or extend their functionality in order to support new features.
- **Evaluation Phase**: This phase concerns the interpretation of the results, as those are retrieved from the breaking point tool. The evaluation is done either empirically or practically regarding the nature of the research question to be answered and the results are examined both on project and artifact level. In order to illustrate the results and cite the outcome of our research we are using tables and charts such as pies and diagrams.

6 Results

In this section we present the answers to the Research Questions stated earlier, as a synthesis of the results emanated from the breaking point tool.

In this section we are going to present the answers to the Research Questions stated earlier, as a synthesis of the results produced from the breaking point tool.

6.1 RQ_1 Breaking Point Selection

In order to answer to this research question, we observed the evolution of the breaking point during successive version of the software. We investigated several classes from each project, so that we could produce safe assumptions.

Initially, we discarded both the approaches which were using sqale index as the factor for calculating the Fitness Function. In case that the investigated artifact is the optimal one (in terms of sqale index) the Fitness Function (picture XX) produces a value of 0. Moreover, the interest is 0 and the division between principal and interest for the calculation of the breaking point produces and infinite result. As the software evolves and the maintenance issues are being fixed, the investigated artifact might become the optimal one between itself and the five most similar to it. Consequently, we are unable to monitor the evolution of technical debt as the infinite results might interfere the sequence of the evolution of the breaking point.

This scenario is verified on table 6.1 which represents the evolution of the breaking point during the successive versions of class StrTokenizer.java from project Commons-Lang. If we take a closer look at the fitness value, we observe fluctuation of the fitness value among the versions which obviously affects the breaking point. From version 3_1 to 3_2_RC1 some modification of the class increased its distance from the optimal one from 0.28 to 0.7. Consequently, the breaking point will appear from 3 version to 1 version of the product. Some versions later from 3_4 to 3_5_RC1 the investigated class reaches the optimal as its fitness value decreases from 0.7 to 0.26 and the breaking point moved further in time from 2 versions to 5.5. However, in the next version we detect the downside of this approach. At version 3_6_RC1 StrTokenizer.java becomes the class with the lowest sqale index among its competitor classes. Probably this occurs because of the modifications that take place on the project as a whole and not only on the investigated artifact. At this point we are unable to further monitor the evolution of the breaking point which is critical for our research and actually this is the main reason for rejecting this approach.

The downside of this approach, comparing to the metrics one, is that, as sqale index is the only factor of calculating the Fitness Function, the possibility for the investigated artifact to be the optimal one reaches 16.6% (1 out of 6). In the alternative approach, the metrics one, the optimal artifact is calculated based on 10 different metrics, thus it is quite impossible for an artifact to be dominant on all those metrics. This actually happens because the metrics we have selected are a mixture of coupling, cohesion and size, so an artifact with high low coupling might lack on cohesion and the opposite. We should note that in our results we have not investigated such an artifact, dominant on all metrics.

Table 6.1: Sqale Index fitness value approach

Version	Fitness Value (Sqale)	Breaking Point (Sqale)	Breaking Point minus Sum Interest (Sqale)
3_7	0	Infinity	-Infinity
3_6	0	Infinity	-Infinity
3_6_RC3	0	Infinity	-Infinity
3_6_RC2	0	Infinity	-Infinity
3_6_RC1	0	Infinity	-Infinity
3_5	0,266055046	5,424891805	-39,63338621

3_5_RC1	0,266055046	5,249895295	-38,35488988
3_4	0,703703704	1,918708715	-14,01777699
3_4_RC2	0,703703704	1,852546346	-13,53440537
3_4_RC1	0,703703704	1,786383976	-13,05103375
3_3_2	0,703703704	1,720221607	-12,56766213
3_3_2_RC1	0,703703704	1,654059237	-12,08429051
3_3_1	0,703703704	1,587896868	-11,60091889
3_3_1_RC1	0,703703704	1,521734498	-11,11754727
3_3	0,703703704	1,455572129	-10,63417565
3_3_RC1	0,703703704	1,389409759	-10,15080403
3_2_1	0,703703704	1,32324739	-9,667432409
3_2_1_RC1	0,703703704	1,25708502	-9,184060789
3_2	0,703703704	1,190922651	-8,700689168
3_2_RC2	0,703703704	1,124760281	-8,217317548
3_2_RC1	0,703703704	1,058597912	-7,705756081
3_1	0,28057554	3,236787023	-17,53922011
3_1_RC1	0,28057554	3,021001221	-16,36993877
3_0_2_RC1	0,287769784	2,75045068	-14,80527535
3_0_1	0,287769784	2,538877551	-13,66640801
3_0_1_RC2	0,287769784	2,327304422	-12,52754068
3_0_1_RC1	0,287769784	2,115731293	-11,37419473
3_0	0,467625899	1,337631298	-6,145930406
3_0_RC4	0,467625899	1,189005598	-5,46304925
3_0_RC3	0,467625899	1,040379898	-4,780168093
3_0_RC2	0,369127517	1,129710845	-5,187885353
3_0_RC1	0,375838926	Infinity	Infinity

By discarding the sqale index factor, we have been left with the two metrics approaches, but a declination on the results that we wanted to present, had us reject the approach of subtracting the already re-payed effort. Before getting into the assumptions we should add a reminder for our approach. In this approach we have been subtracting from the principal, the amount of money that was re-payed in the previous version on the artifact. However, what we observed is that in many cases the repayment effort that has been done was higher than the principal of the current version. The result was a negative principal and consequently a misleading breaking point. This behavior is observed on versions that many lines of code have been added from one version to its successive.

Table 6.2: Metrics fitness value approach

Version	Fitness Value (Metrics)	Breaking Point (Metrics)	Breaking Point minus Sum Interest (Metrics)
3_7	0,651073394	3,687710343	-279,8506024
3_6	0,649886681	3,594726768	-272,7943243
3_6_RC3	0,649886681	3,492020289	-265,0002008
3_6_RC2	0,649886681	3,38931381	-257,2060772
3_6_RC1	0,669442237	3,190599895	-242,1110186
3_5	0,719442237	2,006165005	-219,293533
3_5_RC1	0,719442237	1,941450005	-212,2195481
3_4	0,852468838	1,583873063	-173,1328774
3_4_RC2	0,852468838	1,52925675	-167,1627782
3_4_RC1	0,852468838	1,474640438	-161,192679
3_3_2	0,87128394	1,389359151	-151,870597
3_3_2_RC1	0,87128394	1,335922261	-146,0294202
3_3_1	0,87128394	1,282485371	-140,1882434

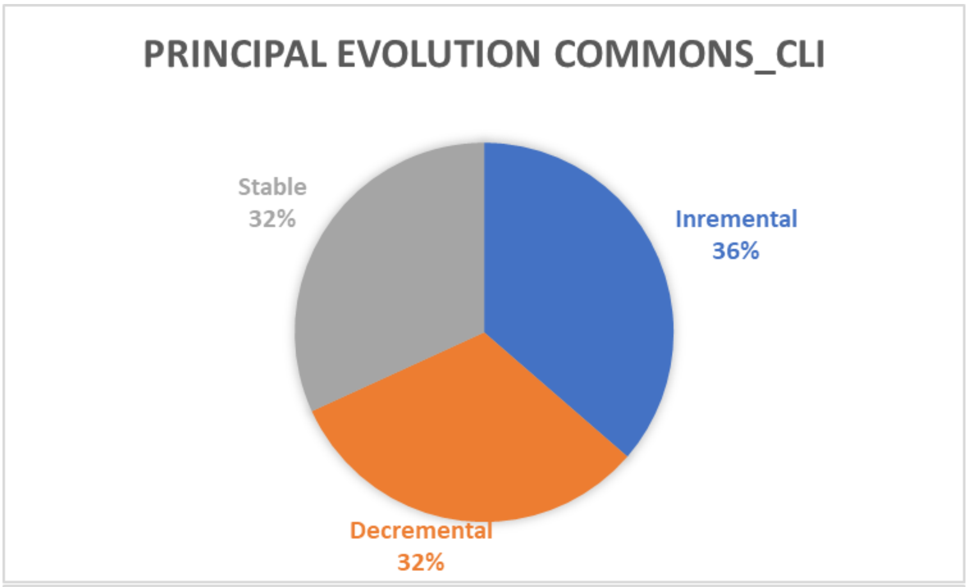
3_3_1_RC1	0,87128394	1,22904848	-134,3470666
3_3	0,87128394	1,17561159	-128,5058898
3_3_RC1	0,87128394	1,122174699	-122,664713
3_2_1	0,851146954	1,094022701	-119,5874231
3_2_1_RC1	0,851146954	1,039321566	-113,6080519
3_2	0,851146954	0,984620431	-107,6286808
3_2_RC2	0,851146954	0,929919296	-101,6493096
3_2_RC1	0,851146954	0,87521816	-95,61023812
3_1	0,718701783	1,263616268	-106,7354068
3_1_RC1	0,718701783	1,179375183	-99,619713
3_0_2_RC1	0,718701783	1,101286538	-92,49786678
3_0_1	0,718701783	1,016572189	-85,38264626
3_0_1_RC2	0,718701783	0,93185784	-78,26742573
3_0_1_RC1	0,718701783	0,84714349	-71,14367975
3_0	0,687690525	0,909582167	-66,91466199
3_0_RC4	0,687690525	0,808517482	-59,47969954
3_0_RC3	0,688798197	0,706315127	-51,96104284
3_0_RC2	0,683920148	0,609731063	-44,85302616
3_0_RC1	0,683920148	Infinity	Infinity

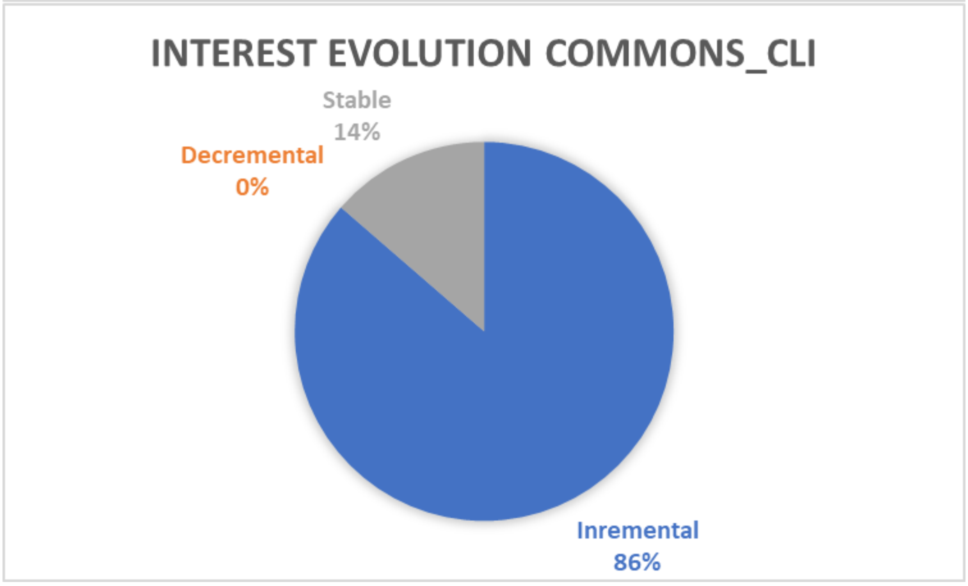
6.2 RQ_2 Evolution of principal and interest

To answer in this research question, we have gathered data for each one of the investigated projects which depict the evolution of principal and interest from the first to the last version of the project. The behavior observed on evolution of those values is either incremental, decremental or stable. The stable result is basically observed on artifacts that are never modified on the evolution of the software. An example of such artifacts are classes that represent exceptions which probably will not be modified further by the time that they are created.

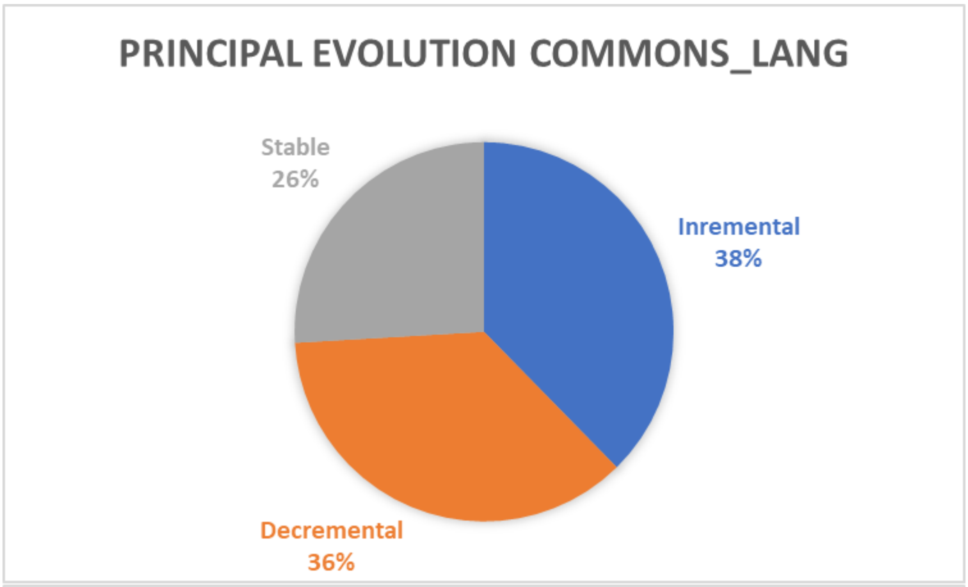
Concerning the evolution of principal, for Commons-cli and Commons-lang we observe a balance as all types are represented by percentages around 30%, at pictures 6.1 and 6.2, however the incremental behavior is the most dominant. The pie charts for Wro4j and Xstream, at pictures 6.3 and 6.4, clearly illustrate an incremental behavior on the principal while for Joda-Time the principal declines as the project evolves, picture 6.5. Aggregated on an artifact level, at table 6.3, 48% of the artifacts are hardly maintained, 36% present an improvement on their code quality while the rest of the artifact remain untouched. We can assume that only for one out of five projects the maintainability issues are fixed along the evolution, however as the sample of investigated projects increased this percentage will sure be declining.

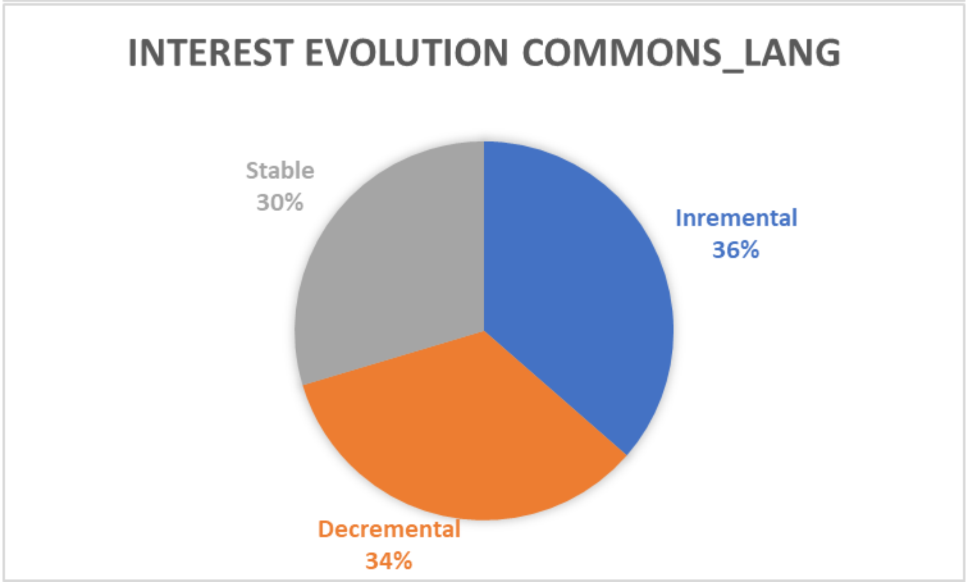
Regarding the evolution of interest, we reached our goal as our calculations show an exponential increase. All the investigated project has an incremental behavior in terms of interest, while on an artifact level 71.8% of the artifacts adopt the same behavior, table 6.4. A fact that is worth to be discussed is that none of the artifacts produces a decremental behavior on two of the open source projects, Commons-cli and Xstream.



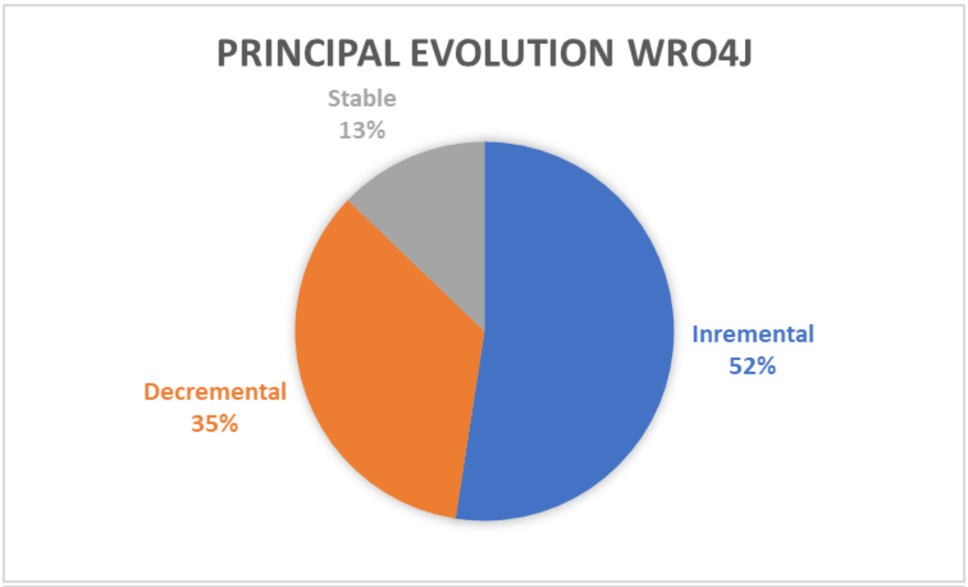


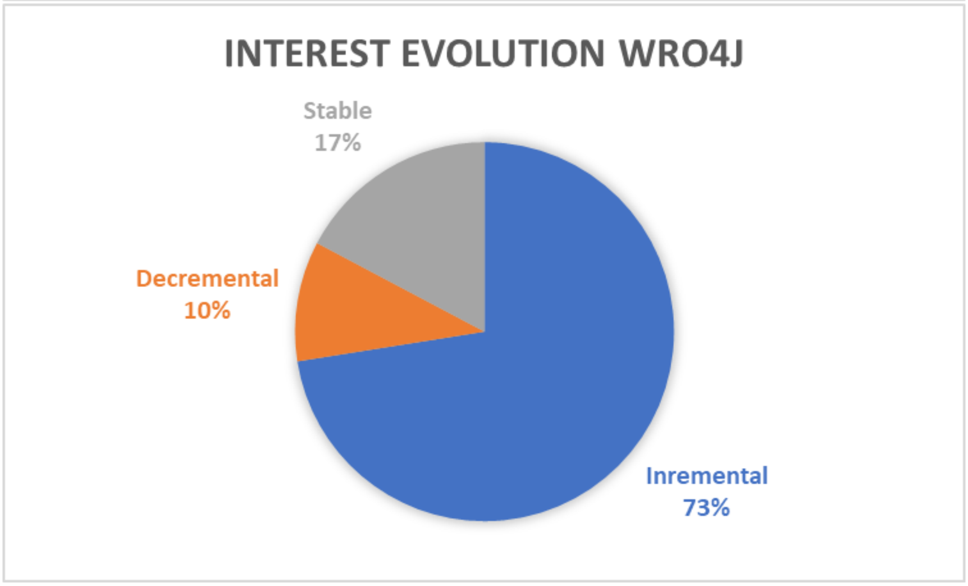
Picture 6.1: Principal and interest evolution for Commons-cli



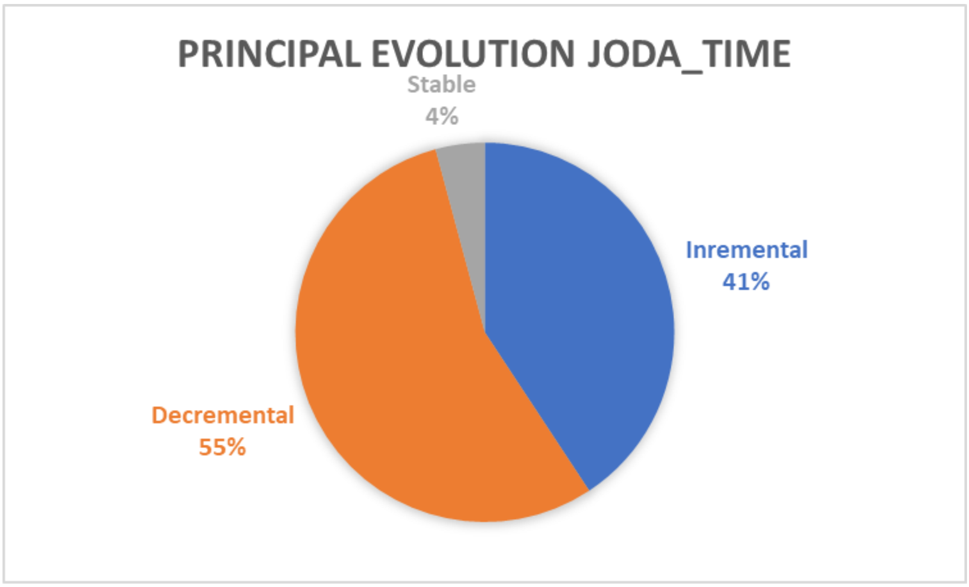


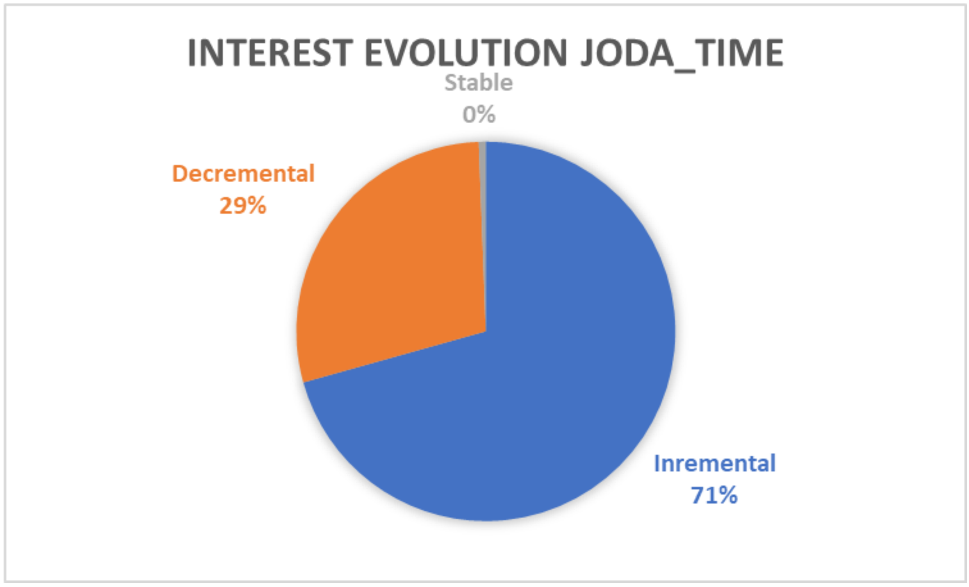
Picture 6.2: Principal and interest evolution for Commons-lang



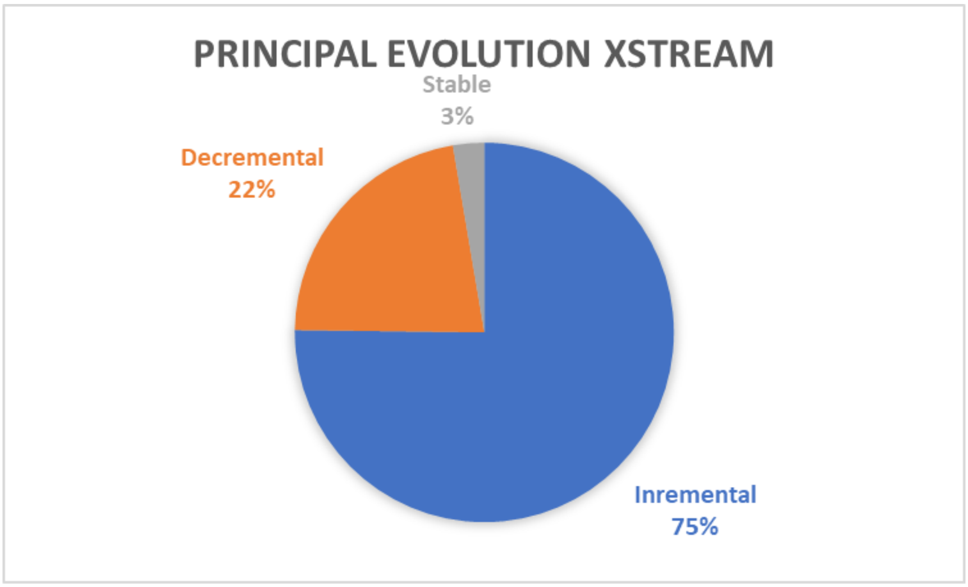


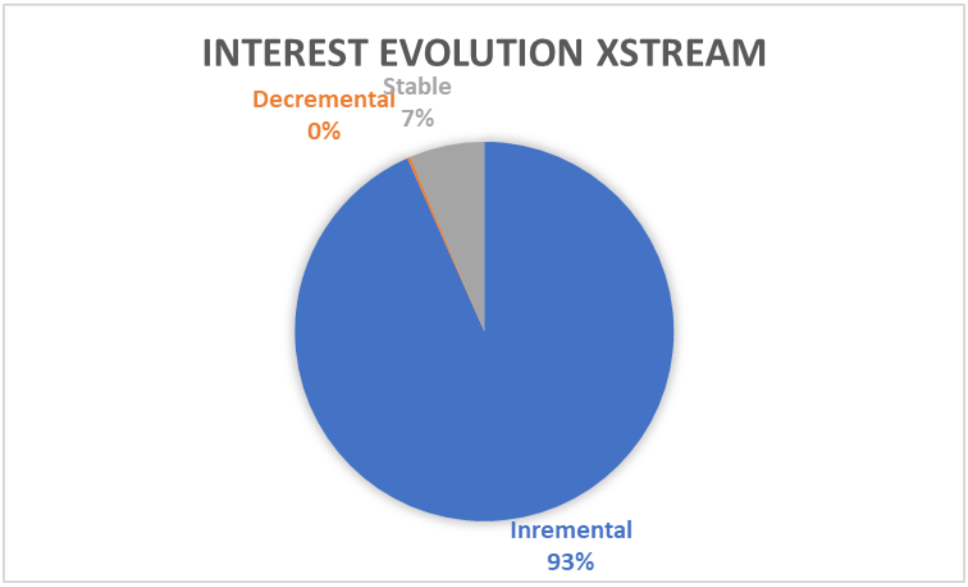
Picture 6.3: Principal and interest evolution for Commons-wro4j





Picture 6.4: Principal and interest evolution for Joda time





Picture 6.5: Principal and interest evolution for x-stream

Table 6.3: Principal evolution overview

	Incremental	Decremental	Stable	Behavior
Commons-Cli	36%	32%	36%	INCREMENTAL
Commons-Lang	38%	36%	26%	INCREMENTAL
Joda-Time	41%	55%	4%	DEREMENTAL
Wro4j	52%	35%	13%	INCREMENTAL
Xstream	75%	22%	3%	INCREMENTAL
	48%	36%	16%	

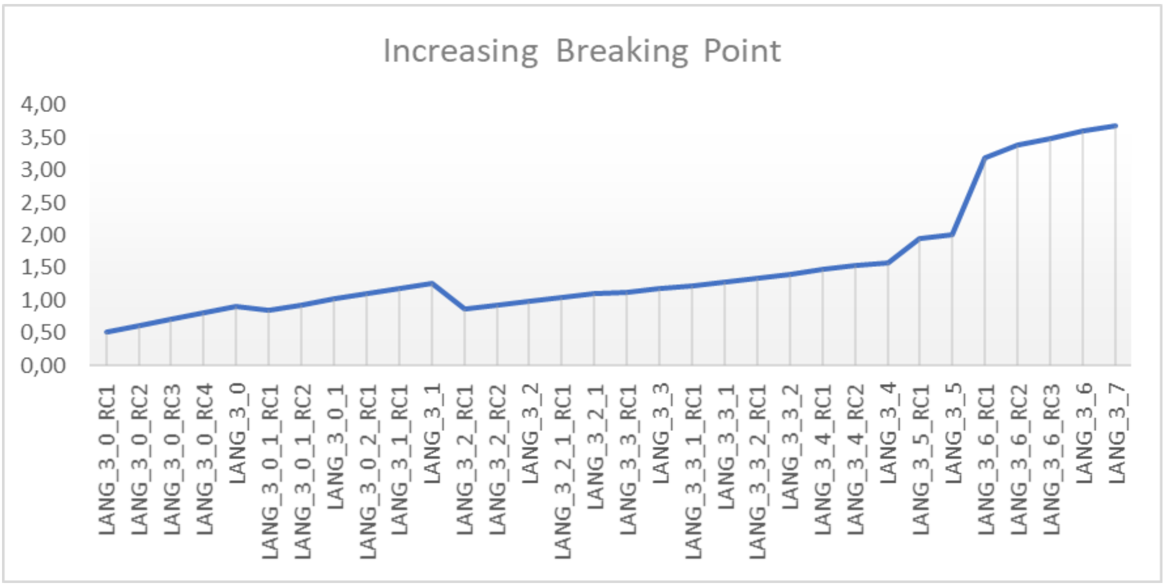
Table 6.4: Interest evolution overview

	Incremental	Decremental	Stable	Behavior
Commons-Cli	86%	0%	14%	INCREMENTAL
Commons-Lang	36%	34%	30%	INCREMENTAL
Joda-Time	71%	29%	0%	INCREMENTAL
Wro4j	73%	10%	17%	INCREMENTAL
Xstream	93%	0%	7%	INCREMENTAL
	71.8%	14.6%	13.6	

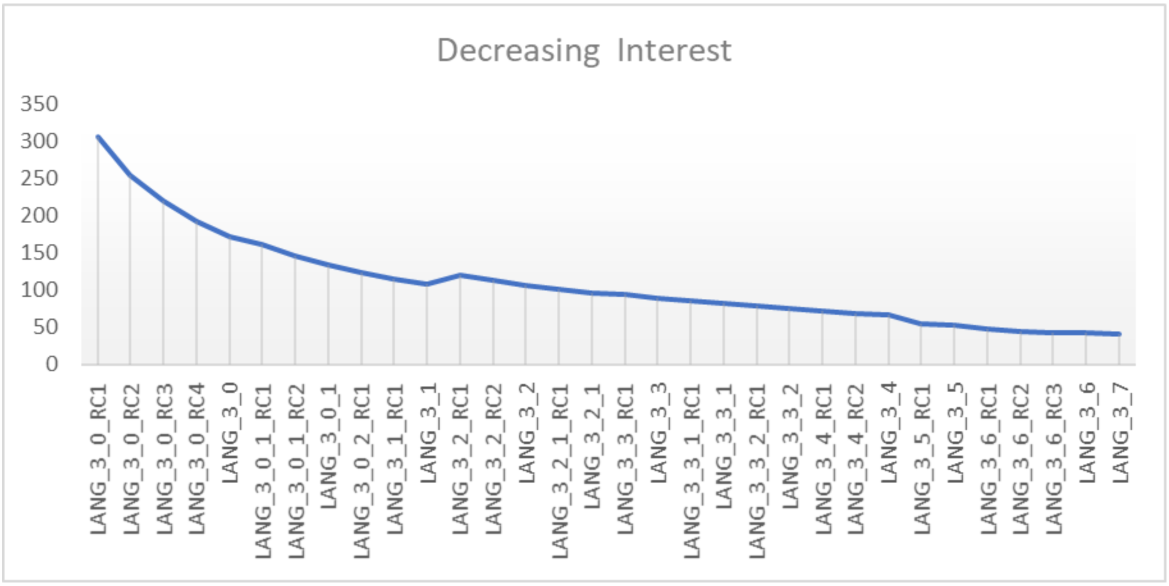
6.3 RQ$_3$ Evolution of principal and interest

To answer research question three, we followed the same strategy. We have created pie charts which represent the behavior of the artifacts for each one of the investigated projects. Moreover, we present graphs which monitor the evolution of the breaking point, as a factor of principal and interest, and depict the importance of interest for its calculation.

Pictures 6.6 and 6.7 represent the behavior of StrTokenizer.java from Commons-lang project of Apache Foundation. This class is a typical example of an artifact, whose breaking point moves to the future because the interest decreases. A better interpretation of our model denotes that StrTokenizer.java reaches the values of the optimal class during the evolution of the project. More specifically, from version LANG_3_0_RC to LANG_3_1 the interest decreases from 300$ to 120$, thus the breaking point moves from 0.5 versions, which actually denotes the next version of the software, to 1.2 versions. However, the addition of code on version LANG_3_1 generates new technical debt items, the interest slightly grows and the breaking point returns to its previous levels. At this point we observe a continuous improvement on the quality of the artifact until the last investigated version of the artifact. The interest drops to about 40$ and the breaking point will appear at 3.6 versions. We should also mention that the principal of this class remains almost stable as it ranges from 150$ to 155$.

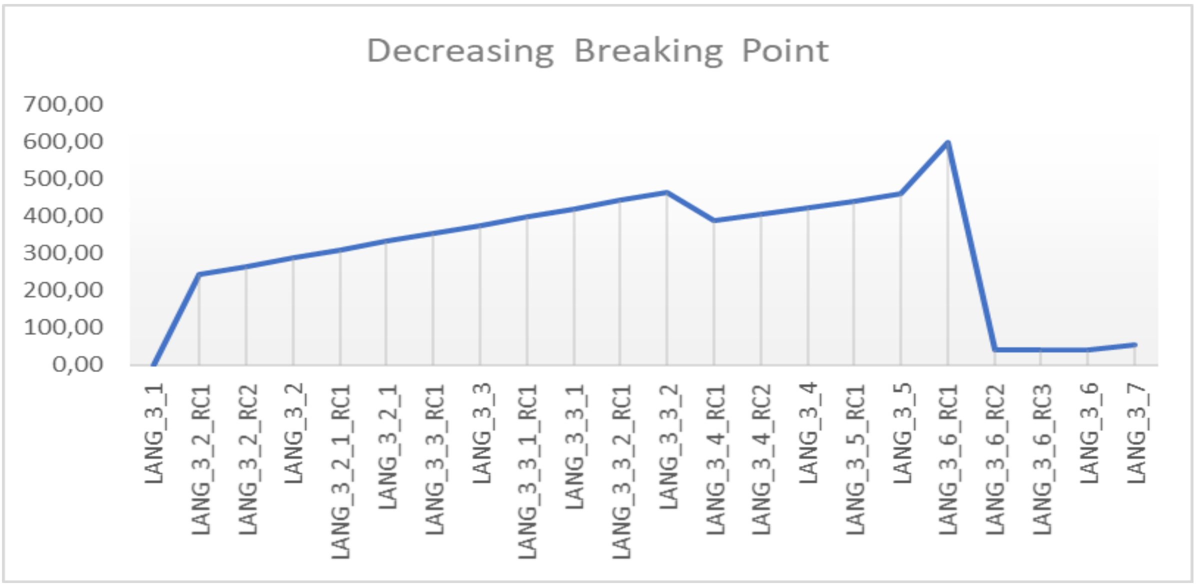


Picture 6.6: Increasing breaking point

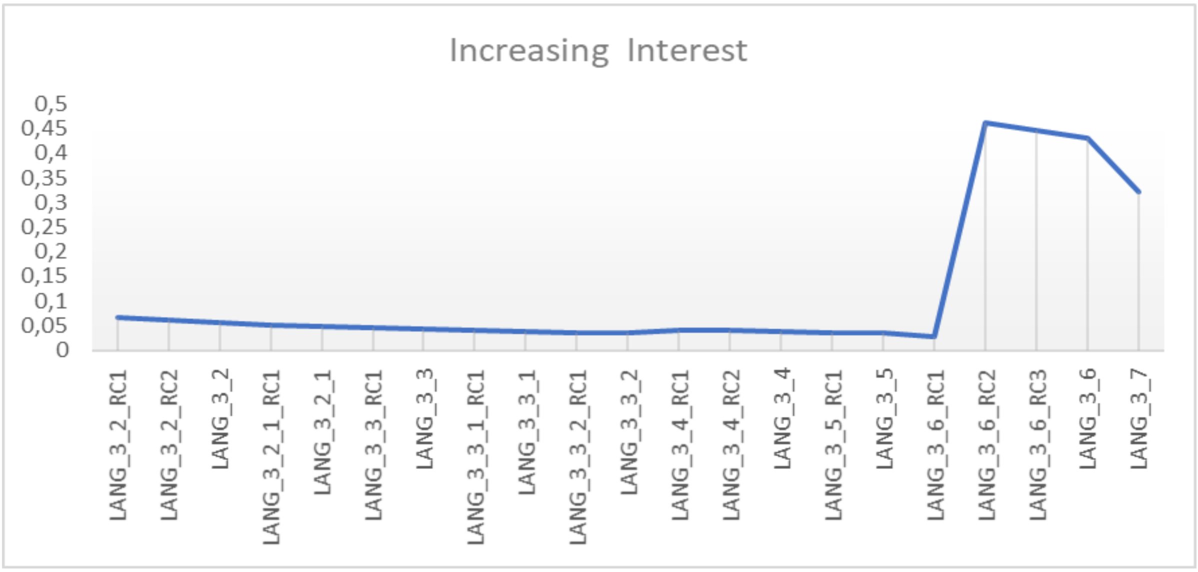


Picture 6.7: Decreasing interest

ImmutablePair.java is an almost optimal class as the representing graphs at pictures X and Z present. This can be confirmed by the values of both breaking point and interest. The first one reaches its peek at 500 on LANG_3_6_RC1 while the interest is almost zero. The fact that we want to prove by presenting this example, is the direct dependence of breaking point to the respective interest. The addition of code on version LANG_3_6_RC_2 causes interest to grow on almost 0.5$, and as the principal remains almost stable, the breaking point reduces from 500 versions to 20.



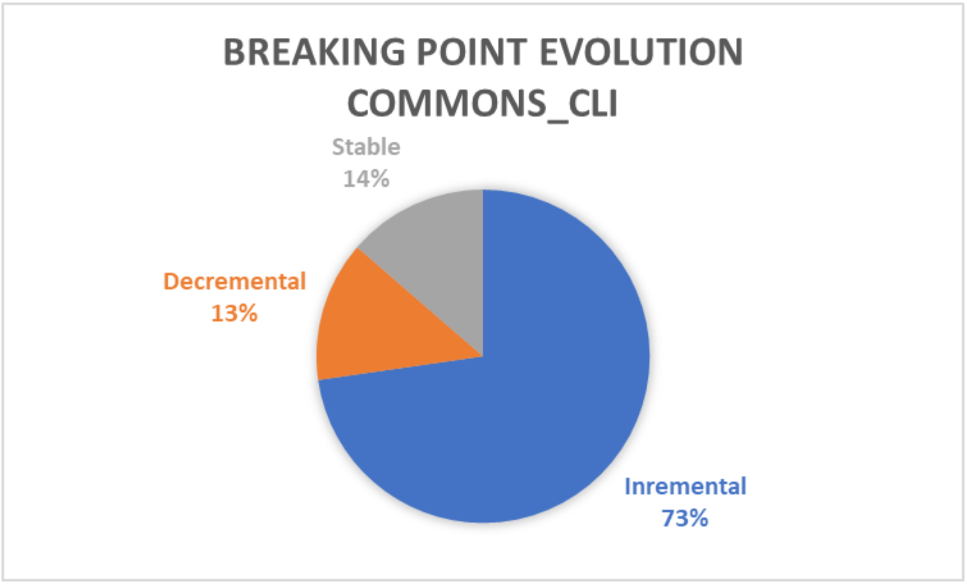
Picture 6.8: Decreasing breaking point



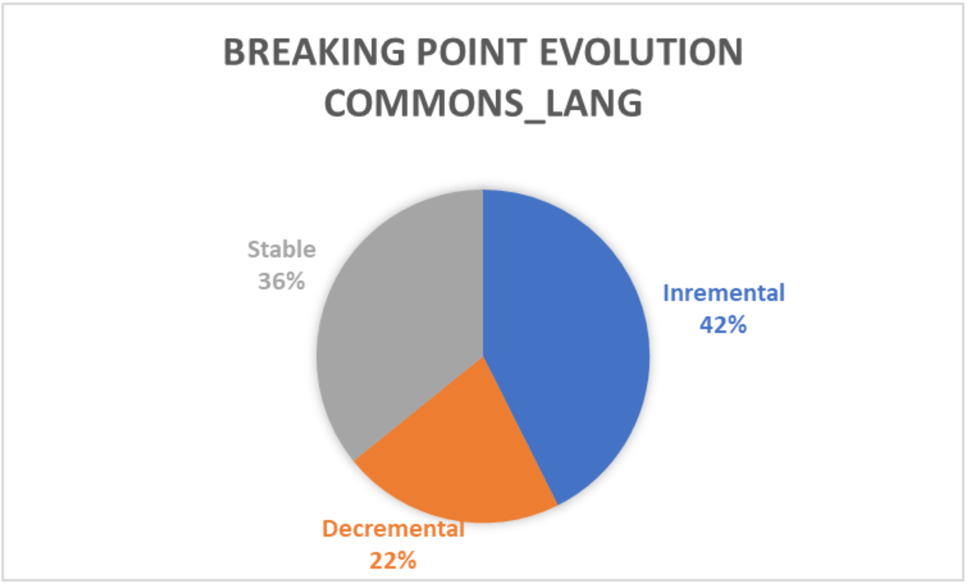
Picture 6.9: Increasing interest

Regarding the combination of interest and principal in the breaking point, we can observe that the examined cases are divided, some of the producing incremental breaking point (which is a sign of good quality), whereas others decremental breaking point, which suggests systems with worse quality.

Commons-cli on picture 6.10 and Commons-Lang on picture 6.11 are the two projects with an overall incremental behavior in terms of breaking point. Specifically, for commons-cli, the aggregation of incremental and stable items reaches the 87% of the artifacts, whereas only 13% produce a loss of quality over the versions. Those percentages for commons-lang reach 78% and 22% respectively. Given the fact that those two systems are part of the Apache ecosystem, which is famous for the great quality of products they deliver, the result can be considered intuitive.

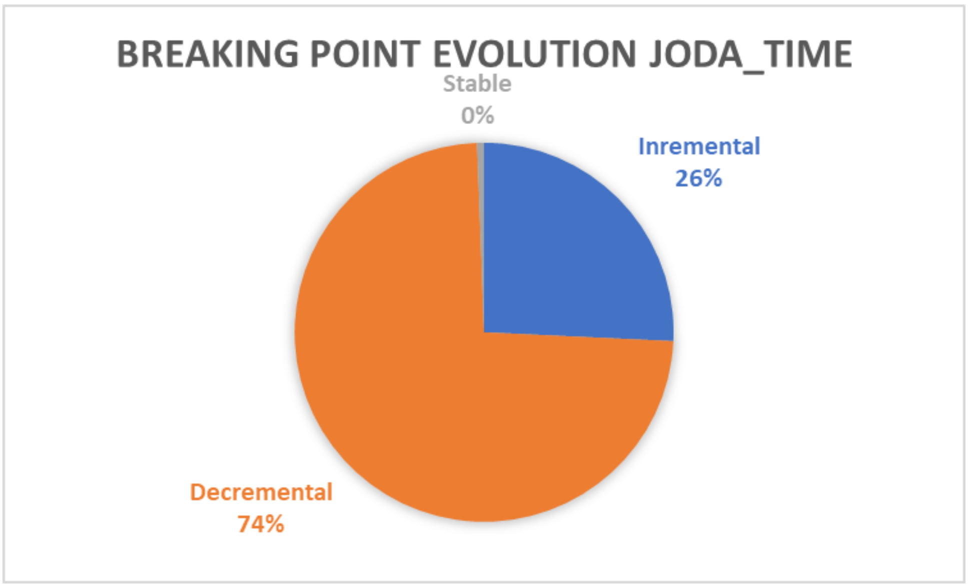


Picture 6.10: Breaking point for Commons-cli

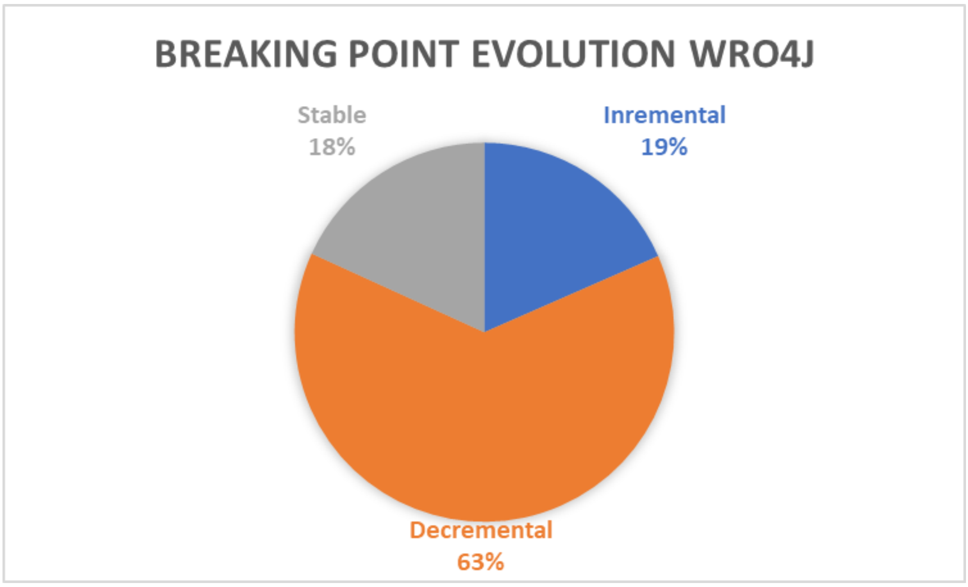


Picture 6.11: Breaking point for Commons-cli

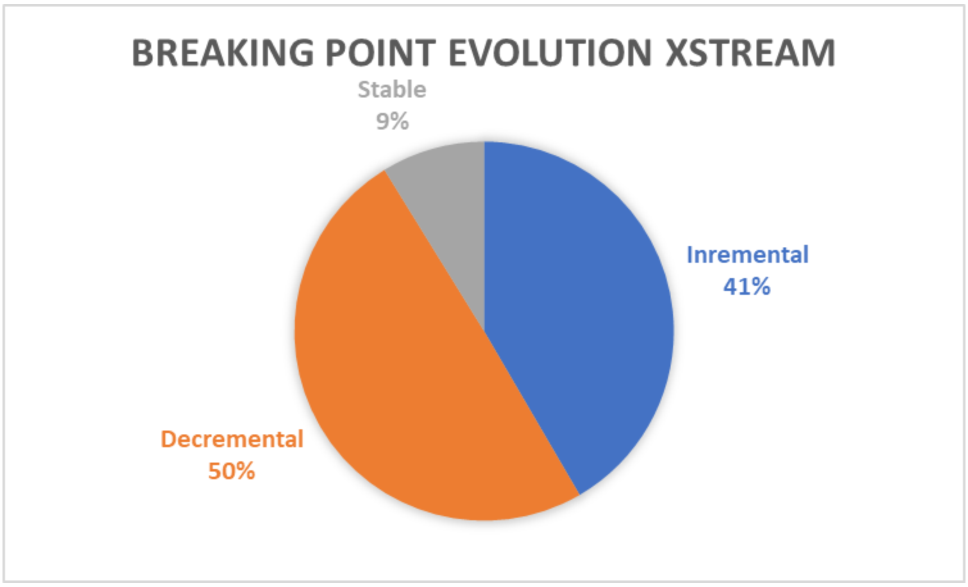
The rest of the projects produce negative results which can be depicted on pictures 6.12, 6.13 and 6.14. Xstream can be characterized for its medium quality as positive and negative values share the same percentage on 50%. However, Joda-time and Wro4j produce defective results, as the decremental behavior is overwhelming, as it reaches 74% and 63% respectively. It is worth mentioning that on Joda-time, no item with a stable breaking point does exist. This fact means that all the classes that represent the project are being modified from version to version. Each modification generates new technical debt costs, which are not re-payed on their majority. This fact results into poor quality and increased maintenance costs.



Picture 6.12: Breaking point for Joda-time



Picture 6.13: Breaking point for Wro4j



Picture 6.14: Breaking point for Xstream

Concluding on this research question, we can assume that the most important factor between principal and interest in the calculation of the breaking point is the second value. Interest represents the distance from the optimal, evolves or diminishes each time that a code modification takes place on the artifact and actually it leverages the final result. Regarding the sustainability of the artifacts, as table 6.5 depicts, the three values are sharing almost the same percentages. However, those values depend on the quality of the projects examined as we have already mentioned. Hence, we can conclude to the fact that the breaking point, apart from a metric that denotes the beginning of the financial loss for the software company, could effectively be a metric for characterizing the quality of the project.

Table 6.5: Breking point evolution overview

	Incremental	Decremental	Stable	Behavior
Commons-Cli	73%	13%	14%	INCREMENTAL
Commons-Lang	42%	22%	36%	INCREMENTAL
Joda-Time	26%	74%	0%	DECREMENTAL
Wro4j	19%	63%	18%	DECREMENTAL
Xstream	50%	50%	9%	DECREMENTAL
	36%	37%	27%	

7 Conclusions

Nowadays, technical debt is receiving increasing interest by both academia and practitioners, leading to an explosion of studies in this field. The purpose of this study was (a), to evaluate FITTED, a framework for managing interest in technical debt, and (b) to monitor the evolution of the breaking point during successive versions of software, by applying FITTED on open-source Java projects.

We can assume that our case study, confirmed the FITTED framework theory. Primarily we observed an incremental behavior for the principal which confirms that as software evolves artifacts generate technical debt items. However, the decremental behavior concentrated a considerable amount which must be taken into account for our future work, as it indicated the repayment effort. Moreover, concerning interest, the assessment that interest accumulates exponentially was also confirmed overwhelmingly. Another important conclusion is that as principal fluctuates very close to its initial value, interest is the variable that leverages the breaking point. The decrease of interest causes the breaking point to move towards the distant future, whereas its increase denotes that the loss period will begin sooner.

Finally, we can characterize the breaking point, as variable with multidisciplinary nature such as technical debt itself. It can be interpreted as a variable which (a), indicates the quality of the software it describes and (b) denotes the number of future versions when the loss period will begin for the company. Thus, it can be used by software developers and project managers respectively.

8 Future Work

Regarding our future work, initially we plan on analyze more open source projects in order to further validate the breaking point tool and evaluate the results that our current study produced. Moreover, we plan on extending our research field into industry. We will approach companies which are willing to share their Java projects, or mobile Android applications in order to analyze software on a more realistic factor. In particular, we will ask from the developers to list a number of artifacts which, are difficult to be maintained or extended, thus will produce technical debt, and we will compare those with our findings. Finally, in the distant future, we plan on releasing the breaking point tool as an open source project, for use by both researchers and practitioners.

Bibliography

[1] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman, "Managing Technical Debt in Software Engineering": Dagstuhl Seminar April 17–22, 2016.

[2] Zengyang Li, "Managing Technical Debt in Software Architecture", Institute of Mathematics and Computing Science, University of Groningen, June 2015.

[3] Zengyang Li, Paris Avgeriou, and Peng Liang. "A Systematic Mapping Study on Technical Debt and Its Management", Journal of Systems and Software, 101(3):193–220, 2015.

[4] Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou, "The financial aspect of managing technical debt: A systematic literature review," Information and Software Technology, Elsevier, vol. 64, pp. 52–73, Aug. 2015.

[5] Areti Ampatzoglou, Apostolos Ampatzoglou, Paris Avgeriou, and Alexander Chatzigeorgiou. "Establishing a framework for managing interest in technical debt". In 5th International Symposium on Business Modeling and Software Design (BMSD). SciTePress, 2015.

[6] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, "Estimating the Breaking Point for Technical Debt", 7th International Workshop on Managing Technical Debt (MTD '15), IEEE Computer Society, 2015.

[7] Zengyang Li, Paris Avgeriou, and Peng Liang, "A Systematic Mapping Study on Technical Debt and Its Management. Journal of Systems and Software", 101(3):193–220, 2015.

[8] Areti Ampatzoglou, Apostolos Ampatzoglou, Paris Avgeriou, and Alexander Chatzigeorgiou. "A Financial Approach for Managing Interest in Technical

Debt”. In Lecture Notes in Business Information Processing, pages 117–133. Springer, 2016.

[9] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman, “Identification and management of technical debt: A systematic mapping study,” Information and Software Technology, vol. 70, pp.100–121, 2016.

[10] Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, “A Financial Approach for Managing Interest in Technical Debt”, LNBIP, Springer, vol. 257, pp. 117-133, 2016.

[11] Mishkin F., Eakins S., “Financial Markets and Institutions”, 7th edn. Prentice Hall, Upper Saddle River (2012)

[12] V. R. Basili, G. Caldiera, and H. D. Rombach, “Goal Question Metric Paradigm”, Encyclopedia of Software Engineering, John Wiley, 1994.

[13] Wake, W.C.: Refactoring Workbook, 1st edn. Addison-Wesley Professional, Boston (2003).

[14] Z. Codabux and B. Williams, “Managing technical debt: An industrial case study,” 4th International Workshop on Managing Technical Debt (MTD’ 13), 2013, pp. 8–15.

[15] W. Cunningham, “The WyCash Portfolio Management System,” Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum), New York, NY, USA, 1992, pp. 29–30.

[16] U.S. Bureau of Labor Statistics, “National Occupational Employment and Wage Estimates,” United States, Mar. 2015

[17] L. Prechelt, “An Empirical Comparison of Seven Programming Languages,” Computer, vol. 33, no. 10, pp. 23–29, Oct. 2000.

[18] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, “Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt,” 10th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 2015, pp. 50–60, 2015.

[19] D. Feitosa, A. Ampatzoglou, P. Avgeriou, and E. Y. Nakagawa, “Investigating Quality Trade-offs in Open Source Critical Embedded Systems”, Quality of Software Architectures (QoSA’ 15), 2015.

[20] B. Kitchenham and S. L. Pfleeger, “Principles of Survey Research Part 2: Designing a survey”, Special Interest Group on Software, ACM, 27 (1), pp. 18-20, January 2002.

[21] P. Kruchten, R. L. Nord, I. Ozkaya, "Technical Debt: From Metaphor to Theory and Practice", Software, IEEE Computer Society, vol. 29, no. 6, pp. 18-21, November-December 2012

[22] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” Journal of Systems and Software, Elsevier, vol. 101, pp. 193–220, March 2015.

[23] E. Lim, N. Taksande, and C. Seaman, “A Balancing Act: What Software Practitioners Have to Say About Technical Debt,” Software, IEEE Computer Society, vol. 29, no. 6, pp. 22–27, Nov. 2012.

[24] A. Martini and J. Bosch, “Towards prioritizing Architecture Technical Debt: information needs of architects and product owners,” 41st Euromicro SEAA Conference, Funchal, Madeira, August 2015.

[25] A. Martini, J. Bosch, and M. Chaudron, “Investigating Architectural Technical Debt Accumulation and Refactoring over Time: a MultipleCase Study,” Information and Software Technology, July 2015.

[26] www. sonarqube .org (official site of sonarqube)

[27] blog.sonarsource.com (official forum of sonarqube)

[28] P. Deitel, H. Deitel: Programming in Java, 8th edition

[29] L. James: Beginning Perl

[30] Modern Perl, 4th edition

[31] Ramakrishnan, Gehrke: Relational Database Managerment Systems, 3rd edition

Appendix

Analysis Phase Functions

```java
private void sonarAnalysisActionPerformed(java.awt.event.ActionEvent evt) {

        mProjectUrl = mGitRepoInputTxt.getText();

        //check if git repository is valid
        if (!mProjectUrl.contains(".git")) {
            String infoMessage = "Please enter a valid .git repository";
            JOptionPane.showMessageDialog(null,
                infoMessage,
                "",
                JOptionPane.INFORMATION_MESSAGE);
            return;
        }

        String[] repoDetails = mProjectUrl.split("/");

        mProjectOwner = repoDetails[3];
        mProjectName = mProjectUrl.substring(mProjectUrl.lastIndexOf("/") + 1,
            mProjectUrl.indexOf(".git")).trim();


        boolean projectExists = mSonarDao.isProjectAlreadyAnalyzed(mProjectName);

        //check if project already exists
        if(projectExists){
           String infoMessage = "This project already exists in the database";

           JOptionPane.showMessageDialog(null,
                infoMessage,
                "" ,
                JOptionPane.INFORMATION_MESSAGE);
           return;
        }

        //analyze project with sonarqube
        String command = "./sonarAnalyzeProj.pl " + mProjectUrl +" > analysisOutput.log"
        SSHConnection sshConnection = new SSHConnection(command);
        sshConnection.executeCommand();

        //analyze project with metrics calculator
        Versions versions = null;
        VersionsFetcher fetcher = new VersionsFetcher(
            mProjectUrl,
            mProjectName,
            mProjectOwner,
            mBuildToolFlag);
        try {
            versions = fetcher.fetchVersion();
        } catch (GitAPIException) {
            Logger.getLogger(AnalysisForm.class.getName()).log(Level.SEVERE, null, ex);
        }

        //add .jar for each version for analysis
        if (versions != null && versions.getVersions() != null && !versions.getVersions().isEmpty()) {
            for (Version version : versions.getVersions()) {
                boolean addMoreJars = true;
                while (addMoreJars) {
                    int result = JOptionPane.showConfirmDialog(null,
                            "Add the jar file for version:" + version.getVersion(),
                            "ADD .JAR",
                            JOptionPane.YES_NO_OPTION);
                    if (result == JOptionPane.YES_OPTION) {
                        JFileChooser chooser = new JFileChooser("/gitTest");
                        chooser.showOpenDialog(null);
                        File file = chooser.getSelectedFile();
                        if (file != null) {
                            String path = file.getAbsolutePath();
                            storeMetricsIntoDatabase(path, version.getVersionCommit().getCommitId(), version.getVersion());
                            addMoreJars = addMoreForVersion();
                        }
                    }else{
                        addMoreJars = false;
                    }
                }
            }
        } else {
            String infoMessage = "No version available for this project";
            JOptionPane.showMessageDialog(null, infoMessage, "", JOptionPane.INFORMATION_MESSAGE);
        }
    }
```

Computation Phase Functions

```
public Results(String computationScope, String selectedProject) {
        initComponents();

        Dimension dim = Toolkit.getDefaultToolkit().getScreenSize();
        this.setLocation(dim.width / 2 -
 this.getSize().width / 2, dim.height / 2 - this.getSize().height / 2);

        // Configuration from analysis form
        mProjectName = selectedProject;
        mComputationScope = computationScope;

        // Initializaton of dao classes
        mMetricsDao = new MetricsDaoImpl();
        mSonarDao = new SonarDaoImpl();

        getProjectHistory();
        getSonarSnapshotIdForVersion();
        getArtifactsForVersion();
        getArtifactsFromMetrics();
        getMetrics();
        preparePerArtifactHash();
        calculateSimilarityForVersion();
        calculateSumInterest();
        initVersionsComboBox();
    }

    private void initVersionsComboBox() {
        mProjectVersions.forEach(projectVersion -> {
            mVersionsComboBox.addItem(projectVersion.getVersionName());
        });
        mSelectedVersionIndex = 0;
        mProjectVer-
sions.get(0).getVersionArtifacts().forEach(versionArtifact -> {
            mArtifactsComboBox.addItem(versionArtifact.getArtifactName());
        });
        mSelectedArtifactIndex = 0;
    }

    private void getProjectHistory() {
        // Retrieve project history from metrics.
        mProjectHistory = mMetricsDao.getProjectHistory(mProjectName);
        mProjectVersions = mProjectHistory.getProjectVersions();
        Sys-
tem.out.println("Finished retrieving project history from metrics");
    }

    private void getSonarSnapshotIdForVersion() {
        // Add snapshot_id from sonar as field on each project version to help
 us retrieve the metrics from sonar.
        Ar-
rayList<SonarVersion> sonarVersions = mSonarDao.getSonarIdsForVersions(mProjec
tName, mProjectHistory.getCommitHashesForVersions());
        //discardVersionsNotAnalyzedFromBothTools(sonarVersions);
        if (mProjectVersions.size() == sonarVersions.size()) {
            for (int i = 0; i < mProjectVersions.size(); i++) {
                for (int j = 0; j < sonarVersions.size(); j++) {
                    if (mProjectVersions.get(i).getCommitHash() != null
                            && sonarVersions.get(j).getCommitHash() != null
                            && mProjectVersions.get(i).getCommitHash().equals(
sonarVersions.get(j).getCommitHash())) {
                        mProjectVer-
sions.get(i).setSonarSnapshotId(sonarVersions.get(j).getSnapshotId());
                    }
```

```java
                }
            }
            System.out.println("Finished retrieving project history from sonar");
        } else {
            System.out.println("Error on sonar analysis!Sonar hasn't analyzed all the required version for us to present the results!");
        }
    }

    private void getArtifactsForVersion() {
        for (int i = 0; i < mProjectVersions.size(); i++) {
            ArrayList<VersionArtifact> versionArtifacts = mSonarDao.getArtifactsForVersion(mComputationScope, mProjectVersions.get(i).getSonarSnapshotId());
            if (!versionArtifacts.isEmpty()) {
                mProjectVersions.get(i).setVersionArtifacts(versionArtifacts);
            } else {
                mProjectVersions.remove(i);
                i -= 1;
            }
        }
        System.out.println("Finished retrieving artifacts from sonar");
    }

    private void getArtifactsFromMetrics() {
        for (int i = 0; i < mProjectVersions.size(); i++) {
            ArrayList<String> metricsArtifacts = mMetricsDao.getClassesAnalyzedForProject(mProjectVersions.get(i).getCommitHash());
            mProjectVersions.get(i).setMetricsArtifacts(metricsArtifacts);
            mProjectVersions.get(i).discardNonAnalyzedClasses(mComputationScope);
        }
        System.out.println("Finished retrieving artifacts from metrics");
    }

    private void getMetrics() {
        for (int j = 0; j < mProjectVersions.size(); j++) {
            List<VersionArtifact> versionArtifacts = mProjectVersions.get(j).getVersionArtifacts();
            for (int i = 0; i < versionArtifacts.size(); i++) {
                VersionArtifact versionArtfact = versionArtifacts.get(i);
                getMetricsFromSonar(versionArtfact);
                String commitHash = mProjectVersions.get(j).getCommitHash();
                getMetricsFromMetricsCalculator(commitHash, versionArtfact);
            }
        }
        System.out.println("Finished retrieving metrics from both tools");
    }

    private void getMetricsFromSonar(VersionArtifact versionArtfact) {
            HashMap<Integer, Integer> metricsForClass = mSonarDao.getSpecificMetrics(versionArtfact.getSnapshotId(), mComputationScope, versionArtfact.getArtifactName());
            versionArtfact.setNcloc(metricsForClass.containsKey(MetricId.NCLOC.id())
                    ? metricsForClass.get(MetricId.NCLOC.id())
                    : 0);
            versionArtfact.setClasses(metricsForClass.containsKey(MetricId.CLASSES.id())
                    ? metricsForClass.get(MetricId.CLASSES.id())
                    : 0);
            versionArtfact.setFunctions(metricsForClass.containsKey(MetricId.FUNCTIONS.id())
                    ? metricsForClass.get(MetricId.FUNCTIONS.id())
                    : 0);
            versionArtfact.setStatements(metricsForClass.containsKey(MetricId.STATEMENTS.id())
                    ? metricsForClass.get(MetricId.STATEMENTS.id())
                    : 0);
            versionArtfact.setComplexity(metricsForClass.containsKey(MetricId.COMPLEXITY.id())
                    ? metricsForClass.get(MetricId.COMPLEXITY.id())
                    : 0);
            versionArtfact.setSqaleIndex(metricsForClass.containsKey(MetricId.SQALE_INDEX.id())
                    ? metricsForClass.get(MetricId.SQALE_INDEX.id())
                    : 0);
        }

        private void getMetricsFromMetricsCalculator(String commitHash, VersionArtifact versionArtfact) {
            String className = versionArtfact.getShortNameForMetrics();
            MetricsClass metricsClass = ("DIR").equals(mComputationScope)
                    ? mMetricsDao.getMetricsForPackage(commitHash, mProjectName, versionArtfact.getArtifactName())
                    : mMetricsDao.getMetricsForClass(commitHash, mProjectName, className);
            versionArtfact.setDac(metricsClass.getDac());
            versionArtfact.setDit(metricsClass.getDit());
            versionArtfact.setLcom(metricsClass.getDac());
            versionArtfact.setMpc(metricsClass.getMpc());
            versionArtfact.setNocc(metricsClass.getNocc());
            versionArtfact.setNom(metricsClass.getNom());
            versionArtfact.setRfc(metricsClass.getRfc());
            versionArtfact.setWmpc(metricsClass.getWmpc());
            versionArtfact.setSize1(metricsClass.getSize1());
            versionArtfact.setSize2(metricsClass.getSize2());
        }

        private void calculateSimilarityForVersion() {
            for (int i = 0; i < mProjectVersions.size(); i++) {
                List<VersionArtifact> versionArtifacts = mProjectVersions.get(i).getVersionArtifacts();
                calculateSimilarityForArtifacts(i, versionArtifacts);
            }
            System.out.println("Finished calculating similarity");
        }

        private void calculateSimilarityForArtifacts(int versionIndex, List<VersionArtifact> versionArtifacts) {
            for (int i = 0; i < versionArtifacts.size(); i++) {
                VersionArtifact investigatedArtifact = versionArtifacts.get(i);
                for (int j = 0; j < versionArtifacts.size(); j++) {
                    if (i != j) {
                        VersionArtifact secondArtifact = versionArtifacts.get(j);
                        double classesSimilarity = calculateCoupling(investigatedArtifact.getClasses(), secondArtifact.getClasses());
                        double complexitySimilarity = calculateCoupling(investigatedArtifact.getComplexity(), secondArtifact.getComplexity());
                        double functionsSimilarity = calculateCoupling(investigatedArtifact.getFunctions(), secondArtifact.getFunctions());
                        double nclocSimilarity = calculateCoupling(investigatedArtifact.getNcloc(), secondArtifact.getNcloc());
                        double sqaleSimilarity = calculateCoupling(investigatedArtifact.getSqaleIndex(), secondArtifact.getSqaleIndex());
                        double statementsSimilarity = calculateCoupling(investigatedArtifact.getStatements(), secondArtifact.getStatements());
                        double similarity = (classesSimilarity
                                + complexitySimilarity
                                + functionsSimilarity
                                + nclocSimilarity
                                + sqaleSimilarity
                                + statementsSimilarity) / 6;
                        investigatedArtifact.addTopArtifact(secondArtifact, similarity);
                    }
                }
                investigatedArtifact.calculateOptimalValues();
                investigatedArtifact.calculateFitnessValue();
                investigatedArtifact.calculateFitnessValueSqale();
                ArrayList<Integer> nclocForPastCommits = getNclocForPastVersions(versionIndex, investigatedArtifact.getArtifactShortName());
                investigatedArtifact.calculateAverageK(nclocForPastCommits);
                investigatedArtifact.calculateBreakingPoint();
            }
        }

        private double calculateCoupling(int metric1, int metric2) {
            double similarity = 0;
            if (metric1 != 0 && metric2 != 0) {
                similarity = 100 - (Math.abs(metric1 - metric2) / (double) Math.max(metric1, metric2) * 100);
            }
            return similarity;
        }

        private void preparePerArtifactHash() {
            mPerArtifactHash = new HashMap<>();
            mProjectVersions.forEach((version) -> {
                version.getVersionArtifacts().forEach((artifact) -> {
                    if (mPerArtifactHash.containsKey(artifact.getArtifactShortName())) {
                        mPerArtifactHash.get(artifact.getArtifactShortName()).add(artifact);
                    } else {
                        ArrayList<VersionArtifact> artifacts = new ArrayList<>();
                        artifacts.add(artifact);
                        mPerArtifactHash.put(artifact.getArtifactShortName(), artifacts);
                    }
                });
            });
        }

        private ArrayList<Integer> getNclocForPastVersions(int versionIndex, String artifactName) {
            ArrayList<Integer> nclocForPastVersions = new ArrayList<>();
            ArrayList<VersionArtifact> artifacts = mPerArtifactHash.get(artifactName);
            for (int i = versionIndex; i < artifacts.size(); i++) {
                nclocForPastVersions.add(artifacts.get(i).getNcloc());
            }
            return nclocForPastVersions;
        }

        private void calculateSumInterest() {
            Iterator<ArrayList<VersionArtifact>> iterator = mPerArtifactHash.values().iterator();
            while (iterator.hasNext()) {
                produceSumInterest(iterator.next());
            }
            System.out.println("Finished calculating sum interest");
        }

        private void produceSumInterest(ArrayList<VersionArtifact> artifacts) {
            for (int i = 0; i < artifacts.size(); i++) {
                VersionArtifact artifact = artifacts.get(i);
                double sumInterestRealK = 0;
                double sumInterestSqaleRealK = 0;
                if (i < artifacts.size() - 1) { // if not the first version
                    for (int j = i + 1; j < artifacts.size(); j++) { // for each version priot to the investigated
                        if (j + 1 < artifacts.size()) {
                            sumInterestRealK += artifacts.get(j).getRealK() * artifacts.get(j + 1).getFitnessValue();
                            sumInterestSqaleRealK += artifacts.get(j).getRealK() * artifacts.get(j + 1).getFitnessValueSqale();
                        }
                    }
                }
                artifact.setSumInterest(sumInterestRealK);
                artifact.setSumInterestSqale(sumInterestSqaleRealK);
                artifact.calculateBreakingPointSumInterest();
            }
        }
```

Presentation Phase Functions

```java
private void mVersionsComboBoxActionPerformed(java.awt.event.ActionEvent evt) {

        int selectedVersion = mVersionsComboBox.getSelectedIndex();
        if (mSelectedVersionIndex == selectedVersion) {
            return;
        }

        mArtifactsComboBox.removeAll();
        mSelectedVersionIndex = selectedVersion;
        mSelectedArtifactIndex = 0;

        mProjectVersions.get(mSelectedVersionIndex).getVersionArtifacts().forEach(versionArtifact -> {
            mArtifactsComboBox.addItem(versionArtifact.getArtifactName());
        });
    }

    private void mArtifactsComboBoxActionPerformed(java.awt.event.ActionEvent evt) {

        int selectedArtifactIndex = mArtifactsComboBox.getSelectedIndex();
        if (mSelectedArtifactIndex == selectedArtifactIndex) {
            return;
        }

        mSelectedArtifactIndex = selectedArtifactIndex;
        mSimilarityTable.removeAll();
        mMetricsTable.removeAll();

        VersionArtifact selectedArtifact = mProjectVersions.get(mSelectedVersionIndex).getVersionArtifacts().get(mSelectedArtifactIndex);

        SimilarityModel similarityModel = new SimilarityModel(selectedArtifact.getTopSimilar(), selectedArtifact);
        mSimilarityTable.setModel(similarityModel);
        MetricsModel metricsModel = new MetricsModel(selectedArtifact.getTopSimilar(), selectedArtifact);
        mMetricsTable.setModel(metricsModel);

        mPrincipalLabel.setText("Principal: "+selectedArtifact.getPrincipal());
        mInterestLabel.setText("Interest: "+selectedArtifact.getInterestAvgK());
        mBreakingPointLabel.setText("Breaking Point: "+selectedArtifact.getBreakingPoint());
    }

    private void reportButtonActionPerformed(java.awt.event.ActionEvent evt) {
        try {
            String fileName = mProjectName + ".csv";
            PrintWriter writer = new PrintWriter(fileName, "UTF-8");
            writer.println("Version;"
                    + "Scope;"
                    + "Name;"
                    + "Principal;"
                    + "Avg K;"
                    + "Real K;"
                    + "Fitness Value (Metrics);"
                    + "Fitness Value (Sqale);"
                    + "Interest in AvgK (Metrics);"
                    + "Interest in RealK (Metrics);"
                    + "Interest in AvgK (Sqale);"
                    + "Interest in RealK (Sqale);"
                    + "Breaking Point (Metrics);"
                    + "Breaking Point (Sqale);"
                    + "Breaking Point minus Sum Interest (Metrics);"
                    + "Breaking Point minus Sum Interest (Sqale);"
                    + "Avg Similarity;");
            for (ProjectVersion version : mProjectVersions) {
                for (VersionArtifact artifact : version.getVersionArtifacts()) {
                    writer.println(version.getVersionName() + ";"
                            + getStringForScope() + ";"
                            + artifact.getArtifactName() + ";"
                            + artifact.getPrincipal() + ";"
                            + artifact.getAvgK() + ";"
                            + artifact.getRealK() + ";"
                            + artifact.getFitnessValue() + ";"
                            + artifact.getFitnessValueSqale() + ";"
                            + artifact.getInterestAvgK() + ";"
                            + artifact.getInterestRealK() + ";"
                            + artifact.getInterestSqaleAvgK() + ";"
                            + artifact.getInterestSqaleRealK() + ";"
                            + artifact.getBreakingPoint() + ";"
                            + artifact.getBreakingPointSqale() + ";"
                            + artifact.getBreakingPointWithSum() + ";"
                            + artifact.getBreakingPointWithSumSqale() + ";"
                            + artifact.getAverageSimilarity() + ";"
                    );
                }
            }
            writer.close();
        } catch (IOException e) {

        }
    }

    private void resultsButtonActionPerformed(java.awt.event.ActionEvent evt) {

        Iterator<String> iterator = mPerArtifactHash.keySet().iterator();
        ArrayList<ResultsArtifact> results = new ArrayList<>();

        while (iterator.hasNext()) {
            String artifact = iterator.next();
            ArrayList<VersionArtifact> artifacts = mPerArtifactHash.get(artifact);

            double principalEvolution = artifacts.size() == 1
                    ? artifacts.get(0).getPrincipal()
                    : artifacts.get(0).getPrincipal() - artifacts.get(artifacts.size() - 1).getPrincipal();
            double interestEvolution = artifacts.size() <= 2
                    ? 0
                    : artifacts.get(0).getInterestAvgK() - artifacts.get(artifacts.size() - 2).getInterestAvgK();
            double bpEvolution = artifacts.size() <= 2
                    ? 0
                    : !Double.isInfinite(artifacts.get(0).getBreakingPoint())
```

```java
                        ? findLastAppliedBreakingPoint(artifacts)
                        : 0;
                ResultsArtifact resultsArtifact = new ResultsArtifact(artifact, principalEvolution, interestEvolution, bpEvolution);
                results.add(resultsArtifact);
            }

            try {
                String fileName = mProjectName + "_overview.csv";
                PrintWriter writer = new PrintWriter(fileName, "UTF-8");
                writer.println("Name;"
                        + "Principal Evolution;"
                        + "Principal Evolution Type;"
                        + "Interest Evolution (Metrics);"
                        + "Interest Evolution Type;"
                        + "Breaking Point Evolution (Metrics);"
                        + "Breaking Point Evolution Type;");
                for (ResultsArtifact artifact : results) {
                    writer.println(artifact.getArtifactName() + ";"
                            + artifact.getPrincipalEvolution() + ";"
                            + getStringForEvolution(artifact.getPrincipalEvolution()) + ";"
                            + artifact.getInterestEvolution() + ";"
                            + getStringForEvolution(artifact.getInterestEvolution()) + ";"
                            + artifact.getBpEvolution() + ";"
                            + getStringForEvolution(artifact.getBpEvolution()) + ";");
                }
                writer.close();
            } catch (IOException e) {

            }
        }

        private void mProjectReportButtonActionPerformed(java.awt.event.ActionEvent evt) {
            try {
                String fileName = mProjectName + "_project_overview.csv";
                PrintWriter writer = new PrintWriter(fileName, "UTF-8");
                writer.println("Version Name;"
                        + "Principal;"
                        + "Interest;"
                        + "Breaking Point;");
                for (ProjectVersion projectVersion : mProjectVersions) {
                    double totalPrincpal = 0;
                    double totalInterest = 0;
                    for (VersionArtifact versionArtifact : projectVersion.getVersionArtifacts()) {
                        totalPrincpal += versionArtifact.getPrincipal();
                        totalInterest += versionArtifact.getInterestAvgK();
                    }
                    double breakingPoint = totalPrincpal / totalInterest;
                    writer.println(projectVersion.getVersionName() + ";"
                            + totalPrincpal + ";"
                            + totalInterest + ";"
                            + breakingPoint + ";");
                }
                writer.close();
            } catch (IOException e) {

            }
        }
```