UNIVERSITY OF MACEDONIA
SCHOOL OF INFORMATION SCIENCES
DEPARTMENT OF APPLIED INFORMATICS

Graph-Based Software Evolution Analysis of Object-Oriented Systems

Theodoros Chaikalis

Ph.D. Dissertation

Supervisor: Assoc. Prof. Alexander Chatzigeorgiou

Advisors: Prof. Maria Satratzemi, *Prof.* Ioannis Stamelos

Thessaloniki, 06/2016

Graph-Based Software Evolution Analysis of Object-Oriented Systems

Theodore Chaikalis

BSc, MSc, Applied Informatics, UoM, 2007, 2009

Ph.D. Dissertation

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Macedonia

Thessaloniki, Greece

2016

Abstract

Contemporary software systems evolve over a large number of versions due to the need for continuous adaptive and corrective maintenance. The duration and cost of this process call for software systems that can be extended or modified in an effortless and rapid way. To this end, the maintainability of such systems can be enhanced by taking advantage of software history and analyzing the evolution of the source code and other artifacts stored in version control systems. In this context, inspired by applications of networks in other domains, this thesis proposes graph-based tools and techniques for different aspects of software lifecycle that facilitate the evolutionary analysis of object-oriented systems.

A fundamental process that determines not only the structure of a software system but also the future maintainability, is the distribution of feature implementation code to different software modules, also known as feature scattering. Towards the better understanding of this phenomenon, a set of techniques for the analysis of the evolution of feature scattering, based on the classes and methods involved in the implementation of high-level, distinct and observable pieces of functionality is introduced. The proposed analyses have been applied on several versions of four open-source projects. Based on the results, the applied techniques appear to be promising since they allow software stakeholders to assess the evolution of feature scattering in various ways, thus gaining insight into the associated implications.

Furthermore, in this thesis an attempt to develop a prediction model capable of forecasting trends in the evolution of networks representing Java systems has been made. The proposed model incorporates findings regarding the growth patterns of software networks, such as the conformance to the preferential attachment model, the duplication model, the effect of node age, and the number of node and edge removals. Moreover, by acknowledging the importance of domain specific characteristics, the model has been enhanced by rules specific to object-oriented design. Evaluation against ten open-source projects showed that the proposed forecasting approach can provide sufficient insight to the future evolution

trends. Software maintenance can benefit from the application of such models by focusing on parts of the network whose properties tend to deteriorate.

Finally, by acknowledging the lack of a tool for effortless software evolution analysis, a platform for the facilitation of software engineering research has been developed. The tool under the name ‘SEAgle’ is a web-based platform that enables users to analyze any git open source repository that is freely available on the internet. SEAgle calculates a multitude of metrics of three different categories and publishes the results in a free online web page and renders them available through well documented REST web services.

Keywords:
Software Evolution, Software Modeling, Graph, Network, Software Metrics, Network Modeling, Forecasting Model, Prediction Model, Feature Scattering, Formal Concept Analysis, Multidimensional Scaling, Preferential Attachment, Duplication Model

Περίληψη

Τα σύγχρονα αντικειμενοστρεφή συστήματα λογισμικού εξελίσσονται συνεχώς, οδηγούμενα από τις συνεχείς ανάγκες, αφενός για προσθήκη νέων λειτουργικών χαρακτηριστικών και αφετέρου για διόρθωση σφαλμάτων. Η διάρκεια και το κόστος αυτής της διαδικασίας καθιστούν επιτακτική την ανάπτυξη ευέλικτων συστημάτων λογισμικού, τα οποία δύνανται να προσαρμόζονται σε αλλαγές εύκολα και γρήγορα. Γι' αυτόν τον σκοπό, η συντηρησιμότητα των συστημάτων μπορεί να βελτιωθεί με την χρήση ιστορικών δεδομένων με σκοπό την ανάλυση της μέχρι σήμερα εξέλιξής τους. Σε αυτό το πλαίσιο, εμπνεόμενη από τις εφαρμογές των γράφων σε διάφορα πεδία, η παρούσα διατριβή προτείνει ένα σύνολο από τεχνικές και μεθοδολογίες βασισμένες σε γράφους. Οι τεχνικές αυτές εφαρμόζονται σε διάφορες φάσεις του κύκλου ζωής λογισμικού, έχοντας πάντα σαν απώτερο στόχο την ανάλυση της ιστορικής εξέλιξης και όχι κάποιας συγκεκριμένης χρονικής αποτύπωσης.

Μια θεμελιώδης διαδικασία στην ανάπτυξη αντικειμενοστρεφούς λογισμικού, η οποία καθορίζει όχι μόνο την δομή των μονάδων λογισμικού αλλά και το επίπεδο της μελλοντικής συντηρησιμότητας, είναι η διαδικασία κατανομής του κώδικα που υλοποιεί τις λειτουργικές απαιτήσεις στις υπό-μονάδες λογισμικού. Έχοντας ως στόχο την καλύτερη κατανόηση αυτής της διαδικασίας, προτείνουμε μια σειρά από εργαλεία και απεικονίσεις για την ανάλυση της εξέλιξης του τρόπου υλοποίησης των λειτουργικών απαιτήσεων. Η ανάλυση περιλαμβάνει την ανίχνευση των κλάσεων και των μεθόδων που παίρνουν μέρος στην υλοποίηση κάθε λειτουργικής απαίτησης. Οι προτεινόμενες μέθοδοι ανάλυσης εφαρμόσθηκαν σε τέσσερα συστήματα λογισμικού ανοικτού κώδικα και σε γενιές που κάλυπταν όλη τη διάρκεια ζωής τους. Σύμφωνα με τα αποτελέσματα, οι προτεινόμενες τεχνικές είναι ελπιδοφόρες, καθώς επιτρέπουν στην ομάδα διαχείρισης λογισμικού να εκτιμήσει με εύκολο τρόπο την εξέλιξη του τρόπου υλοποίησης κάθε λειτουργικής απαίτησης, καθώς και τις μονάδες λογισμικού στις οποίες 'εξαπλώνεται'.

Επιπροσθέτως, στην παρούσα διατριβή προτείνεται ένα μοντέλο πρόβλεψης της εξέλιξης λογισμικού συστημάτων σε γλώσσα Java. Το μοντέλο δημιουργείται αναλύοντας την ιστορία της εξέλιξης του κάθε έργου και ενσωματώνει συγκεκριμένα χαρακτηριστικά όπως αυτό της επιλεκτικής προσκόλλησης νέων κόμβων, το μοντέλο αναπαραγωγής με πανομοιότυπο τρόπο, την ηλικία των κλάσεων και τις διαγραφές ακμών. Επίσης,

αναγνωρίζοντας το γεγονός πως μια προσπάθεια αποτύπωσης του τρόπου εξέλιξης του λογισμικού δεν μπορεί να αγνοεί κανόνες και αρχές που διέπουν την ανάπτυξή του, το προτεινόμενο μοντέλο, για πρώτη φορά ενσωματώνει συγκεκριμένες αρχές σχεδίασης οι οποίες θα πρέπει να ακολουθούνται κατά τη διάρκεια προσομοίωσης της εξέλιξης. Η αξιολόγηση του μοντέλου έγινε σε δέκα συστήματα λογισμικού ανοικτού κώδικα και κατέδειξε την δύναμή του, η οποία του επιτρέπει να προβλέπει με σημαντική ακρίβεια τη δομή συστημάτων λογισμικού στο εγγύς μέλλον. Ο τομέας της συντήρησης λογισμικού μπορεί να επωφεληθεί από τη χρήση ενός μοντέλου σαν το προτεινόμενο, καθώς μπορεί να επικεντρώσει ή και να εντείνει τις προσπάθειες συντήρησης σε τμήματα του λογισμικού στα οποία υπάρχουν ενδείξεις μεταβολής επί το χείρον.

Τέλος, αναγνωρίζοντας την έλλειψη ενός εύκολου στη χρήση εργαλείου για εξελικτική ανάλυση λογισμικού, υλοποιήθηκε μια πλατφόρμα για την απρόσκοπτη ανάλυση της εξέλιξης αντικειμενοστρεφών συστημάτων γραμμένων στην γλώσσα Java. Η πλατφόρμα με όνομα SEAgle είναι προσβάσιμη από οποιαδήποτε συσκευή μέσω του διαδικτύου και επιτρέπει στους χρήστες της ανάλυση οποιουδήποτε δημόσιου αποθετηρίου λογισμικού ανοικτού κώδικα. Το SEAgle υπολογίζει και προσφέρει μέσω δημόσιας διεπαφής προγραμματισμού εφαρμογών μια πληθώρα μετρικών από τρεις κατηγορίες: γράφου, πηγαίου κώδικα και δραστηριότητας αποθετηρίου.

Acknowledgements

Carrying out a doctoral dissertation requires a significant investment of time, effort and psychological capital. The completion of this thesis would be infeasible without the tireless assistance of my wife Iliana. I would like to thank her and express my gratitude for her support during the various difficulties. Her stoicism during the long nights of work was remarkable.

The second most important person, without whom it would not be possible to complete this work is Assoc. Prof. Chatzigeorgiou who is far more than a professor to me. Alexander is a close friend, an inspiring mentor and an ideal partner.

I would also like to thank my family for their encouragement and generous financial support. I always try to do my best to make them proud.

Finally, I want to express my deep gratitude to the Bodossaki Foundation, for the financial support throughout the scholarship, that helped me focus only on my research activities and pursue publications of high quality.

Contents

List of Figures

List of Tables

1 Introduction

You know you're getting old when everything hurts. And what doesn't hurt doesn't work.

Hy Gardner, American reporter

Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable.

David Lorge Parnas, Canadian software engineer

1.1 Evolution in the context of Software Maintenance

Aging is inevitable and no human being can evade from its consequences. Interestingly, analogies from this well-known process, can also be drawn to the field of computer software. However, the fact that software is comprised of abstract mathematical structures and implements a series of calculations in a strict and well-defined manner, raises the question on how this quality degradation happens. The truth is that software programs are complex structures that need constant modification and maintenance in order to keep delivering the necessary services to their clients. However, in many occurrences and for various reasons, software industries fail in performing appropriate maintenance activities and their software systems become obsolete. According to David Parnas (1994), two are the main causes of the so-called "Software Aging" phenomenon: the failure to adapt to changing needs and the changes that are performed on the software per se.

According to the IEEE international standard for Software Engineering and Lifecycle (2006), software maintenance *is the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage.* Maintenance activities can be split into four categories, namely corrective, adaptive, perfective and preventive. Corrective maintenance

is the reactive modification of a software product that is performed after delivery in order to correct discovered problems. Adaptive maintenance refers to post-delivery modifications that are performed for keeping the program usable in a changing technical environment. Perfective maintenance contains modifications and enhancements intended to improve the overall quality of the system, such as performance improvements or new feature additions. Finally, in the category of preventive maintenance fall changes such as module reorganization and refactorings, that take place in order to facilitate upcoming modifications of the system (Swanson, 1976; "IEEE Std 14764," 2006; Godfrey and German, 2008).

According to the previous description of maintenance activities, it can be deduced that software maintenance is a constant process that starts just after product delivery, continues through the whole software lifecycle, and cannot be appointed to a specific time window. During maintenance each system undergoes feature additions, defect removals, structural design changes and performance improvements. In other words, software systems *evolve* through maintenance activities. The evolution of software systems is nothing more than the record of the maintenance activities that have been applied on them. The term "Software Evolution" and the foundations of research on the scientific area that has created, should be accredited to Meir Lehman who introduced the prominent laws of software evolution (Lehman, 1980). In brief, these laws describe the tendency of software systems to become more complex, increase in size, decline in quality and become less satisfying to their users.

Although software evolution is a multi-parametric process that is affected by a plethora of exogenous factors, such as human intervention, user demands, software defects and hardware infrastructure, significant effort has been put on the modeling of this activity. A model that effectively captures the past of a system's evolution can be used to successfully interpret present phenomena as well as to attempt predictions for future software characteristics. However, the construction of such a model requires a significant amount of activities to precede. These activities typically include careful study and observation of underlying phenomena in the domain in which the model will be applied. One of the most important characteristics that warrants investigation, is the way in which requirement specifications are implemented and specifically the evolution of the implementation trends. A second fundamental characteristic is the distribution of module

dependencies and the locality of new modules that implement novel features. After the successful capture of the trends, the construction of a primary version of the model that encapsulates as many of the captured trends as possible, follows. Finally, the model creation process demands the repetitive application and configuration of the primary version until the model effectiveness reaches an acceptable level.

The understanding and modeling of the way that systems evolve, provides a number of advantages that can increase software maintainability and reduce maintenance costs. By applying an evolution prediction model, a practitioner can explore the trends about the size of modules as well as the nature and count of relationships among them. In this way, he gains a glimpse in the future of the software structure and can timely modify and lead the development efforts accordingly in order to prevent undesirable situations. An advanced identification of a design problem can significantly reduce maintenance costs, since as it has been thoroughly documented, a late bug identification can lead to up to 100 times higher correction costs (Boehm, 1981; Stecklein et al., 2004).

One of the most efficient, malleable and powerful abstract modeling tools is unequivocally the graph. In contemporary science, graphs are used to model a vast amount of activities and structures that exist around us, both concrete such as transportation networks, and intangible such as human friendship networks. Furthermore, the modeling of a problem by means of graphs, enables the application of a number of graph-related algorithms and therefore provides a more efficient path to the solution. Famous examples are algorithms for web page ranking such as the Hyperlink-Induced Topic Search (HITS) developed by Jon Kleinberg (1999), and the PageRank™ developed by the founders of Google, Sergey Brin and Larry Page (1999). Software systems, especially those implemented using object-oriented designs, can be naturally represented as graphs, where software modules are modelled as graph nodes and module interdependencies are represented as edges that connect nodes.

1.2 Objectives

This thesis tries to shed light on different perspectives regarding the construction and lifecycle of object-oriented software systems. The first one is the specificities of feature implementations, the evolution of feature scattering and the effect of preventive maintenance on this scattering phenomenon. The second perspective refers to the

investigation of possible patterns that govern the evolution of software systems and the attempt to formally represent these patterns as a model that encapsulates the process that each software project evolves.

1.2.1 Investigation of feature scattering evolution

The need to continuously monitor software quality and to facilitate software maintenance calls for an appropriate interpretation of requirements traceability in the context of software evolution. Under this perspective, this work proposes several means for the analysis and visualization of data concerning the evolution of the scattering in the requirements implementation and the distribution of methods implementing a specific feature in the involved classes. We also propose methods for the modelling of reuse among features at the method level (i.e. how many methods are shared by two features), a matter that is important since an eventual reuse of classes and methods among features provides a reasonable justification for extended feature scattering over source code, which would otherwise be interpreted as a worrying symptom. Finally, the impact of refactoring application on feature scattering is evaluated, in order to investigate whether common refactorings affect the distribution of methods that participate in the implementation of a feature.

The data, the model and the visualizations that can be extracted allow software stakeholders (and particularly maintainers and quality engineers) to shed light on questions such as:

- How fast is the number of classes and methods involved in the implementation of a certain feature increasing with the passage of software versions?
- Is the distribution of methods contributing to the implementation of a feature uniform?
- Is this distribution becoming more unbalanced as software evolves?
- Are classes/methods reused in the implementation of different features?
- How similar are features to each other, based on their common implementation, and how is this similarity changing over time?
- Is refactoring application improving the scattering of features in source code or not?

To illustrate that the extracted data can provide insight into the evolution of the examined systems, the proposed analyses were employed for a number of successive versions of four open-source projects and for several of their features. The examined systems should be regarded as a sample to exemplify the use of the proposed analyses. It should be clarified that emphasis is given in the proposed techniques rather than the actual results and therefore no attempt to generalize the findings is being made.

For the purpose of the proposed analyses, tools and techniques borrowed from various fields have been employed: Formal investigation and visualization of the evolution of feature scattering is performed by using Formal Concept Analysis, a technique that has also been used for the identification of features in source code (Eisenbarth et al., 2003; Poshyvanyk and Marcus, 2007). The Gini coefficient (Gini, 1921), a measure of statistical dispersion typically used for quantifying the inequality of income distribution, is employed to observe the evolution of the distribution of the methods implementing a certain feature over the involved classes. The evolution of method reuse by different features is studied by using a measure of similarity that has been originally applied in paleontology in order to illustrate part-whole relations. Finally, the evolution of feature similarity based on their common methods is visualized by exploiting multi-dimensional scaling, a widely used tool for data visualization.

1.2.2 Introduction of a software evolution model

Software systems and object-oriented designs in particular, can be naturally represented as graphs. By drawing ideas from social network analysis, the goal is to establish a set of techniques for studying software evolution where systems are represented as networks of interconnected nodes (classes). In particular, the goal is to:

a) study evolutionary trends during software growth by analyzing network properties over successive past software versions,

b) derive models that are capable of forecasting future software evolution in terms of network metrics, based on the observed phenomena and

c) introduce domain knowledge in the construction of the corresponding models in order to improve their accuracy.

The latter is of major importance when modeling a physical or technological network since the interpretation of raw data without considering domain knowledge might

lead to confusing results and unreliable conclusions (Willinger et al., 2009). Whenever possible, an attempt to associate the observed trends at the network level to qualitative properties of the underlying software system has been made.

A model that enables the prediction of the evolution of certain architectural parameters can be valuable to software maintainers. In particular, the ability to forecast the growth of classes and packages as well as the coupling among them can assist the development teams to focus on parts of the design that warrant increased attention. As it has been extensively pointed out by previous empirical studies, heavily loaded modules and excessive coupling are associated with increased fault-proneness and increased maintenance effort (Bhattacharya et al., 2012; Jenkins and Kirk, 2007; Wilkie and Kitchenham, 2000; Yu et al., 2002; Zimmermann and Nagappan, 2008). Moreover, it has been made clear by previous research (Zimmermann and Nagappan, 2008) that network measures can predict critical and error-prone software modules far more accurately than classic complexity metrics.

This thesis proposes a network-based prediction model for software evolution, combining information from past data and domain-related rules. Moreover, several processes of software growth are taken into account, including the creation of relations among existing and new classes and the removal of edges, rather than simply relying on a model that captures how new classes are added to a system. Finally, preliminary analysis revealed that representing software systems as “flat" networks ignoring the existence of structural communities, significantly constrains accuracy, and thus we incorporate the notion of packages in our model. Architectural modularization when representing software systems as networks has also been considered in other research efforts (Turnu et al., 2013; Zanetti and Schweitzer, 2012). Obviously, a prediction model for software evolution is an ambitious goal and its accuracy is constrained by the numerous exogenous factors affecting the software development process. Nevertheless, the fact that the analysis is performed at the relatively abstract network level, enables us to obtain an insight into future trends. Moreover, as it will be shown, the obtained accuracy can be gradually improved by enhancing the model with additional parameters. Evolutionary trends are discussed for ten open-source projects against which the accuracy of the proposed model is tested. The proposed study is backed up by a tool that has been developed for software evolution

analysis in terms of networks. Both the results and the tool are freely available from a public website (2014).

1.2.3 Platform for software evolution analysis

Software evolution analysis and software quality monitoring in general can be significantly facilitated by software tools that help researchers and practitioners in the gathering of necessary metrics and monitoring of various software characteristics. Driven by the shortage of an easy-to-use, service-oriented application, in the context of this dissertation we created a platform for the facilitation of research on software engineering. The platform works in a single-click approach and by cloning remote open-source git repositories, recreates the source code for the total number of defined versions and also constructs the graph that corresponds to each software version. The platform then calculates metrics of three different domains, namely source code, graph-based and repository-based metrics and stores the values in a database that is publicly accessible through an open REST API.

1.3 Contribution

The contribution of this thesis spans to multiple fields. First of all, to the best of our knowledge, it is the first time that feature scattering is investigated through an evolutionary prism. The similarity between features has been captured by means of common methods, while a significant enhancement is proposed for the quantification of similarity. Instead of using a widely adopted similarity measure which does not adequately capture the observed phenomena, the proposed methodology suggests a similarity metric originating from paleontology that successfully demonstrates the notion of subset similarity which is crucial in our context. Furthermore, the monitoring of the impact that refactorings induce on the distribution of feature implementation is considered another novel feature of this dissertation.

Also, concept lattices have been introduced from a new perspective, for the formal, graph-based representation of the classes and methods that contribute to the implementation of a given feature. By visually comparing concept lattices of two different versions, a maintainer can quickly gain insight into the changes on the distribution of code that implements a specific feature.

In the context of software modeling, it should be noted that the created graph model innovatively considers not only the introduction of new nodes and edges, but also encapsulates the patterns of edge deletions and acknowledges the presence of architectural modularization as expressed by software packages.

The main contribution of the proposed evolution model lies in the ability to estimate the network structure representing a software system instead of forecasting an individual software characteristic. In this way, since the entire software structure can be anticipated, the benefit lies in the ability to reason about future values of several software properties emerging from the network topology.

Finally, this thesis also contributes to the field of software engineering tools by introducing 'SEAgle', a web-based platform that enables the analysis of any git open source repository and is freely available on the internet. SEAgle calculates a multitude of metrics of three different categories and publishes the results in a free online web page. Categories include graph-based metrics such as number of nodes, clustering coefficient and graph density, repository-based metrics such as number of commits per version and source code metrics such as complexity, coupling and cohesion. Also, by acknowledging the need of empirical studies to explore relations among variables, the tool offers direct correlation analysis between any two monitored variables.

1.4 Outline

Chapter 2 presents a thorough literature review about the domains of the employed techniques. The review is divided in 3 chapters. The first section deals with related work on techniques dealing with identification of feature implementations, quantification of feature scattering and visualization of feature dependencies. The second section introduces the domain modeling and discusses approaches that deal with software modeling and forecasting of specific software characteristics. The last section presents the related work in the field of tools for software engineering research.

Chapter 3 is dedicated to the topic of feature scattering by initially introducing the proposed techniques and afterwards by presenting the results of the empirical evaluation on four open-source systems. The introduction of the proposed tools starts with the most important, but coarse-grained metric for the quantification of feature scattering, which is the number of classes involved in the implementation of a feature. A formal, powerful and

concise method for the representation and visualization of feature scattering, is the Formal Concept Analysis (FCA). FCA is applied for the modeling and analysis of the relations between features and classes that implements them. The next subchapter presents a more fine-grained analysis, which is the investigation of the number of methods that contribute to the implementation of each examined feature as well as the distribution of those methods over the involved classes. Subsequently, the work proposes metrics for the quantification of the dispersion of methods over the involved classes that implement a specific feature. The first metric, Gini coefficient, is usually employed in the field of economics and quantifies the inequality of a distribution. In the present context, Gini quantifies the degree of inequality in the distribution of methods over the involved classes. The second metric employed is the Degree of Scattering which quantifies the extend by which the implementation of a feature is scattered among many classes. Finally, the presentation of tools for the monitoring of the evolution of feature scattering is completed by the visualization of the similarity among features by means of common methods. The selected metric for the quantification of similarity stems from paleontology and effectively captures the underlying phenomena. For the visual representation of similarities among features, an inverse notion has been adopted, the notion of distance. Consequently, distances among features are depicted in the Euclidean space by using a technique known as Multidimensional Scaling.

Chapter 4 deals with the model of software evolution, the case study design, the description of the phenomena that inspired the forecasting model and the evaluation of the model. The description starts with an introduction to the problem as well as the disadvantages of existing modeling and forecasting methods, by using properties of the ten selected open-source systems as indicative examples.

The analysis of the proposed model begins with the confirmation of the existence of Preferential Attachment (PA) phenomena in the evolution of software. Preferential attachment model suggests that the underlying distribution of edges over the graph nodes follows a power law. However, further investigation reveals the inability of PA model to sufficiently capture the mechanics of software evolution. After acknowledging the fact that software enhancement involves addition of functionality in the form of repetitive steps that modify specific modules, a duplication model has been adopted along with the PA model.

The enrichment of the model continues with the introduction of node age. Indeed, our experiments reveal a decay over time in the activity of edge creations (association creations by classes), consequently the effect of node age has been incorporated in the proposed model. The same observation also holds for the edge removal process, the neglect of which would deteriorate the effectiveness of the proposed model.
Specificities of the domain on which every model will be applied, constitute a critical parameter which should not be neglected according to our point of view. To this end, the proposed model innovatively incorporates rules that standardize the process of object-oriented software development.

The chapter closes with the presentation of the evaluation of the proposed model on ten open-source software systems. The evaluation performed on both the network and the software aspects of the examined systems. Graph-based evaluation implemented by comparing graph properties that provide a representative picture of a network topology. These properties are the distribution of the distance between all node pairs (also known as hop plot), the in-degree distribution, the graph diameter, the number of nodes and edges and graph density. For the evaluation from a software perspective, we compare the predictions of in-degree increase for selected classes against the actual evolution. Moreover, we compare the predicted afferent and efferent coupling at the package level to the actual values of the last examined version.

Chapter 5 presents SEAgle, the platform that has been created for the purposes of this thesis. SEAgle is an online platform that allows by a single-click, Google-like process, the cloning of any open-source git repository, the evolutionary analysis of three aspects (Graph, Source Code, Repository) of the software, and the calculation of a number of metrics. Calculated metrics regard all three aspects and are available through a website and also through a public REST API.

Finally, threats and limitations to validity are discussed in Chapter 6, while conclusions and axes for future research are presented in Chapter 7.

2 Literature Review

The whole of science is nothing more than a refinement of every day thinking

Albert Einstein

Empirical research in Software Engineering usually employs the application of structural, syntactical, semantic or other kind of analysis on specific snapshots of software systems. However, despite the fact that the examination of a static snapshot of a system generates a significant amount of useful information, the overall history of the evolution of the system is still neglected in practice. On the other hand, evolutionary analysis can spotlight historical trends and reveal patterns of module 'behaviors' that otherwise would remain invisible. To this end, the totality of the analyses and tools presented in this thesis are applied in historical timespans of the software systems and not on specific static snapshots.

As described in the introduction, this thesis investigates the evolution of feature scattering in an attempt to better understand the way that functional requirements are implemented and furthermore to monitor the effect of preventive maintenance on the scattering trends. Moreover, a model of software evolution with the outmost target of forecasting the future growth patterns is presented along with a platform that implements the proposed model and calculates a series of software related metrics.

2.1 Feature Scattering

One of the major difficulties of software maintenance is the linking of certain functional requirements with the corresponding software modules that implement them, a process known as Requirements Traceability. This is a crucial part of program understanding and a non-trivial task since the required information is in most cases inefficiently documented (Antoniol et al., 2002; Biggerstaff et al., 1994; Eisenbarth et al.,

2003; Trifu, 2010). According to Gotel and Finkelstein (1994), Requirements Traceability is the ability to follow a requirement from its specification through its deployment in code, in both a forward and backward direction. This activity is also defined as Concern (Eaddy et al., 2007; Trifu, 2010), Concept (Biggerstaff et al., 1994), or Feature Location (Eisenbarth et al., 2003) since the goal is to identify the source code elements implementing a certain functional requirement. In the following, we adopt the term *feature* as defined by Eisenbarth et al. (2003) which refers to a distinct, observable, unit of behavior of a system that can be exercised by the end user.

Several studies have indicated that extensive feature scattering (i.e. when the implementation of a feature is scattered throughout a large number of software modules) and feature coupling (i.e. increased inter-dependence between features), are factors that increase error-proneness and instability (Conejero et al., 2009; Eaddy et al., 2007; Eisenbarth et al., 2003; Filho et al., 2006; Garcia et al., 2005; Gibbs et al., 2005; Greenwood et al., 2007; Koschke and Quante, 2005; Revelle et al., 2011; Robillard and Murphy, 2007; Wilde and Scully, 1995). As an illustrative example, Robillard and Murphy (2007) stress that in order to modify the "save" feature of JHotDraw, the developer has to follow the implementation of this feature throughout at least 35 classes, which are at the same time involved in other features as well, imposing a significant challenge. The problem of feature scattering might deteriorate as software evolves not only due to the expected enhancement of functionality over time, but also due to poor design decisions. In extreme cases, it can lead to systems where a single feature involves hundreds of classes and over a thousand of methods.

The primary challenge in the field of feature to source code mapping is the correct identification of software components implementing a certain feature. Feature identification approaches can be categorized as static, dynamic and hybrid, depending on the nature of the processed information.

Static techniques are mainly based on various Information Retrieval (IR) methods that involve textual matching of terms contained in the project’s requirement documentation that describe a feature, to source code identifiers on the premise that they have meaningful names (Antoniol et al., 2002; Conejero et al., 2009). IR models that are usually employed are Vector Space Model (VSM), Latent Semantic Indexing (LSI), and Probabilistic Network (PN) (Zou et al., 2009). The first steps on automated static Feature

Location were made by Biggerstaff et al. (1994) who have built a tool that locates identifiers in source code and clusters them in order to facilitate Feature Location. Antoniol et al. (2002) proposed a method that employs both Probabilistic Network and Vector Space Model, in order to analyze the mnemonics that serve as identifiers in source code and use them to associate high-level concepts with program concepts. Marcus et al. (2003; 2004) employed Latent Semantic Indexing in order to locate concepts in source code, while, for the same purpose, Shepherd et al. (2007) have made use of Natural Language Processing, a method that originates from Artificial Intelligence. In some approaches, IR methods are assisted by different techniques, as in the work of Poshyvanyk and Marcus (2007), who used Formal Concept Analysis in order to refine the mapping tables that resulted from Latent Semantic Indexing, or Zhao et al. (2003, 2004), who presented a non-interactive method that also employs a structural analysis process named Branch-Reserving Call Graph, which is a call graph with the addition of branch information.

Dynamic approaches entail the execution of a number of test cases that exercise the desired feature in order to enable the capturing of the execution trace and to determine the software modules that are involved in the feature's implementation. Wilde and Scully (1995) presented "Software Reconnaissance", a method which discovers software modules that implement a particular feature. Dynamic execution tracing (slicing) has been also adopted by Wong et al. (1999) who identify source code elements that implement a specific feature or group of features.

The combination of static and dynamic methods has been recognized as an approach that considerably improves the effectiveness of feature identification. In the hybrid approach of Eisenbarth et al. (2003) and Koschke et al. (2005), features are invoked based on execution scenarios, in order to collect dynamic information. Aided by Formal Concept Analysis, the proposed methodologies create concept lattices whose interpretation combined by a static dependency graph, lead to a mapping between features and computational units. Poshyvanyk et al. (2007) and Liu et al. (2007) propose methods that locate features by exploiting the advantages of two distinct methods, namely Latent Semantic Indexing, and Probabilistic Ranking of entities that came of scenario executions.

A study on the evolution of features and their implementation has been performed by Greevy et al. (2006), who categorize software entities according to the level of

participation in features. Furthermore, they investigate the changes in categorization during the evolution of software.

A set of useful metrics for the analysis of how features are implemented in source code has been proposed by Wong et al. (2000), who quantified the closeness between a feature and a software component involved in its implementation. Eaddy et al. (2007, 2008) evolved and extended Wong's metrics by investigating the consequences of scattered and tangled concern implementation (crosscutting concerns) in the quality of programs, in terms of defects. They examined the correlation between the number of bugs and metrics that quantify the scattering of concerns in code (e.g. Degree of Scattering, DoS) at class and method level. Their results indicate a relatively strong correlation between DoS and number of defects. Finally, Conejero et al. (2009) have also employed crosscutting metrics in order to predict possible software instability in early development artifacts such as requirement descriptions.

The majority of previous studies on requirements traceability focus on creating precise and accurate feature location techniques (Zou et al., 2009) aiming at the analysis of individual software versions. This work emphasizes the need to investigate the evolution of feature scattering over software versions and also performs a more fine-grained analysis which considers not only the evolution of classes and methods involved in the implementation of a feature but also the common classes among features (Greevy et al., 2006; Wong et al., 2000).

2.2 Software Evolution Modelling

Motivated by the phenomenal growth of social networks during the last decade, significant research has been performed on the study of the evolutionary trends exhibited by social networks such as Flickr and LinkedIn (Leskovec et al., 2008), technological networks such as those formed by web pages (Faloutsos et al., 1999) and routers (Li et al., 2005) as well as biological and other networks (Easley and Kleinberg, 2010). The ultimate goal in these studies is to interpret the macroscopic phenomena at the network level, relate them with the microscopic behavior of individual nodes and eventually forecast the future evolution of the corresponding networks (Leskovec et al., 2008).

Software systems have also been treated as networks of various forms within the software engineering community. The most common form assumes that software modules

are represented as nodes while relations among them correspond to edges. Other software artifacts but also people involved in the software development process have been considered as nodes leading to different kinds of networks. Modeling software systems as networks enabled a graph-based treatment and analysis with the goal of investigating several properties, such as scale-freeness (Louridas et al., 2008; Mens, 2008; Potanin et al., 2005; Taube-Schock et al., 2011; Wheeldon and Counsell, 2003), and the presence of small-world phenomena (Kleinberg, 2000; Valverde and Sole, 2003a). However, with the exception of few recent research efforts (Bhattacharya et al., 2012; Li et al., 2013; Zheng et al., 2008), most of the studies that employ graphs for the representation of software, focus on static snapshots of the examined systems in the sense that individual software versions have been analyzed without paying attention to the evolution of the corresponding networks.

A large number of approaches have been developed aiming at the prediction of individual software characteristics or properties. In all cases, a forecasting model is built based on the analysis of an information set (historical data, models and assumptions available at a given time) and thus every forecast is conditional on this information. All kinds of forecasting methods can be classified under the two broad categories of explanatory (or causal) and time series models. Explanatory models (including the widely used regression analysis) assume that the variable to be forecasted exhibits a causal relationship with other independent variables. Time-series forecasting treats the examined system as a black box and attempts to derive the generating process of a set of time-dependent data (Granger and Newbold, 1977). Quite often these approaches are enhanced by artificial intelligence (such as neural networks (Vinay Kumar et al., 2008) and genetic algorithms (Chang et al., 2008)) or probabilistic techniques (such as Bayesian models (van Koten and Gray, 2006)). Indicative approaches where a forecasting method has been employed to predict future values for a particular software characteristic are shown in Table 1. Although this list is not exhaustive, it covers a large portion of software properties, which have been the target of forecasting.

Table 1: Forecasting methods employed to predict various software characteristics

Characteristic	Method used
Bug Presence / Error-prone components	Logistic Regression (Basili et al., 1996) Correlation Analysis (Nagappan et al., 2006; Zimmermann et al., 2007) Principal Component Analysis (Arisholm and Briand, 2006), (Nagappan et al., 2006; Zimmermann et al., 2007) Bug Caching, Historical Analysis (Kim et al., 2007) Multivariate Adaptive Regression Splines (Jiang et al., 2013) Cost-benefit analysis (Arisholm and Briand, 2006)
Bug Fixing Time	Monte Carlo Simulation (Zhang et al., 2013)
Bug Complexity	Clustering with k-means and k-medoids (Nagwani and Bhansali, 2010)
Bug Density	Correlation Analysis, Multiple Regression (Nagappan and Ball, 2005)
Cost - Effort	Expert Judgment (Jørgensen, 2007) Analogy Based estimation (Li et al., 2007, 2009) Artificial Neural Networks (Jun and Lee, 2001; Li et al., 2009; Vinay Kumar et al., 2008) Bayesian Network Models (Pendharkar et al., 2005), (USC-CSE, 1997) Multiple Regression (USC-CSE, 1997)
Project Scheduling	Genetic Algorithms (Chang et al., 2008)
Code clones	ARIMA Time Series (Antoniol et al., 2001) Dynamic Programming Matching (Kontogiannis et al., 1996) Statistical Pattern Matching (Kontogiannis et al., 1996)
Design Model Evolution	ARMA Time Series (Yazdi et al., 2014)
Change Proneness	Sequential pattern mining (Kagdi, 2007) Association rule mining (Ying et al., 2004)
Project Survivability	Information Theory (Raja and Tretter, 2012)
Maintainability	Hierarchical Multidimensional Assessment (Coleman et al., 1994) Polynomial Maintainability Assessment (Coleman et al., 1994) Fuzzy Prototypical Knowledge Discovery (Genero et al., 2001) Bayesian Network - Naïve-Bayes Classifier (van Koten and Gray, 2006) Regression Tree (van Koten and Gray, 2006; Zhou and Leung, 2007) Multivariate Linear Regression (van Koten and Gray, 2006; Zhou and Leung, 2007) Artificial Neural Network (Zhou and Leung, 2007) Multivariate Adaptive Regression Splines (Zhou and Leung, 2007) Non-Homogeneous Poisson Process (Shibata et al., 2007)

The analysis of the evolution of software systems, as well as the attempts to facilitate the understanding of the way that systems evolve, have drawn considerable interest in the last years (Mens, 2008). The significance of studying the evolutionary trends during software growth, has been highlighted quite early in the history of software engineering as vividly captured by the laws of software evolution defined by Lehman (1980). A discussion of software evolution from several perspectives and a comparison to

other kinds of evolution in various domains has been presented in the study by Godfrey and German (2008). One of the most interesting conclusions is that software evolution can offer insight into questions of both science and engineering. Different types of evolution analyses have been surveyed by Gîrba and Ducasse (2006), where a set of requirements for building evolution meta-models has been proposed.

Since the proposed model essentially encompasses the use of networks as a representation medium and as a tool to facilitate the study of software evolution, we focus here at previous research efforts related to tools for software evolution analysis as well as works that employed graphs for this purpose.

Richard Wettel (2009) and Wettel et al. (2011) introduced a software visualization tool for the facilitation of monitoring software evolution that models source code packages as city districts and classes as buildings. The height of the buildings depicts the number of methods while the area depicts the numbers of attributes. In this way by examining the historical snapshots of the "software city", one can easily gain a visual, coarse-grained overview of the evolution of the software system.

Bevan et al. (2005) proposed a tool called "Kenyon" to support software evolution research. Kenyon eases the extraction of data from source code repositories providing support for multiple Configuration Management Systems. The provided infrastructure enables the access and processing of collected data thus supporting several types of analyses such as the investigation of code feature evolution and history analysis in terms of graph-based software representations.

D'Ambros and Lanza (2008) developed the "Churrasco" framework which enables the analysis of a source code repository located in a remote version control system (and of the accompanying bug tracking system data if available) and the modeling of software evolution. The framework provides a visualization module offering interactive visualizations concerning the evolution of dependencies among modules and size metrics, and an annotation module that supports collaborative analysis where users can share annotations on model entities.

Although not directly comparable to the prediction of future evolution that is proposed in this work, existing research efforts in the field of change proneness prediction can also be considered as relevant, since forecasting is the goal. Gîrba et al. (2004) attempt to identify software components that changed in the recent versions of software systems

under the assumption that such components are more likely to change again in the near future. They also propose rules for the characterization of the evolution of class hierarchies in order to facilitate the identification of changes and change propagation among class hierarchies.

Vasa et al. (2007) employed type dependency graphs to analyze consecutive releases of several object-oriented systems with the goal of understanding how changes are distributed over the classes and interfaces of systems and how these changes are affected by size, popularity and complexity of classes. They calculated 25 different metrics for each class in all consecutive project versions and compared the values between two versions to decide if the class has been changed or not. Their results indicate that as software evolves, and after a first period of intense functionality addition, the size and complexity measures of classes stabilize and do not fluctuate intensively. Another interesting outcome is that classes with high fan-in values (popular ones) are more likely to change from one version to the next, in other words the increased size, popularity and complexity of a class affect its change-proneness.

The proposed use of networks/graphs for software evolution analysis is based on the fact that several artifacts of the software development process have long been modeled as graphs. Such graphs have been employed to represent, for example, data or control dependencies between statements, relations between software components, links between project activities and priorities between requirements (Ghezzi et al., 2003). Many researchers have also studied the properties of the corresponding networks such as the presence of power laws in the distributions of different software components (Concas et al., 2007; Fortuna et al., 2011; Louridas et al., 2008; Myers, 2003; Potanin et al., 2005; Sun et al., 2009; Turnu et al., 2011; Wheeldon and Counsell, 2003), the existence of small-world phenomena (Kleinberg, 2000; Valverde and Sole, 2003a), and the definition and identification of network motifs (Ma et al., 2008; Valverde and Sole, 2003a). Taube-Schock et al. (2011) studied connectivity in 97 open source systems and concluded that scale-free structure in the source code translates directly to high coupling and therefore claim that high coupling cannot be eliminated from software design.

Furthermore, several approaches have tried to explore the potential of network properties, such as node degree and centrality, to act as software quality indicators or bug predictors. Pinzger et al. (2008) represented the interconnection of developers with

software modules in a network structure, and found a correlation between the centrality of software modules in the network and the error proneness of these modules. A similar work has been carried out by Meneely et al. (2008). Zimmermann and Nagappan (2008) modelled the interactions among binaries of the Windows Server 2003 as a binary dependency network and investigated the correlation between several network measures (especially centralities) and the number of defects. Their results highlight the fact that network measures are much better predictors of critical binaries than the traditional complexity metrics. They also claim that binary dependency network metrics can predict the number of defects. Turnu et al. (2013) propose the fractal dimension network metric as a software complexity indicator. They also provide evidence of correlation between the fractal dimension metric and the number of defects. Zanetti and Schweitzer (2012) proposed a metric that quantifies the modularity of a system by using its representation as a software network.

Paymal et al. (2011) have investigated the changes to graph vertex properties such as degree, betweenness centrality, clustering coefficient and measured the extent of disruption after perfective maintenance in the evolution of JHotDraw application. Wang et al. (2009) studied the evolution of the call graphs corresponding to 223 consecutive versions of the Linux kernel, employing several graph properties as metrics, observing that the network growth relies heavily on preferential attachment. A network representation of the Linux operating system has also been studied by Fortuna et al. (2011), who analyzed dependencies among packages for the first ten releases of Debian. According to their findings, the system exhibits high modularity which increases with the passage of versions, although not with a constant rate. The authors claim that the increased modularity, albeit not totally preventing incompatibilities among modules, helps to avoid conflicts between software subcomponents and eases the installation of autonomous modules.

A more systematic study of the evolution of networks representing software systems as well as the introduction of models that govern their evolution has also been attempted by a limited number of researchers. Li et al. (2013) proposed a model according to which, software networks grow not only by individual node additions, but also with multiple node attachments in the form of complete modules. The model starts with the insertion of single nodes in the existing network, but after some initial steps, the number of nodes added in each step is increasing. At first, the nodes to be inserted form a new sub-

network S' according to the Preferential Attachment model. Then the entire sub-network S' is attached to the existing network according to a set of defined rules that govern the edge creation and the target node selection.

Bhattacharya et al. (2012) capture software structure by means of networks, at two levels, namely the source code and the developer collaboration level. A set of graph metrics such as number of nodes and edges, average degree, clustering coefficient and edit distance, is employed to study the evolution over the entire lifespan of 11 open-source projects. The employed metrics have been shown to be valid predictors of bug severity, high-maintenance software modules and failure-prone releases.

The proposed approach differs from the aforementioned efforts because it aims at predicting the future evolution of the entire network topology representing an object-oriented system. Moreover, the proposed prediction model considers a multitude of parameters such as generic growth models (e.g. preferential attachment), past evolution data as well as domain knowledge for object-oriented design. In addition, system structure in the form of packages is also considered in the proposed model.

One representative case where the consideration of additional parameters appears to improve the forecasting power is the work by Zheng et al. (2008) who represented the Gentoo Linux open-source operating system as a complex network with software packages as the nodes and inter-package dependencies as the edges in an attempt to study its evolution by means of networks. Their empirical results indicated that existing network models are not capable of predicting the actual evolution of the software systems and therefore they propose an evolution model that considers node age in parallel with node degree, called Degree and Age Dependent Adjustable Evolution (DAAE) model. The introduction of age seems to improve the forecasting power of their model, which however has not been evaluated thoroughly.

The conclusions of this approach are in line with our observations that the Preferential Attachment model by itself is not capable of forecasting the evolution of software systems due to their inherent special characteristics. However, apart from taking into account the node age, we also consider specific domain rules that govern the development of software systems. Furthermore we model the creation of edges between existing nodes, between existing and new nodes as well as the removal of existing edges, issues that have not been taken into account in previous models but which fundamentally

affect the topology of the resulting software networks. Finally, the evaluation is performed on ten open-source systems (models proposed by Zheng et al. (2008) and Li et al. (2013) are evaluated on a single system), which enables the generalization, up to a certain degree, of the observed evolutionary trends in software evolution.

2.3 Software Network Analysis Tools

Various noteworthy tools have been developed by research teams for large (social or other) network analysis, featuring numerical calculations and visual representation of the corresponding data. Wikipedia lists over 70 products under the term social network analysis tools and libraries. A number of them like UCINet (2014), Gephi (2014), Pajek (2014) and GUESS (2014) are standalone programs for network visualization and metrics computation while others are software libraries offering customizable capabilities like SNAP (2014) for C++ and NetworkX (2014) for Python. Notable systems like NetMiner (2014), igraph (2014) and Cytoscape (2014) have focused in the efficient graphical representation of the underlying networks while others like SNAP focus in the analysis of ultra large-scale connected components employing techniques based on sampling.

However, none of these tools focuses on software and all of them assume that a dataset containing an edge list, a node list or a full matrix representation is already available. Consequently, in order to analyze software one has to build his own parser to extract the software representation and port the results to the corresponding tool.

On the other hand, several tools have been developed to assist software maintenance by visualizing various aspects of software evolution; however, to the best of our knowledge these tools do not treat the software system as a network. These tools essentially offer a timeline for the visualization of software artifacts, such as project hierarchies (González-Torres et al., 2011) or metric values (Pinzger et al., 2005).

The increasing interest on software repository mining led to the creation of several tools, frameworks and techniques that facilitate the overall process. Most of the existing approaches has been recorded by Chaturvedi et al. (2013) who reviewed all papers published in conferences related to repository mining since 2007. In more than half of the papers the proposed approach is backed up by a tool developed for this purpose.

One of the most prominent tools is SonarQube (2014), which is an online platform that evaluates software quality and through a reporting mechanism it provides an overview

of the project state as well as the estimated technical debt. This type of continuous analysis can certainly be applied on projects retrieved from public repositories. However, SonarQube has not been designed with the software engineering researcher in mind, as each project has to be downloaded individually. More targeted to software engineering research is Ohloh (2014), a public directory of open source projects that provides basic information about the size and developer contribution among others. A notable tool for Automated Software Engineering called Kenyon has been developed by Bevan et al. (2005). Its main feature is the ability to facilitate the creation of new evolution analysis tools as well as the data sharing among them. Deep IntelliSense (Holmes and Begel, 2008) is a Visual Studio plugin that can provide information related to dependencies among software modules in order to help developers better understand the way that each artifact has evolved. Linstead et al. (2009) proposed "Sourcerer", an infrastructure that automatically parses and analyzes online software repositories in order to provide information about the program functions and source code similarities as well as developer activities and similarities among developer programming styles.

However, despite the abundance of tools and platforms, the systematic presentation of the evolution of size properties, software metrics and repository activity together in a single dashboard and without the need for human intervention, is still not available. Therefore, software engineers are still compelled to collect data from many different sources, transform the data in a common format and finally combine information from each field to perform meaningful queries. Furthermore, although the field of Mining Software Repositories is rather mature, there is still a shortage of general frameworks that will integrate and ease the mining of the repositories, the analysis of source code, commits and bugs and the reporting capabilities. The main obstacle that prevents the creation of such tools (also known as automated software engineering tools) is the difficulty in scaling. That is the weakness to handle the enormous amount of data that modern repositories contain (Shang et al., 2010), however in the last years some approaches did emerge. From our personal experience a further weakness is the difficulty in setting up, configuring and using these tools.

3 Evolution of Feature Scattering

Any sufficiently advanced bug is indistinguishable from a feature.

Bruce Brown

3.1 Concept Description

In order to investigate the evolution of feature scattering over several versions of a software system, the classes and methods implementing each feature should be identified for each of the examined versions. This constitutes one of the major challenges in the area of Requirements Traceability and particularly of Feature Location (Antoniol et al., 2002; Biggerstaff et al., 1994; Eisenbarth et al., 2003; Trifu, 2010). In our case, the extraction of classes and methods involved in the implementation of selected features has been performed by employing dynamic analysis with the use of a Java Profiler ("Java Profiler - JProfiler," 2012). Dynamic analysis as an approach for feature location has also been adopted in other efforts (Eisenbarth et al., 2003; Koschke and Quante, 2005; Poshyvanyk et al., 2007).

In order to capture the creation of class instances and method calls related to a specific feature, we set up a scenario that exercises the feature and executes it in analogy to the approach employed by Wilde and Scully (1995), while the program is running in profiling mode. For example, in the case of the JMol chemical structure viewer, the scenario for profiling the rendering of molecules which are stored in files of type *mol*, contains the following steps: 1. Click File Menu, 2. Select Open File, 3. Navigate to Aspirina.mol file, 4. Click OK. The analysis is restricted only to the source code of the system classes of the projects. In other words, methods and classes belonging to external packages and libraries are excluded. No further filtering on the obtained data is performed.

The entire process that have been followed in order to analyze the scattering of features is illustrated in Figure 1. In the first step, selected features are exercised on the

application of interest while being monitored by the profiler. Next, the methods invoked in the executed feature are analyzed to obtain the classes in which they reside and to generate the reports shown on the right hand side of Figure 1. Regarding the reports, their interpretation can be performed in the following sequence: An overview of feature scattering evolution is provided by the graphs showing the number of involved classes and methods in each version. A formal representation of feature scattering and its evolution can be obtained by formal concept analysis. Further insight into the problem of feature dispersion can be obtained by studying the distribution of methods among the involved classes. Finally, similarity among features in terms of common methods should be examined, since this could provide a justification for the increased scattering. Each analysis is described separately in the following subsections.

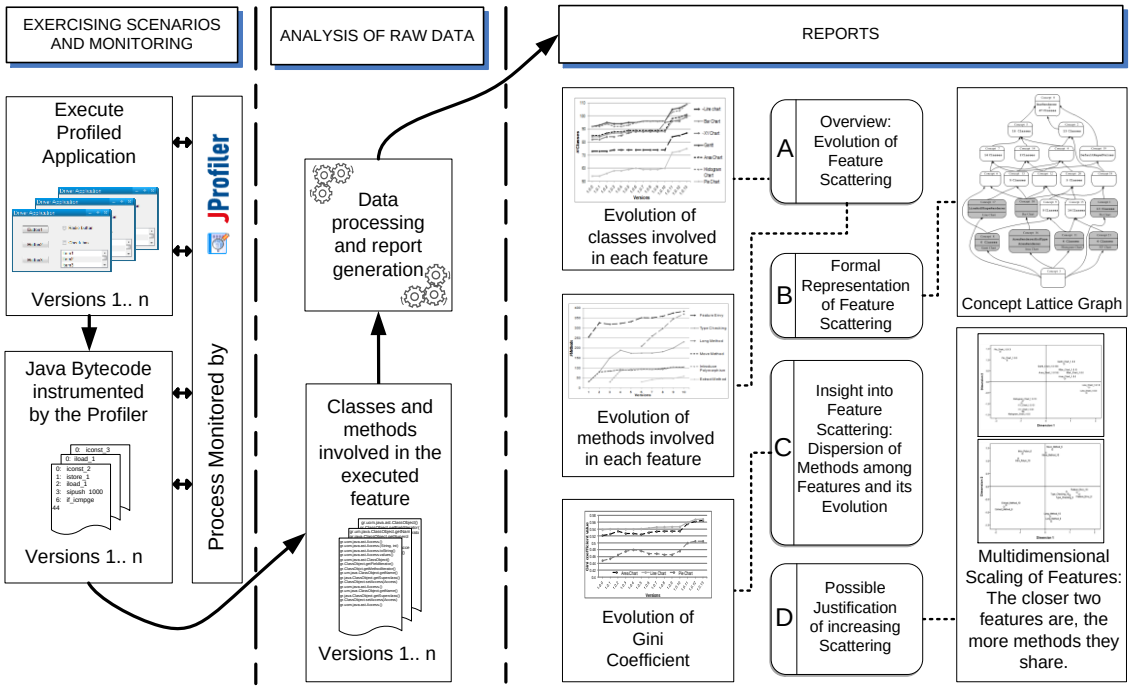


Figure 1 - Data collection and analysis process

3.1.1 Classes involved in the implementation of features

A number of studies conclude that extensive scattering of a given feature in numerous classes hinders not only the tracing of requirements in code, but also the comprehensibility of the underlying flow of events and therefore encumbers extensibility (Eisenbarth et al., 2003; Filho et al., 2006; Garcia et al., 2005; Gibbs et al., 2005; Greenwood et al., 2007; Koschke and Quante, 2005; Revelle et al., 2011; Robillard and Murphy, 2007; Wilde and

Scully, 1995). Furthermore, according to Eaddy et al. (2008), the scattering of feature implementation across the program is statistically connected to the number of defects, and consequently programs with increased feature scattering would probably exhibit more defects and inferior quality. The first goal of our study is to measure the number of classes that contribute to a specific feature by using the metric Count of Number of Classes (CDC) (or methods – CDO) that has been introduced by Filho et al. (2006) and has also been employed in Aspect-Oriented programming (Garcia et al., 2005; Marcus and Maletic, 2003). However, since our goal is to gain insight into the evolution of feature scattering, apart from measuring the number of classes statically, which is for a given snapshot of the examined systems, we also measure the evolution of CDC over a number of successive software versions.

3.1.2 Formal Analysis of Feature Scattering Evolution

Formal Concept Analysis (FCA) deals with binary relations and uses mathematical lattice theory in order to identify meaningful groups of objects that share common attributes (Ganter and Wille, 1999). It has been applied in the field of feature location in order to facilitate the process of tracing specific code units that implement a feature and also to increase the accuracy of the proposed methodologies (Eisenbarth et al., 2003; Poshyvanyk et al., 2007). Poshyvanyk and Marcus (2007) used Formal Concept Analysis to model the relation between methods and attributes, while Eisenbarth et al. (2003) exploited FCA to model relationships between concepts and computational units that implement them. Inspired by those approaches, we have applied FCA in order to model and analyze the relations between features and classes that implement each feature. Our goal is to study the evolution of feature scattering by comparing the concept lattices of different versions. The following paragraph briefly describes the theoretical background and provides an example for better understanding of the underlying notions.

Considering the implementation of a feature f (from the set of all examined features F) by a class c (from the set of system classes C) as a relation $r \subseteq F \times C$ the tuple (F, C, r) is a formal context. A formal context is essentially a binary relation table, indicating which of the classes are involved in the implementation of each feature. A tuple (F_i, C_i) is called a concept if and only if all features in the set F_i (extent of the concept) are implemented by all classes in the set C_i (intent of the concept).

We can define a partial ordering relation for the concepts (F_i, C_i) in a formal context by inclusion: if (F_i, C_i) and (F_j, C_j) are concepts, (F_i, C_i) $\leq$ (F_j, C_j) whenever $F_i \subseteq F_j$ or dually whenever $C_i \supseteq C_j$. Based on this partial ordering, a formal context can be graphically represented as a Directed Acyclic Graph (DAG) where nodes represent concepts and edges denote the relations between them. Usually, the sparse form of the concept lattice is employed, where a particular node n is labeled only with each class $c \in C$ and each feature $f \in F$ that is introduced by node n.

Let us consider the example shown in Figure 2 (Eisenbarth et al., 2003) adapted to illustrate relations among features and classes. Considering features $\{f_1, f_2, f_3\}$ and classes $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$ for a hypothetical system, the set of relations among them can be represented as a two-dimensional matrix also known as Formal Context (Figure 2(a)). The concepts that can be derived from this matrix of relations are shown in Figure 2(b). The most general concept (i.e. the classes common to all features) is denoted by $\top$ while the most special concept (i.e. the features containing all classes) is denoted by $\perp$. Figure 2(c) depicts a graphical representation of the same information known as a concept lattice (sparse form).

	c_1	c_2	c_3	c_4	c_5	c_6	c_7
f_1	x			x		x	x
f_2		x		x	x		x
f_3			x		x	x	x

(a)

$\top$	$\{f_1, f_2, f_3\}$, $\{c_7\}$
Concept 1	$\{f_1, f_2\}$, $\{c_4, c_7\}$
Concept 2	$\{f_1, f_3\}$, $\{c_6, c_7\}$
Concept 3	$\{f_2, f_3\}$, $\{c_5, c_7\}$
Concept 4	$\{f_1\}$, $\{c_1, c_4, c_6, c_7\}$
Concept 5	$\{f_2\}$, $\{c_2, c_4, c_5, c_7\}$
Concept 6	$\{f_3\}$, $\{c_3, c_5, c_6, c_7\}$
$\perp$	$\{\emptyset\}$, $\{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$

(b)

(c)

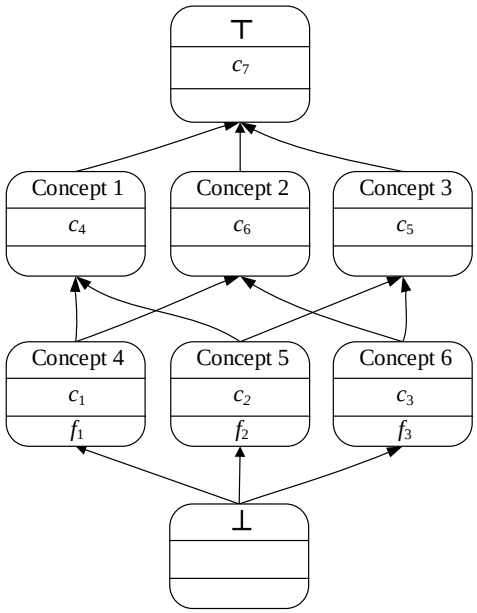


Figure 2 - Example of Formal Concept Analysis.
Formal Context (a), Concepts of the formal Context (b), and the sparse representation of the corresponding concept lattice (c).

From the analysis of a concept lattice, the following two basic pieces of information can be extracted (several other conclusions that can be drawn and which are relevant to feature scattering are presented in Section 3.2.2):

- A feature *f* involves all classes at and above the node at which the feature appears. For example, feature f_1 (introduced in Concept 4) requires 4 classes, which can be found by traversing upwards all paths starting from Concept 4 and ending at the top node, namely c_1, c_4, c_6, and c_7.
- A class *c* is required for all features at and below the node at which the class appears. For example, class c_4 is involved in the implementation of f_1 and f_2.

The aforementioned analysis can be applied to different software versions in order to investigate the evolution of feature scattering, as it will be shown in the case studies (Section 3.2.2).

3.1.3 Distribution of Methods Among Classes

The number of system modules that implement a specific feature might provide a useful insight into the feature's scattering but it is a rather coarse grained analysis due to the lack of information about the way in which methods are distributed over the corresponding classes. In this context, we have recorded the number of methods contributing to the implementation of each examined feature for each of the involved classes. Moreover, we studied the evolution of the distribution over a number of generations.

Textbooks that provide practical guidelines for proper object-oriented design and programming (Arthur J. Riel, 1996; Sharp, 1997) as well as general object-oriented design principles to avoid the creation of "God" or "Blob" classes, advise that a systems' functionality should be distributed over the classes of their specific domain. Under this perspective we suggest that methods implementing a feature should be distributed as uniformly as possible over the involved classes, otherwise classes with a lion's share of feature responsibilities will emerge. The problem often manifests itself in even more worrying form, in the sense that these God classes tend to attract even more functionality over time.

To provide a graphical illustration of this "rich-get-richer" concept, which is frequently and strongly present in technological and social networks (Barabási and Albert,

1999a), in the evolution of feature scattering, Figure 3 shows the number of methods for three classes that contribute to the implementation of the Gantt chart drawing functionality in project JFreeChart, for the first and last examined versions, respectively. The total number of classes that contribute to this functionality is 63. The three classes shown in Figure 3 contain 13% of all methods in the first version and 25% of all methods in the last version. In other words, 3 out of 63 classes ended up in carrying out one quarter of the Gantt chart functionality (measured in methods). This can be regarded as a definite sign of unbalanced distribution of methods among classes involved in the implementation of a feature.

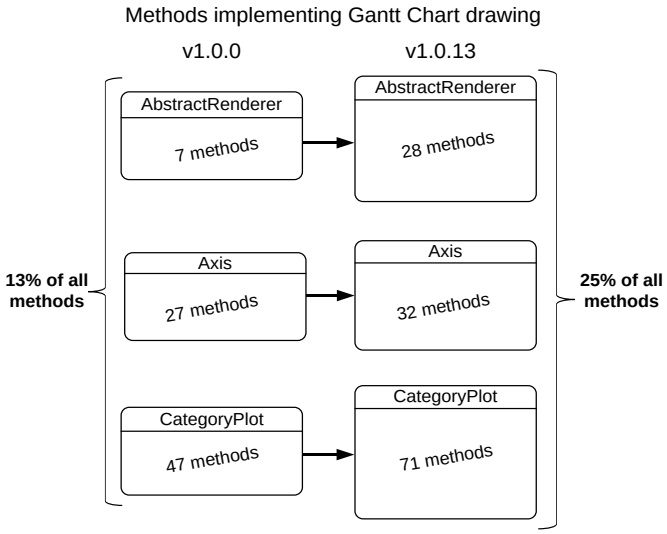


Figure 3 - Evolution of the number of methods for JFreeChart's Gantt chart

3.1.4 Quantifying the Evolution of Method Distribution

The distribution of methods among the classes that contribute to the implementation of a feature could be investigated accurately if it was presented in a dynamic form, where information about all historical versions of the project will be embedded. For this purpose we have employed the Gini coefficient (Gini, 1921), which is a measure of statistical dispersion. The Gini coefficient, a single numeric value between 0 and 1, has been widely employed in a wide range of diverse fields to study the inequality of a distribution. Most commonly it is used as a measure of inequality of wealth in a country but recently it has also been employed in the field of Software Engineering. Vasa et al. (2009) employed the Gini coefficient to quantify the distribution of selected metrics over all system modules as an approach that outperforms the explanatory efficiency of the mean value, which is usually employed. Results of the evolutionary analysis of successive

releases for numerous projects revealed high Gini values resulting from skewed distributions, while the authors highlighted the increased functionality that few classes must carry, making them oversized and rigid. Goeminne & Mens (2011) used the Gini index for the study of the distribution of developer contribution to open source projects. Results came out with high Gini values indicating that the majority of development effort is carried out by a small, core group of people, while the rest of the development community contributes only a fraction of work.

A low value for the Gini coefficient implies a uniform distribution of a measure over the elements of a population. In our context, a low value indicates that the methods contributing to the implementation of a certain feature are distributed in a relatively uniform fashion over the involved classes. On the other hand, a high value indicates an uneven distribution and in the extreme case where the Gini coefficient is close to one, a single involved class would contain almost all of the required functionality for a feature. Essentially the Gini coefficient quantifies in the form of a clean and separate metric the localization and distribution of feature implementation.

Usually the deviation from the perfectly even distribution is depicted graphically by means of the Lorenz curve (Lorenz, 1905) which, in our context, plots the proportion of the total number of methods (y axis) that are cumulatively contained in the bottom x% of the classes. As an example, let us consider the functionality related to the creation of a XY Chart in version 1.0.13 of JFreeChart. Figure 4 shows the cumulative distribution of methods over the cumulative distribution of classes. A perfectly uniform distribution of the methods contributing to the execution of this feature over the involved classes, would be represented by the 45 degree line, usually referred to as the line of equality (x% of the classes contain x% of the methods). The Gini coefficient can be obtained as the ratio of the area that lies between the line of equality and the Lorenz curve over the total area under the line of equality. The further the Lorenz curve from the 45 degree line lies, the higher the Gini coefficient value is. According to the results, the distribution of methods contributing to the XY Chart feature is highly skewed. As it can be observed, around 90% of the classes host 50% of the involved methods, which means that another 10% of the classes host the rest 50% of the methods. The corresponding Gini coefficient in this case is 0.581.

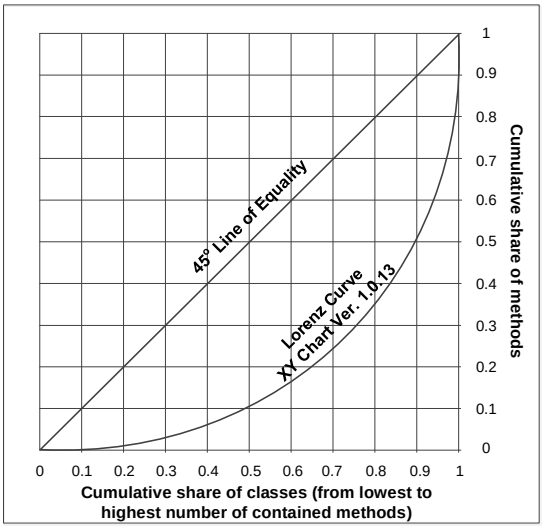


Figure 4 - Graphical Representation of Gini coefficient

Another metric that can quantify the modularity of features is the Degree of Scattering (DoS), originally defined by Eaddy et al. (2007). The Degree of Scattering quantifies the extent by which the implementation of a feature is scattered among many classes. It builds upon the Concentration (CONC) metric that was defined by Wang et al. (2000). The purpose of Concentration is to "quantitatively reflect how much of a feature is in a component" by considering the blocks of code that belong to a software component and are executed by a feature. We believe that without loss of generality, we can consider methods as blocks of code and classes as software components. So in our context Concentration of a class C in a feature F can be defined as:

$$CONC(F,C) = \frac{Methods\ of\ class\ C\ related\ to\ Feature\ F}{Methods\ related\ to\ Feature\ F}$$

Degree of Scattering is a measure of the statistical variance of the concentration of a feature across all program elements in relation to the worst case, where the feature's implementation is uniformly distributed across all program classes (Eaddy et al., 2008).

$$DoS(F) = 1 - \frac{|S_C| \sum_{C \in S_c} (CONC(F,C) - CONC_{WORST})^2}{|S_C| - 1}$$

where:

S_C is the set of classes that contribute to the feature *F*.

$CONC_{WORST}$ is the concentration of the worst case, where the implementation of *F* is uniformly distributed across all involved classes and is calculated as $\frac{1}{|S_C|}$.

As an example, let us consider the following system with 2 classes contributing to one feature (left hand side of). Class *A* contains 4 methods and class *B* contains 2 methods participating to the implementation of feature *F*, as shown in the left-hand side of Figure 5. The values of the Gini coefficient and the DoS metric are also shown. The Gini coefficient is relatively low, since the entire functionality (6 methods) is spread over two classes in a relatively reasonable way. For the same reason, the degree of scattering is relatively high.

Next, we assume that the system evolves to a second version (right-hand side of Figure 5) and that the change consists in adding four methods to class *A*, contributing to the implementation of feature *F*. Clearly, this modification leads to a system where the functionality is spread in a more unbalanced way than in version 1.

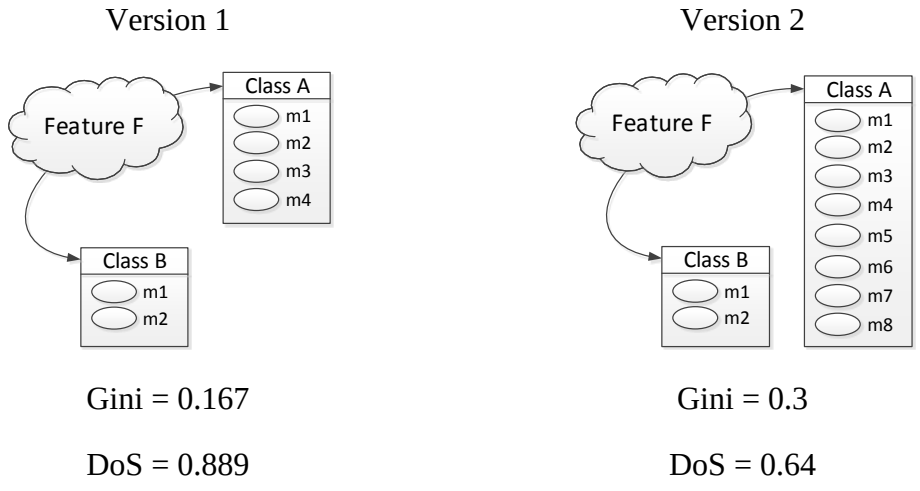


Figure 5 - Evolution of Gini coefficient and Degree of Scattering

As it can be observed from the value for the Gini coefficient and the Degree of Scattering the two measures are somehow antisymmetric, in the sense that the addition of new methods to class A caused an increase to the value of the Gini coefficient and a decrease in the value of DoS. According to the DoS the evolution has led to a system where almost all functionality related to feature F is concentrated into class A (lower scattering). On the contrary the Gini coefficient captures the fact that the functionality related to feature F is distributed in an unbalanced way in the second version, and this is reflected in the increased value of the coefficient.

As it has been mentioned in section 3.1.1 we have opted for absolute measures (i.e. the number of classes and the number of methods) in order to provide a first overview of feature scattering, rather than a measure based upon statistical variance, such as the Degree of Scattering (DoS) across classes. The reason is that, in the context of software evolution, the degree of scattering which quantifies simultaneously both the number of classes implementing a feature and the localization of the implementation in terms of where methods reside, might yield confusing results. Assume for example, that in version i, a number of classes contribute equally (by 25%) to the implementation of a feature and that in the next version i+1 the number of involved classes increases by one, which however, contributes to the implementation of the feature by an extremely small number of methods (e.g. 2%, removed from the other four classes). In that case, a subtle decrease in the degree of scattering would be observed (from 1 to 0.95), as the implementation is localized mostly in the initial classes, whereas a first interpretation should highlight that the number of involved classes has increased and scattering deteriorated. Regarding the use of the Gini coefficient, although its variations are opposite to the variations of the DoS measure, it appears to be more sensitive to such kind of changes which might be valuable when studying feature scattering. For the aforementioned example, the change in the Gini coefficient is much more drastic (from 0 to 0.18) highlighting that the feature's implementation in the second version is non-uniformly spread over the involved classes.

3.1.5 Distance Between Features

So far, the excessive number of classes and methods involved in the implementation of each feature has been recognized as a factor that possibly increases the required effort to understand and maintain the corresponding requirements (Robillard and Murphy, 2007) and even the number of anticipated defects (Eaddy et al., 2008). However, a reasonable question is whether features share classes and methods among their implementations. This would imply that a certain degree of reuse is achieved which reduces development effort and eases maintenance, thus offering a justification for a possibly extended scattering of features in source code. In this section we present results concerning the commonality between features employing a binary similarity measure.

An abundance of distance and similarity measures can be found in the corresponding literature serving a variety of needs (Choi and Cha, 2010). The most

commonly binary measure used for quantifying the similarity between two sets, is the Jaccard similarity which considers the number of elements that are present in both sets as well as the number of elements which are unique in each set (Naseem et al., 2011). The measure that we employed for evaluating the similarity between two features stems from paleontology (Simpson, 1960) and essentially treats two groups as identical if one is a subset of the other. In theory, two features should have a distance equal to zero, if they employ exactly the same set of methods. However, since this might be an unrealistic scenario, we would like to extend the notion of zero distance between two features f_1 and f_2 to the cases where the methods implementing f_1 constitute a subset of the methods implementing f_2.

This measure (Simpson similarity) tends to eliminate the effects of discrepancy in size between two samples and highlights part-whole relations (Simpson, 1960). In analogy to natural evolution where part-whole relations between samples might be informative on the evolution of populations, when assessing the evolution of software, we would also like to gain insight into the degree of reuse among features. In other words, let us consider a feature implemented by certain methods. If a second feature is implemented later, on top of the existing code base, by reusing the already implemented methods (and most probably by adding a number of new methods), this feature should be considered as "close" to the initial one, indicating a high degree of reuse. It might be extremely demanding to expect that a new feature employs exactly the same set of methods (and for this reason we avoided the use of Jaccard similarity) but it would be considered as good practice to reuse all of the existing methods, if possible, and introduce additional methods for the new functionality. This aspect of reuse can be accurately captured by the Simpson similarity.

Under this consideration, the distance of two features according to the Simpson similarity can be calculated as:

$$distance(f_1, f_2) = 1 - similarity(f_1, f_2) = 1 - \frac{|commonMethods(f_1, f_2)|}{\min(|methods_{f_1}|, |methods_{f_2}|)}$$

where:

$|methods_{f_1}|$ corresponds to the number of methods implementing feature f_1

$|methods_{f_2}|$ corresponds to the number of methods implementing feature f_2, and,

$|commonMethods(f_1, f_2)|$ represents the number of common methods between features f_1 and f_2

To obtain a graphical representation of the similarity among features and to provide a tool for assessing whether features are becoming more distant during the evolution, implying reduction in the degree of reuse among them, we propose the use of multi-dimensional Scaling (MDS) for visualizing distances. MDS (Cox and Cox, 2008) is an approach that allows representing information contained in a set of data by a set of points usually in a two-dimensional Euclidean space. These points are arranged spatially in a way that geometrical distance between points reflects the numerical measure of distance between the examined data items. In other words, what multidimensional scaling does is to find a set of vectors in a p-dimensional space (in our case coordinates in a 2-dimensional Euclidean space). As a result, the axes of the extracted plots correspond to the dimensionality of the employed space. The orientation of the axes is arbitrary and any rotation of the plane will give rise to another valid solution. The interpretation of dimensions is at the discretion of the researcher who attempts to identify what is varying as we move along the two axes (Bartholomew et al., 2008). However, the output of multidimensional scaling may be valuable even if one cannot ascribe meaning to the axes, since the graphical representation can facilitate the comprehension of patterns in the data (i.e. one might be able to identify clusters of closely placed points).

Conventional MDS application would lead to two separate Euclidean distance models, one for each of the examined versions. To understand the nature and extent of association between the examined features, the proximity of points in the derived space needs to be interpreted (Singh, 2007). However, the orientation of the axes for each MDS chart can be arbitrary, hindering the comparison between the two versions. Therefore, we adopted a different approach in which all examined features of both versions are fed into a single analysis. Consequently, the resulting diagrams illustrate the distances among all features for two versions, allowing us to investigate the evolution between the similarity of features and consequently the reuse among them.

Multi-dimensional scaling has been previously used by Fisher and Gall (2004) in order to visualize the proximity between Problem Report Data. The distance between two problem reports was defined as the number of commonly modified files to fix both problems, while groups of feature related reports have been formed enabling the identification of hidden dependencies between features. The dependencies among features have been visualized by means of MDS for the Mozilla project and for the years 1999-

2002. In a more general context, Kuhn et al. (2008) employed MDS to map software artifacts to a two-dimensional space employing the vocabulary of each artifact in order to measure the distance among them.

3.2 Case Studies on Feature Scattering

In this section we illustrate the aforementioned techniques and measures in order to study the evolution of feature scattering on four open-source projects, namely JFreeChart, JDeodorant, Jmol and jEdit. JFreeChart is an open-source chart library (JFreeChart, 2013) which has been constantly evolving since 2000. JDeodorant, is an Eclipse plug-in that automatically identifies design problems, known as "bad smells", and eliminates them with appropriate refactoring applications (JDeodorant, 2014). It has been constantly evolving for more than five years as a project of the Computational Systems and Software Engineering Laboratory at the Department of Applied Informatics, University of Macedonia, Greece. Jmol is a Java viewer for chemical structures such as crystals, materials and biomolecules in 3D, which has been evolving since 2002 (Jmol, 2013). jEdit is a text editor especially built for programmers that can be extended by numerous plug-ins and has been evolving since 1998 (JEdit, 2013). The evolution of size characteristics (lines of code (LOC), number of classes (NOC) and number of methods (NOM)) for the examined versions of all projects is shown in Table 2. LOC refers to lines that contain at least one statement, method signature or class definition including lines with comments and blank ones.

Table 2 - Size Characteristics of the Examined Versions/Projects

JFreeChart														
Ver.	***1.0.0***	***1.0.1***	***1.0.2***	***1.0.3***	***1.0.4***	***1.0.5***	***1.0.6***	***1.07***	***1.0.8***	***1.0.9***	***1.0.10***	***1.0.11***	***1.0.12***	***1.0.13***
kLOC	126	126	130	134	138	142	146	157	157	158	161	168	170	177
NOC	465	466	478	493	502	505	516	540	540	540	546	561	563	587
kNOM	5.4	5.4	5.5	5.7	5.9	6.0	6.1	6.6	6.6	6.6	6.8	7.1	7.1	7.4

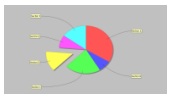
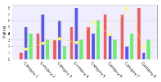
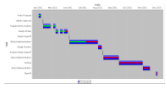
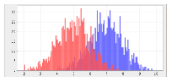
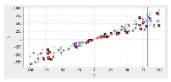
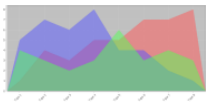
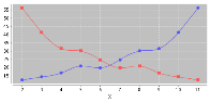
JDeodorant										
Ver.	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***
kLOC	5.1	8.3	14.2	17.2	18.3	18.8	19.8	21.2	24.4	24.4
NOC	53	85	97	104	105	129	134	147	158	170
kNOM	0.51	0.68	0.90	0.990	1.00	1.07	1.12	1.20	1.35	1.46

Jmol													
Ver.	***11.0.0***	***11.0.2***	***11.2.0***	***11.2.3***	***11.2.5***	***11.2.10***	***11.2.14***	***11.4.1***	***11.4.6***	***11.6.1***	***11.6.10***	***11.6.20***	***11.6.27***
kLOC	71	71.1	84	84.1	84	84.2	84.4	96	96.3	108	108.7	108.7	109
NOC	279	280	333	334	333	333	333	387	387	403	403	403	403
kNOM	5.3	5.3	6	6.04	6.04	6.04	6.05	6.5	6.53	7	7	7	7

jEdit												
Ver.	***3.2.2***	***4.0.1***	***4.0.10***	***4.1.0***	***4.1.1***	***4.1.5***	***4.1.9***	***4.1.15***	***4.2.0***	***4.3.0***	***4.4.1***	***4.5.0***
kLOC	63.9	77.6	81.6	88.2	87.2	93.2	100	101	107	130	123	125
NOC	256	295	295	319	312	330	344	344	367	457	453	469
kNOM	3.4	4.02	4.16	4.4	4.5	4.7	5	5.05	5.3	6.7	6.6	6.67

Seven features have been selected for the analysis of JFreeChart, six for JDeodorant, six for jEdit and five for Jmol. Table 3 briefly outlines the examined features of the four projects. It should be mentioned that the selected features cannot be considered a canonical set (according to Kothari et al. (2006) a canonical set consists of a small number of features that are as dissimilar as possible to each other, yet are representative of the entire functionality). Since one of the goals is to investigate the reusability of classes, the selection of features should not focus only on distinct functionalities.

Table 3 - Examined features for each project

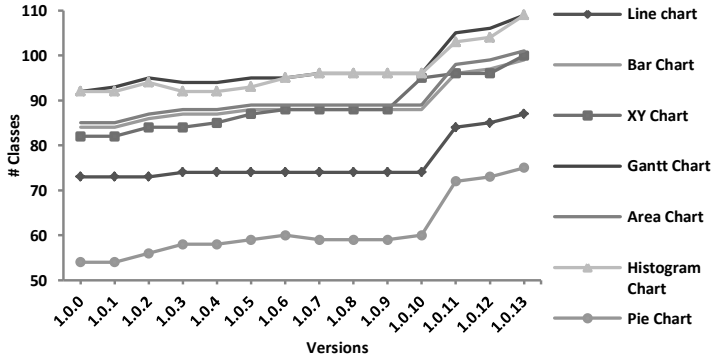
Project	Feature Description	
JFreeChart	Pie Chart	
	Bar Chart	
	Gantt Chart	
	Histogram	
	XY Chart	
	Area Chart	
	Line Chart	
JDeodorant	Feature Envy	Identification of methods suffering from feature envy code smell
	Long Method	Identification of methods which are extremely long, complex and non-cohesive
	Type Checking	Identification of conditional statements that select an execution path based on a specific state (lack of polymorphism)
	Move Method	Elimination of a selected feature envy code smell through move method refactoring application
	Extract Method	Elimination of a selected long method code smell through extract method refactoring application
	Introduce Polymorphism	Elimination of a state checking code smell by introducing polymorphism
Jmol	Open mol file	Opening of a mol file and rendering of the chemical structure in the screen.
	Open cif file	Opening of cif file and rendering of the chemical structure in the screen.
	Change view to Bottom	Change 3D view side of the object to bottom angle
	Display molecule surface	Displaying the Connolly surface of the molecule.
	Export jpeg	Exporting of the rendered molecule view in jpeg format.

jEdit	Open a Java file	Selection and opening of a java file through a File Chooser.
	Select and Replace All	Selection of a non-reserved word and replacement of all of its occurrences.
	Add 2 markers in Java code	Insertion of two markers in two inconsecutive lines of code.
	Navigate through markers	Navigation through the added markers by using the appropriate menu option.
	HyperSearch	Searching for a specific word. HyperSearch lists all occurrences of the search string in a floating window instead of locating the next match.
	Write Java code and save file	Typing of a specific class and saving it as a Java file in order to enable the highlighting of Java reserved words.

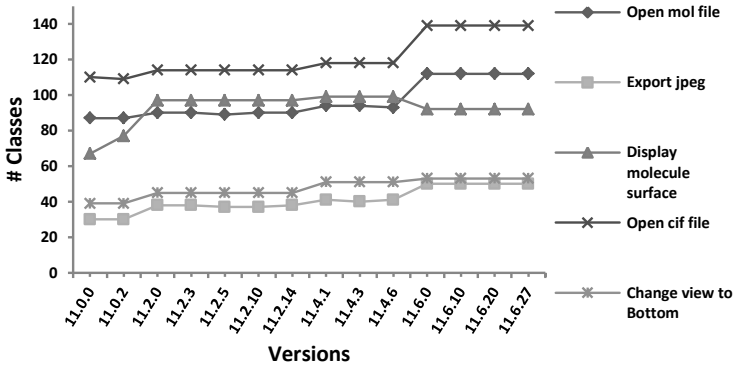
The results from the application of the techniques/measures described in sections 3.1.1 – 3.1.5 on the four case studies are presented in the following subsections in the same order.

3.2.1 Evolution of involved classes

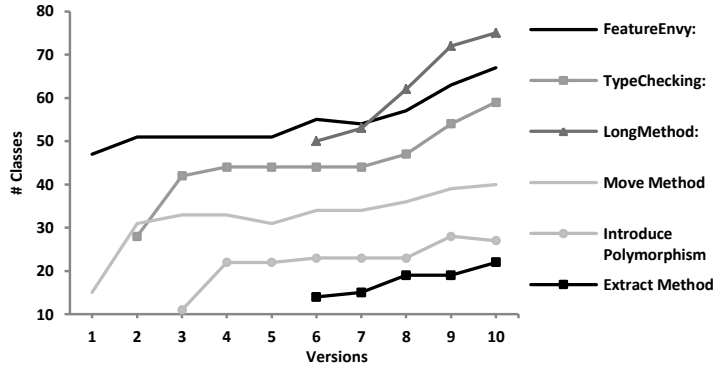
In Figure 6 we illustrate the number of classes which are involved during the execution of a certain feature, for all versions of JDeodorant, JFreeChart, Jmol and jEdit.



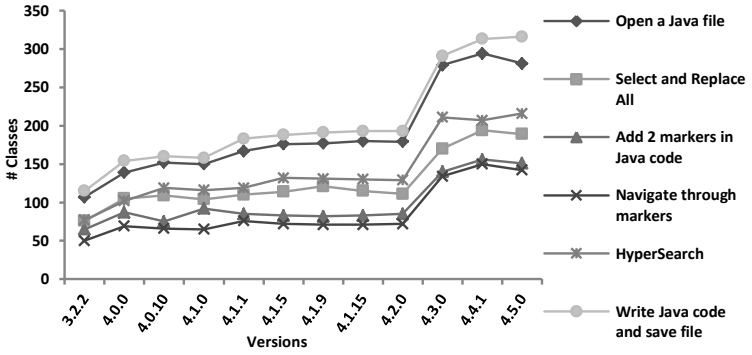
(a) JFreeChart



(c) Jmol



(b) JDeodorant



(d) jEdit

Figure 6 - Number of classes involved in the implementation of each feature.
Projects (a) JFreeChart, (b) JDeodorant, (c) Jmol and (d) jEdit

The experimental results for all projects and for almost all features indicate that the number of classes employed in the implementation of features is monotonically increasing as the projects evolve. A first striking observation is for example, the fact that for writing and saving a Java source code file with jEdit, over 300 classes may be involved in the last examined version. From the reengineering perspective, if a feature's implementation should be extended, adapted or simply analyzed, the maintainer might have to go through

a large number of these classes in order to be able to modify the source code and maintain its external behavior, with profound impact on his productivity. The rate of increase of the involved classes in the implementation of each feature is not constant and this might be caused by various reasons. For example, in project JFreeChart, an abrupt increase in the number of classes involved in the implementation of the selected features occurred between versions 1.0.10 and 1.0.11. According to the release notes this might be related to a significant enhancement of functionality by introducing a new chart theming mechanism. The same observation holds for the transition from version 4.2.0 to 4.3.0 in project jEdit. The release notes revealed that in version 4.3.0 a significant number of enhancements, bug fixes and additions of new functionality have taken place.

The findings regarding the evolution of the number of methods that implement a selected feature are similar: the number of methods involved in each feature appears to be very high and increases with the passage of versions. For example, more than 550 methods might be invoked when drawing a Histogram chart in JFreeChart and close to 400 methods are involved in identifying Feature Envy code smells employing the JDeodorant tool. An impressive number of 1467 methods are invoked in order for a newly-typed Java source file to be saved in jEdit, while Jmol needs over 1000 method invocations to read, render and display a chemical structure that is stored in a ".cif" file.

3.2.2 Concept Lattices

The application of Formal Concept Analysis for the first and last version of project JFreeChart yielded the concept lattices shown in Figure 7. At this point it should be mentioned that one of the major drawbacks of concept lattices is that they do not scale well, consequently in Figure 7 a reduced form of the sparse concept lattice has been employed, i.e. class names are not shown except for the cases where it is necessary for our discussion. Full versions of the concepts lattices are moved to the Appendix chapter A3. The highlighted nodes are concepts which introduce the examined features and thus can serve as the basis for observing the evolution in the number of classes involved in each feature.

According to the semantics of concept lattices applied in our case, the following pieces of information can be derived from the observation of the graphs (Eisenbarth et al.,

2003). Their use can be extended for the interpretation of the evolution in the scattering of features and the reuse of components:

- A feature *f* requires all classes at and above the node at which the feature appears in the sparse lattice representation. For example, feature Line Chart (Concept_19) requires 73 classes in version 1.0.0, which can be found by traversing upwards all paths starting from Concept_19 and ending at the top node. In version 1.0.13, the number of classes involved in the implementation of Line Chart increased to 87.

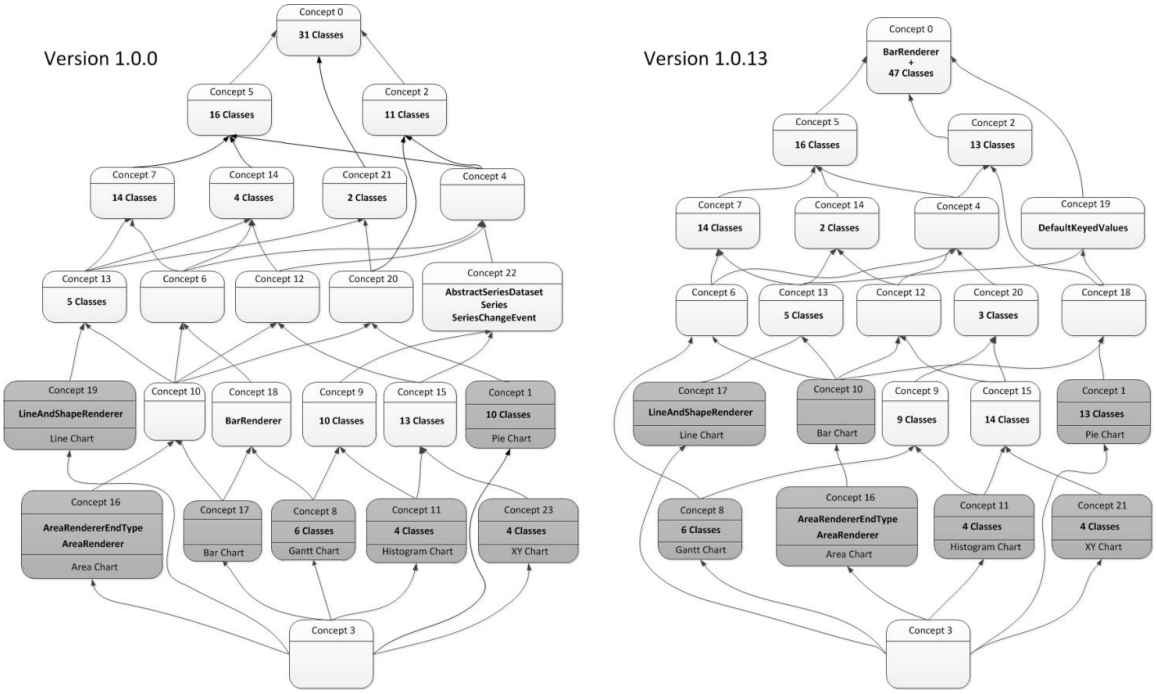


Figure 7 - Concept Lattices for the first and last examined versions of JFreeChart

- A class *c* is required for all features at and below the node at which the class appears in the sparse lattice representation. For example, class `BarRenderer` in version 1.0.0 (Concept_18) is involved only in the implementation of Bar Chart and Gantt Chart, indicating a relatively low degree of reuse for the class. On the other hand, the same class appears in the top node of the concept lattice in version 1.0.13, implying that this class contributes to the implementation of all features, exhibiting a tremendous increase in its reuse.
- A class *c* is specific to exactly one feature *f*, if *f* is the only feature on all paths from the node at which *c* is introduced to the bottom element. For example, in version 1.0.0 the classes which are involved only in the implementation of feature Pie Chart

(Concept_1) are 10, while the number of unique classes for the same feature in version 1.0.13 has risen up to 13.

- Classes jointly required for n features $f_1, f_2, \ldots, f_n$ are classes belonging to concepts which lie on the intersection of all paths from the node at which features $f_1, f_2, \ldots, f_n$ are introduced, to the top element. For example, features Gantt Chart (Concept_8), Histogram Chart (Concept_11) and XY Chart (Concept_23) in version 1.0.0 share classes `AbstractSeriesDataSet`, `Series`, `SeriesChangeEvent` (lying at Concept_22) as well as all classes at concepts 5, 2, and 0. In total, 61 classes are commonly used in the implementation of these three features in the first version of JFreeChart. From the examination of the concept lattice of the last version it can be found that the number of common classes increases to 80.
- Classes required for all functionalities lie at the top element (Concept_0). For version 1.0.0, 31 classes are employed in all examined features, while in version 1.0.13, the number of common classes increases to 48.

3.2.3 Distribution of Methods

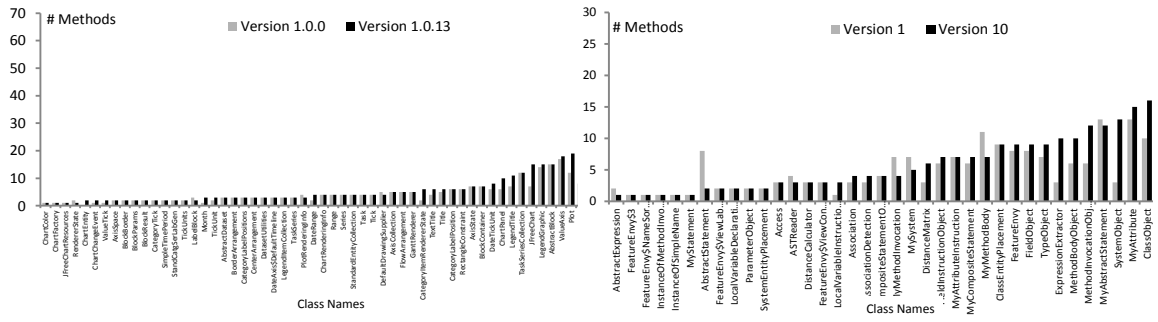
Figure 8 displays the distribution of methods among the classes involved in the implementation of a feature for the first and last version of all examined projects. More specifically, Figure 8(a) depicts the distribution of methods over classes for a Gantt Chart creation with JFreeChart, Figure 8(b) illustrates the same distribution for Feature Envy functionality of JDeodorant while frames (c) and (d) correspond to the exporting of a jpeg image in Jmol and the opening of a Java file in jEdit, respectively. To understand whether this distribution remains unchanged as the system evolves or not, the number of methods that are used in the first (light bars) and the last version (dark bars) is shown for each of the involved classes. (The figures display only the classes that exist in both the first and last version of the examined projects).

An observation that can be made for all projects is the skewed nature of the distribution of methods over classes. For example, in frame (a) it can be observed that most of the involved classes host less than 10 methods contributing to the examined functionality while a relatively small number of classes host over 20 involved methods. The same observation holds for (c), and (d) while in (b) this phenomenon still exists but it is less intense. A characteristic example of the unbalanced distribution of class

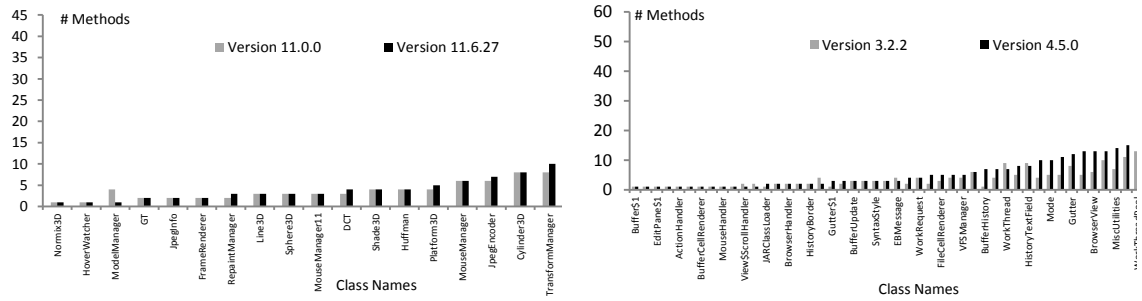
responsibilities is the class `Buffer` that supports the opening of a Java file in jEdit 3.2.2 with 57 methods, and the class `CategoryPlot` from JFreeChart 1.0.0, which supports the creation of a Gantt chart with 47 methods.

A second remark is related to the methods that are introduced during software evolution. From the figures it becomes apparent that classes which already hold an increased number of methods act as attractors to the newly inserted methods, a phenomenon similar to the rich-get-richer rule underlying preferential attachment (Barabasi et al. 2000). For example, in JFreeChart, 20% percent of the total number of additional methods (121 methods, comparing the first and the last examined version), have been added to a single class (class `CategoryPlot` contributed to the Gantt chart functionality 47 methods in the first version and 71 methods in the last one). An exception to this phenomenon is class `Buffer` in jEdit, where despite the fact that it held the majority of methods in the first examined version (57), this number decreased to 40 in the last examined version.

The aforementioned observations imply phenomena which could be rather harmless. For example, the overconcentration of methods in a single class among those implementing a feature might be due to the nature of the involved functionality. On the other hand, highly skewed distributions of methods among the classes involved in the implementation of certain functionalities, which become even more skewed as the systems evolve, could represent inefficiencies of the initial architecture which might go unnoticed by other means, such as metric values or design flaws. In other words, this form of preferential attachment, where new methods are attached to classes that have already a large number of methods contributing to the same feature, might lead to serious maintenance issues. The evolution of these distributions is studied in a more formal manner in the next subsection employing the Gini coefficient.



(a) JFreeChart - Gantt Chart (b) JDeodorant - Feature Envy

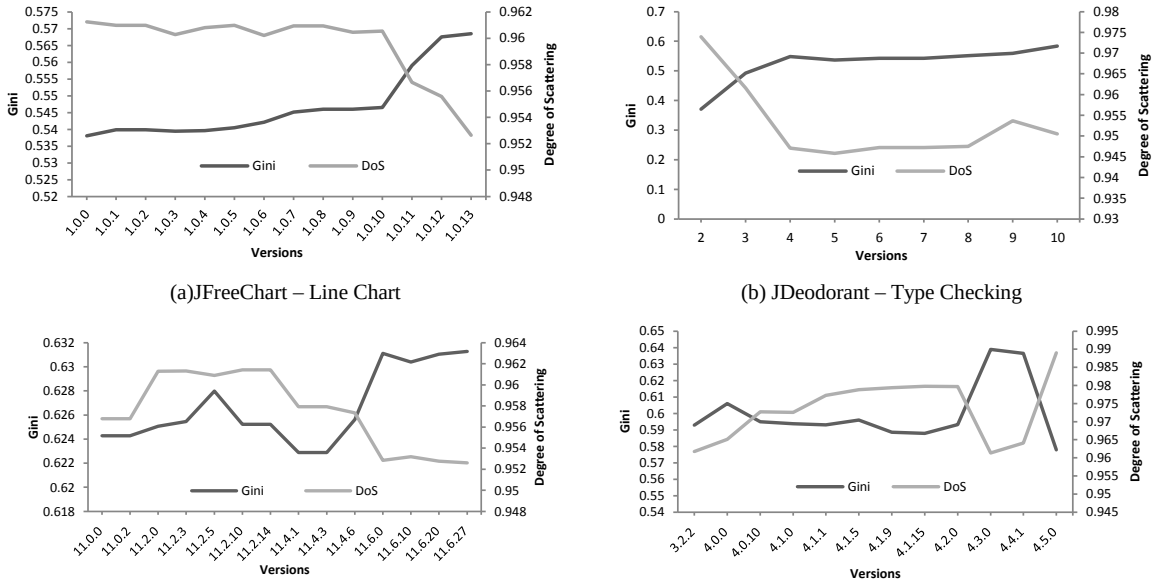


(c) Jmol - JPEG Export (d) jEdit - Open java file

Figure 8 - Distribution of methods over classes for selected features

3.2.4 Gini Coefficient and Degree of Scattering

The evolution of the Gini coefficient over the versions of all examined systems in shown in Figure 9, for selected features. The values range from 0.37 for the second version of project JDeodorant (feature Type Checking) to 0.64 for version 4.3.0 of project jEdit (feature Open Java File). While an absolute value for the inequality in a distribution might be difficult to interpret, its tendency over time might be informative. As it can be observed the value of the Gini coefficient is generally increasing with the passage of software versions, indicating that the distribution of methods among the classes involved in the corresponding feature becomes more unbalanced over time. As already explained, this means that classes with a large share on the total functionality (in terms of methods) attract even more methods as software evolves, becoming a sort of "God" classes in the context of the examined feature. Figure 9 displays also the evolution of the Degree of Scattering. As it can be readily observed, trends of Gini and DoS are quite opposite and in most cases an increase in the value of Gini can be matched to a decrease of DoS and vice versa. Intuitively this observation makes sense by considering the nature of the two metrics. Gini coefficient is analogous to the inequality of a distribution and increases if this inequality deteriorates (i.e. classes that already hold many methods, attract more new methods than the classes with fewer methods), while Degree of Scattering is analogous to the diffusion of methods across classes and the more diffused the methods become, the higher the value of DoS is.



(a)JFreeChart – Line Chart

(b) JDeodorant – Type Checking

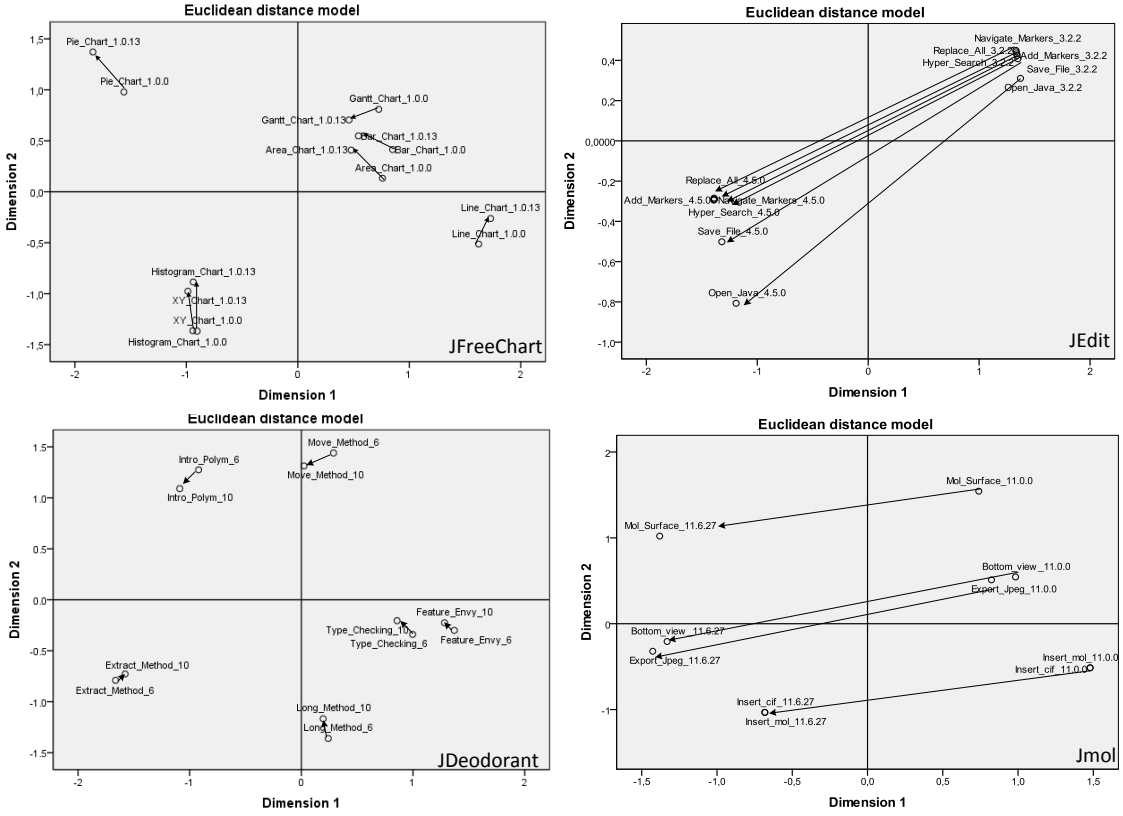
(c) Jmol – Open mol file

(d) jEdit – Open Java file

Figure 9 - Co-Evolution of the Gini coefficient and Degree of Scattering

3.2.5 Multi-dimensional scaling for feature distance visualization

Figure 10 illustrates the output of multi-dimensional scaling for two versions (initial and last one) for all four examined projects, employing as distance the aforementioned Simpson measure. The axes of the generated two-dimensional maps could be interpreted as follows: As we move along Dimension 1 from the right to the left, in projects jEdit and Jmol, it is clearly evident that any variations are due to the passage of software versions. On the other hand, differences along Dimension 2 can be attributed to variations in functionality. Points that come closer indicate an increase in method reuse, while points that diverge indicate the opposite. Similar observations hold for JFreeChart and JDeodorant but are less striking.



* the initial version for JDeodorant is v6 since this is the first version in which all of the examined features are present

Figure 10 - Multidimensional Scaling

The MDS output for JFreeChart (top left corner) depicts three primary clusters of features located at the upper left, lower left and right areas of the diagram. The clusters of features which can be identified based on their distances, are rather reasonable, considering the underlying data structures on top of which they are built. Line Chart, Area Chart, Bar Chart and Gantt chart functionalities are all dependent on a `CategoryDataset` class or subtypes of it. Histogram and XY chart functionalities employ the `XYDataSet` data structure, while the Pie Chart is rather independent, using the `PieDataSet` structure.

Concerning the overall evolution of the system, it can be observed that rather small changes occurred in the distances between the features from the first to the last version. A more careful examination can reveal for example, that the distance between the pair of features Line and Pie Chart, or Histogram and XY Chart, increased with the passage of generations. For example, Histogram and XY Chart are extremely close to each other in version 1.0.0, since they share 365 methods, out of 390 methods contained in the XY Chart, which is the "smaller" of the two features. In version 1.0.13, the number of common methods raised to 492, followed by a concurrent increase of the "smaller" feature which

remains the XY Chart with 520 methods, leading to a slightly higher distance between the two features. The overall evolution of similarity, as the arrows depict, points that the examined features are becoming less similar by employing fewer common methods.

From the Euclidean distance model for JDeodorant (bottom left corner), the most striking observation concerning clusters that can be identified visually, is the cluster containing features Feature Envy, Long Method and Type Checking, at the lower right area of the diagram. These features correspond to code smell identification functionalities which share a number of methods in their implementation and are rather distinct from the other three features corresponding to refactoring application functionalities. Concerning the overall evolution, an improvement in the design properties can be observed, since many of the features appear to converge, in the sense that the corresponding points in the diagram move slightly towards the center of the diagram as the system evolves, implying an increase in the degree of reuse.

The case of jEdit (top right corner) presents also an interesting evolution. It appears from the MDS chart that all features are relatively close to each other, indicating a large degree of reuse among features. This is true both for the first and the last examined version. As an example, in version 3.2.2, features 'Add Markers' and 'Navigate Markers' have 50 classes in common, out of the 65 and 50 classes of the first and second feature, respectively. The same holds for version 4.5.0 where the two features share 139 classes out of 151 and 142 classes, respectively. On the other hand, there is a large displacement between the dots of the first and last version implying limited reuse between the same features as software evolved. For example, feature 'Add Markers' in version 3.2.2 and the same feature in the last version share only 30 out of the initial 65 classes and on top of that, 86 new classes have been added (i.e. 30 out of the 151 classes of the last version).

A similar phenomenon is apparent on the MDS chart for Jmol (bottom right corner). The extent of reuse among features remains relatively stable across versions, whereas the features of the first version share limited classes to the same features of the last version, implying low reuse and the addition of a large number of new classes to each feature.

3.2.6 Impact of Refactorings on the distribution of feature implementation

The application of preventive maintenance activities such as refactorings is rarely considering the impact on feature scattering, whereas the relocation of methods, the

creation of new classes and methods definitely affects the implementation of features. For example, let us consider that a method, contributing to the implementation of a particular feature is moved from a source class A to a target class B after applying the Move Method refactoring. In the extreme case where B was not involved at all in the implementation of the feature prior to the refactoring, moving the method will increase feature scattering in the sense that a larger number of classes will be involved. On the other hand, if both classes are part of the feature implementation and class A contains a larger number of involved methods, moving the method will lead to a more balanced distribution of the functionality across the classes, reflected on a decrease of the Gini coefficient.

To evaluate the impact that selected refactorings have on feature scattering we conducted an experiment by applying consecutive refactorings of the same type for a specific feature of a given system. In particular, we selected one version of each of the examined open-source systems, and employed JDeodorant (2014) in order to identify refactoring opportunities for Extract Class, Extract Method and Move Method refactorings (Fowler et al., 1999) that affect classes which are involved in the feature of interest. The Gini coefficient and the Degree of Scattering have been measured before and after the application of each refactoring, allowing us to assess whether the refactoring improved scattering or not. Figure 11 illustrates the successive values of the Gini coefficient and DoS, resulting from the application of all identified refactorings for a selected feature and for all four examined systems.

The effect of the Extract Class refactoring depends on the number of methods extracted as a new class as well as on the number of methods involved in the feature under study in the original class. For example, the application of Extract Class refactorings in projects JDeodorant and jEdit appears to have a positive impact on the Gini coefficient by relieving in most cases heavily loaded classes (i.e. classes with a large share on the total number of involved methods) from several methods which are moved to the extracted class. On the other hand, in project Jmol all Extract Class refactorings moved a very small number of methods to the new class (usually one), leading to the creation of an additional 'poor' class, i.e. a class with very small participation in the implementation of a feature and thus deteriorating even more the distribution of functionality. According to these observations, if the goal is to achieve a balanced distribution of methods, Extract Class

refactorings should be applied only if a substantial amount of functionality is to be moved to the new class.

Extracting a method from an existing one (which is involved in the implementation of a feature) will affect the distribution of methods and as a result the Gini coefficient, depending on whether the class hosting the original method resides in a class that has a large or small share on the total number of methods implementing a feature. The addition of the extracted method to a class that had a relatively small number of involved methods reduced the Gini coefficient in all cases, while the upward trends are due to the extraction of a new method in a class that was already "rich" in terms of the number of involved methods. As a guideline, from the perspective of feature functionality distribution, one could suggest to avoid performing the Extract Method refactoring for classes that already have a large share of the total number of involved methods.

Moving a method to another class will improve (deteriorate) the distribution of methods involved in a certain feature and consequently decrease (increase) the Gini coefficient, in case the target class to which the method is moved is a 'poor' ('rich') class in terms of method concentration. An exception is the move of a method to a class that was not involved in the implementation of the examined feature prior to the refactoring. In this case, the addition of another "poor" class deteriorates the distribution of methods. This interpretation explains all variations in the charts of Figure 11 for Move Method refactorings.

The aforementioned guidelines do not aim to substitute the already existing criteria or heuristics for assessing the impact of a refactoring. For example, the application of an Extract Method refactoring might be valuable, in the case of a large, complex and non-cohesive method, regardless of the effect on the Gini coefficient. However, these rules might be considered in parallel since in most cases the suggestions are in line with common sense for achieving better design quality. For example, one would rarely perform an Extract Class refactoring if the concept of the extracted class is too limited (i.e. if the number of methods in the new class is very small) and this is in absolute agreement to the observation made earlier regarding the distribution of methods implementing a feature.

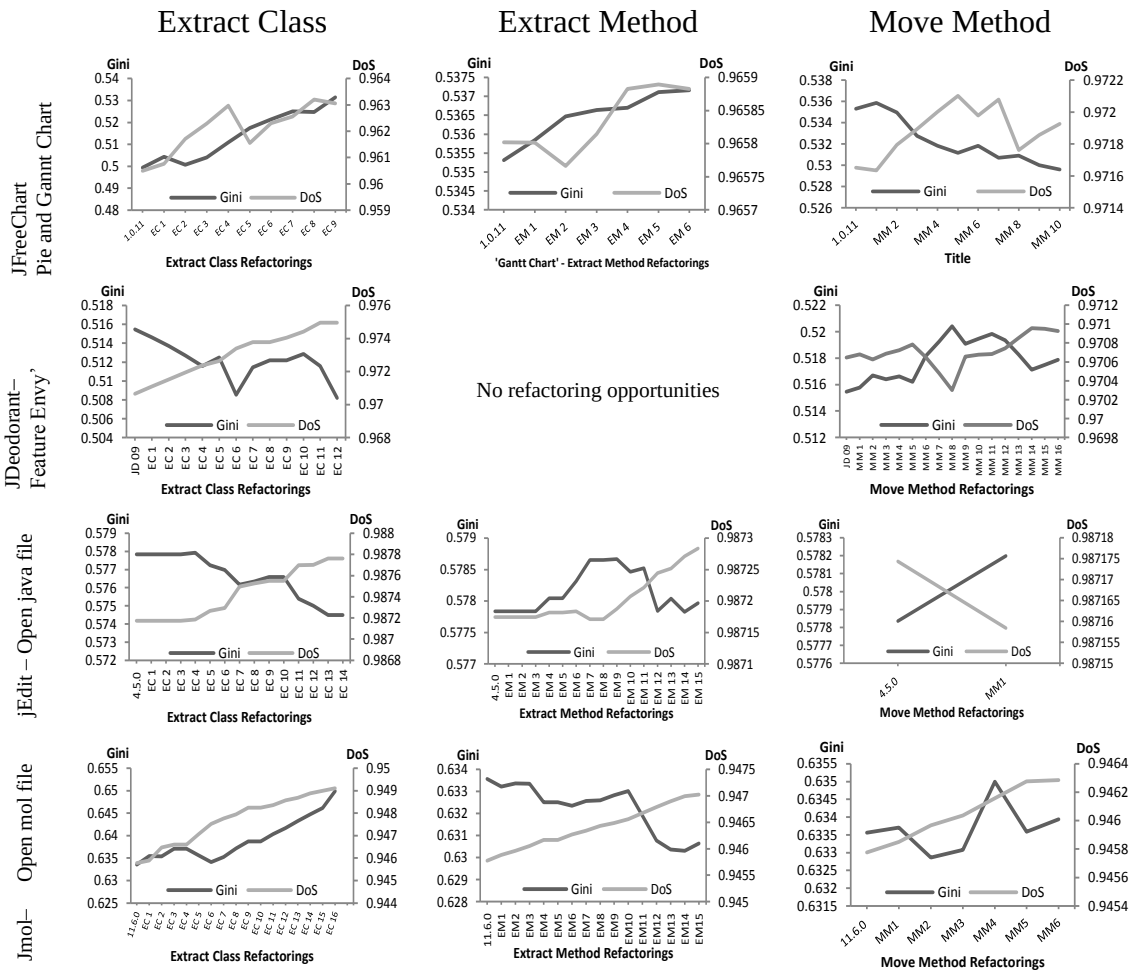


Figure 11 - Impact of refactorings on Gini Coefficient and Degree of Scattering

4 Evolution of Object Oriented Software Networks

Prediction is very difficult, especially about the future.

Niels Bohr, Danish physicist

4.1 Context description

4.1.1 Representing software as graph

Graphs, also known as networks are an extremely powerful abstract tool that is frequently used in the field of computer science for the representation of a variety of structures. From router and communication networks for the simulation of data packages, to website networks for the categorization, indexing and searching of web page content.

In the context of the proposed model we treat object-oriented systems as networks of interconnected classes. Nodes correspond to classes, including abstract and concrete ones. In particular we model the network of classes according to the Law of Demeter (Lieberherr and Holland, 1989). An edge linking two nodes (source and target) indicates that between the corresponding classes there is an 'allowed' (according to the Law of Demeter - LoD) relationship. In the LoD context, a relationship is allowed in case the source class contains references to the target class which are either attributes, local variables to which instances of the target class are assigned, method parameters and return types of the target class type. The resulting graph $G<V, E>$, where V and E is the set of nodes (vertices) and edges, respectively, is directed.

During the evolution of an object-oriented design over a number of versions, new nodes (classes) might be added in any version. All other classes are considered as existing nodes. Given these two types of nodes, the following types of edges have to be taken into account in the development of a prediction model:

- edges leaving from new and reaching existing nodes

- edges leaving from new and reaching new nodes
- edges leaving from existing and reaching any other node..

4.1.2 Examined systems

Evolutionary trends have been investigated for ten open-source systems, which are also the systems against which the proposed prediction model has been evaluated. The systems have been selected according to the following criteria:

- they should be open-source projects since the source code has to be analyzed in order to extract its network representation
- they should be written in Java since our tool (Chaikalis et al., 2014) is currently capable of analyzing Java source code.
- they should have varying sizes to test more effectively the scalability of the model. (The selected projects' size ranges from 55 to 3201 classes)
- a good number of versions should be available, in order to allow an adequate capturing of the project's historical evolution.

A brief description of each project is provided in Table 4. The size characteristics of the networks corresponding to the first and last version of each examined project are shown in Table 5 along with the number of successive versions that have been used in the analysis.

Table 4 – Projects that have been undergone evolution analysis

aTunes	Audio player and music library organizer that supports many types of audio formats (aTunes, 2013)
FreeCol	Turn-based strategy game similar to Civilization with graphical user interface (FreeCol, 2013)
JDeodorant	Eclipse plug-in for the detection and elimination of design flaws by means of refactoring (JDeodorant, 2014).
JEdit	Cross-platform text editor, which can be customized by plugins (JEdit, 2013)
JFreeChart	Chart creation library with an extensive variety of supported charts (JFreeChart, 2013)
Jmol	Viewer that visualizes chemical structures in 3D (Jmol, 2013)
Weka	Machine learning software suite that contains a collection of visualization tools and algorithms for data analysis and predictive modeling. (Weka, 2013)
HFSExplorer	Application that reads and manipulates Mac-formatted hard disks and disk images. (HFS Explorer, 2013)
Presto	Facebook's distributed SQL query engine for running interactive queries against big data sources. (Presto, 2014)
Jetty	Web Server and Servlet Engine with support for SPDY, WebSocket, OSGi and JNDI technologies. (Jetty, 2014)

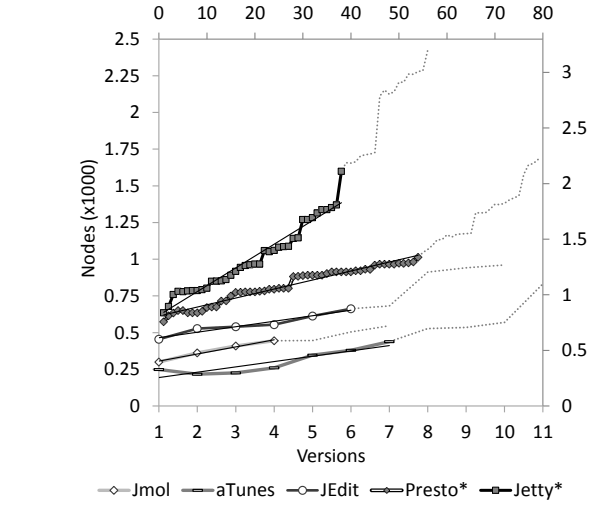
Table 5 – Project Characteristics

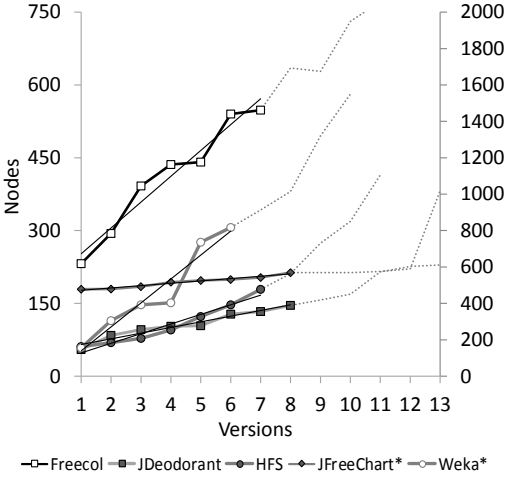
#	Name	Examined Versions	Nodes		Edges	
			First	Last	First	Last
1	aTunes	11	249	832	570	1979
2	FreeCol	10	232	772	732	3975
3	JDeodorant	13	55	229	184	780
4	JEdit	12	455	960	1035	2278
5	JFreeChart	13	478	1018	1575	2892
6	Jmol	8	316	547	615	1130
7	Weka	13	156	1550	599	5884
8	HFS Explorer	11	61	414	125	1233
9	Presto	79	758	2219	2604	7595
10	Jetty	58	838	3201	1652	7952

The number of nodes (classes) for each examined version and for all analyzed projects is graphically depicted in Figure 12. The evolution of each project is split in two periods:

1) the training period on which the evolution has been monitored to configure the model parameters (continuous line) and

2) the testing period against which the forecasting power of the model was tested (dotted line). A linear interpolation is fitted on the evolution of each project during the training period. The goodness of fit (R^2) for each project is shown below the charts. As it can be observed the nodes increase in an almost linear fashion in all cases (the goodness of fit R^2 to a linear function ranges from 0.83 to 0.98). Thus, to avoid unnecessary complexity to the proposed model, we obtain the number of new nodes for each forthcoming software version as the mean value of new nodes per version, of all past software versions. However, it should be noted, that the proposed model does not necessarily rely on a linear fit on the node evolution to estimate the number of nodes that should be added in each version. If historical data can be better captured by a different type of growth model (e.g. logarithmic or exponential), the number of nodes can be estimated accordingly.





Project	aTunes	FrCol	JDeo	JEdit	JFC	Jmol	Weka	HFS	Presto	Jetty
R^2	0.83	0.95	0.95	0.95	0.96	0.98	0.93	0.95	0.96	0.94

Figure 12 - Evolution of node count and R^2 of the corresponding trendlines

In the following sections all aspects of the proposed model will be presented, attempting to justify each decision in relation to the historical trends which are pre-sent in the evolution of the examined systems as well as the corresponding phenomena which have been analyzed in other domains. An overview of the entire model is depicted graphically in Section 4.3 Beyond the parameters which are dictated by the proposed model, we leave for each choice that has to be taken, a certain degree of randomness, since we acknowledge the fact that there are dimensions in the software evolution process which are not strictly following rules or past distributions.

The reason for which we propose a graph based model for the prediction of software evolution is that numerous factors come into play which cannot be accounted for

by a simple regression-based model. To illustrate the inherent limitation of regression as a means of forecasting the evolution of software architecture parameters, such as the afferent coupling of a class, we illustrate in Figure 13 a regression-based forecast of the in-degree.

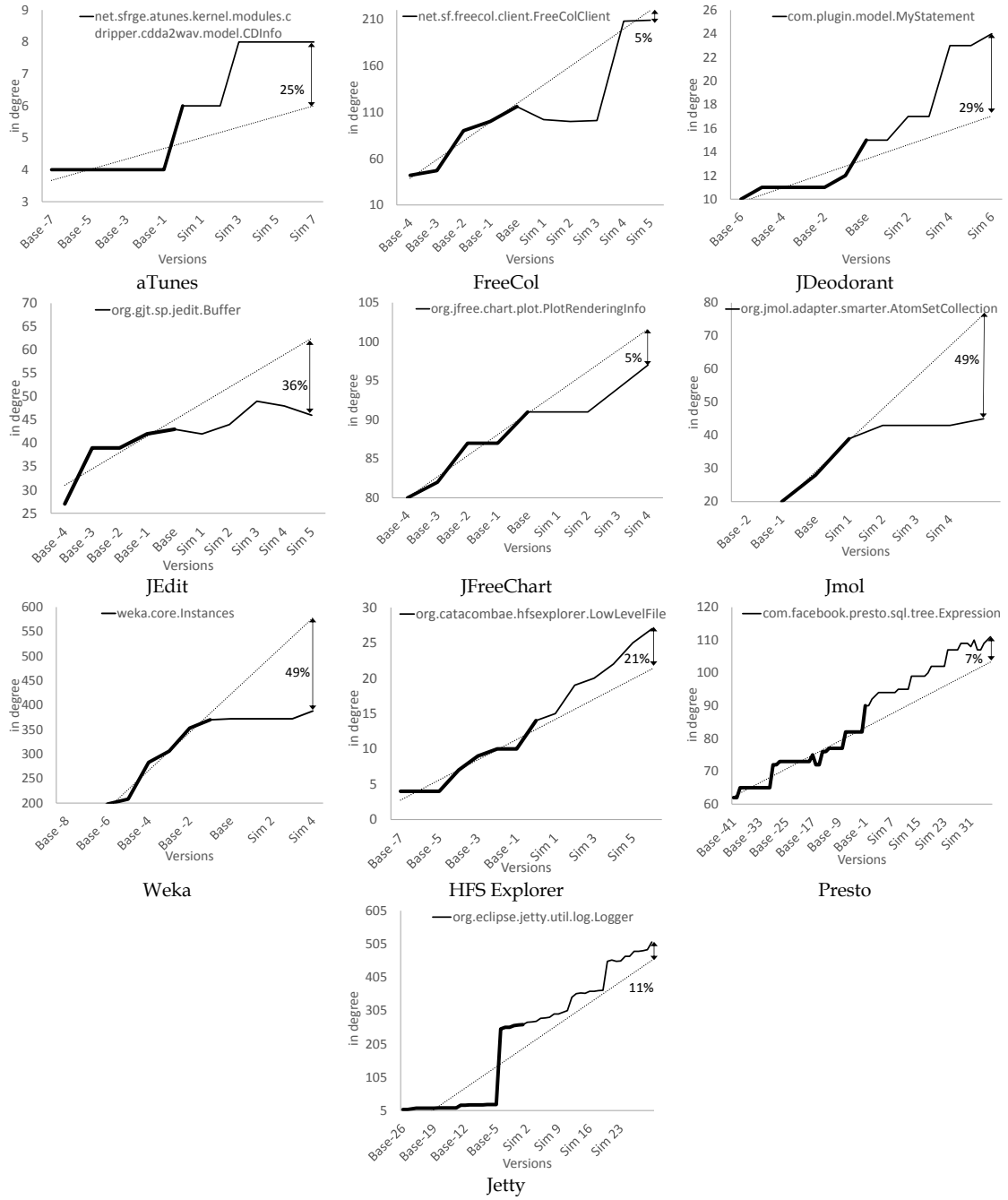


Figure 13 - Regression based forecast of the in-degree of specific classes.
The selected classes exhibited the largest variation in their in-degree between the first and last examined version, for each of the examined projects. The thick line corresponds to the training period. The dotted line corresponds to the linear regression on the training period along with the projection up to the final version.

In particular, the evolution of the in-degree for classes that exhibited the largest variation in their in-degree between the first and last examined version, is shown. The plot includes also the trendline corresponding to a linear function (providing the best goodness of fit), fitted to the data of the versions that form the training set. The size of the training set is set to 50-60% of the total number of versions of each project. The limited ability of the regression model to correctly predict future changes is evident in most cases. The predicted in-degree for the last version differs from the actual in-degree and it is apparent that regression cannot account for the variation in the in-degree that takes place during the evolution of a class.

4.2 Description of the evolution model

A network topology evolves over time by two mechanisms: addition/deletion of nodes and addition/deletion of edges. All other network properties emerge from the resulting network topology. What we have attempted in our approach is to model the node arrival process, the edge creation process (selection of source and target nodes) and the edge removal process (which might also result in node removal).

There is a plethora of properties which can be measured during the evolution of a network, related to all perspectives from which a graph can be studied such as size, connectivity, cycles, coloring, cliques, shortest paths, centralities, etc. From all available graph proper-ties, our survey of numerous models that have been examined in the context of other disciplines revealed that the node degree is the primary measure of interest and common in all studies. For this reason, we have incorporated the study of degree distributions in our model as well.

However, as it will be made clear in the evaluation, models which only consider mechanisms that explain changes in the node degree do not manage to predict accurately the network topology. To improve the accuracy of the edge and node creation processes we have incorporated the effect of source/destination node age because we have observed that this property exhibited regular patterns during the evolution of the examined systems whereas other properties did not exhibit any noteworthy motifs.

The model has been further refined with two other functions related to the software nature of the examined networks: a) the duplication mechanism to reflect the package-level structure of each system and b) the restrictions on node behavior dictated by domain rules.

4.2.1 Preferential attachment and duplication

One of the most striking observations that has been repeatedly made for several technological, biological and social networks, despite their inherent differences, is that very often they all exhibit a common pattern in their degree distribution. The pattern consists in a degree distribution that is heavy tailed and usually follows a power-law, a behavior that has been considered as a strong indication of a scale-free network (Barabási and Albert, 1999b; Taube-Schock et al., 2011), i.e. a network that is self-similar at all scales. In other words, the fraction of nodes $P(k)$ having k edges decays following a power law $P(k) \sim k^{-a}$, with the value of parameter a depending on the nature of the application domain. A different interpretation of the above phenomenon states that:

The probability that a node n has degree d(n) is proportional to $d(n)^{-\alpha}$

$$P(d(n)) = Cd(n)^{-\alpha} \tag{1}$$

where C is a normalization constant and $\alpha > 1$.

If we take the logarithms on both sides of (1), we end up with the following equation:

$$\log(P(d(n))) = -a\log(d(n)) + \log(C) \tag{2}$$

which manifests itself as a straight line in log-log plots of d versus $P(d)$, with $-\alpha$ being the negative slope of the line.

To investigate whether a power law is also valid for object-oriented software systems we plot in Figure 14 the in-degree distribution of the classes for the last version of each examined system. In particular, the figure shows the cumulative number of classes vs. the class in-degree on a log-log plot. The almost straight form of these distributions confirms the presence of a heavy tailed degree distribution. As it would be reasonable to expect, very few classes have a large in-degree (providing services to a large number of dependent clients) while quite many classes are accessed by a limited number of other modules leading to a low in-degree. Similar conclusions regarding the existence of power laws in large software systems have been drawn by other studies (Concas et al., 2007; Fortuna et al., 2011; Louridas et al., 2008; Taube-Schock et al., 2011; Wheeldon and Counsell, 2003).

The generation of scale-free properties and power law distributions is usually attributed to the presence of Preferential Attachment (PA), which postulates that when new nodes are added to an existing network forming edges to existing nodes, the number of edges attracted by each target node is proportional to the target's in-degree (Barabási and

Albert, 1999a). In an object-oriented setting, this evolution model, commonly known as the "rich-get-richer" rule, implies that classes having already a central role in the system (i.e. are a form of "God" classes providing services to numerous clients (Arthur J Riel, 1996)) act as attractors for new classes that join the system.

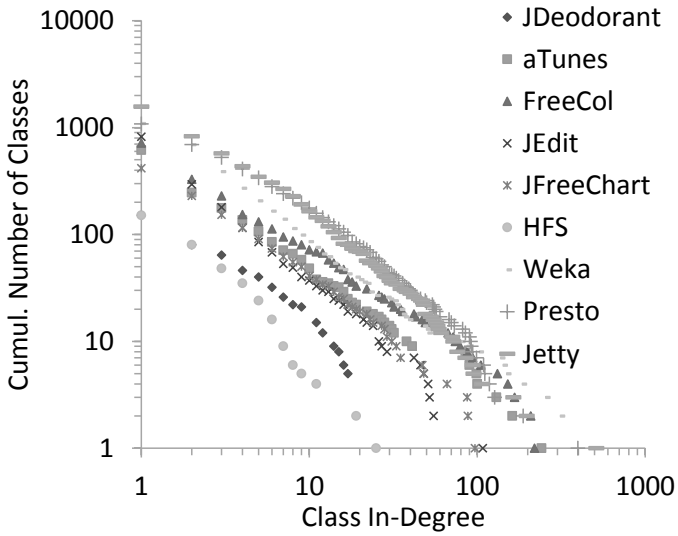


Figure 14 - In-Degree distribution for the examined systems

Moreover, it has been observed that this phenomenon becomes more intense as software evolves. With the passage of versions, a larger percentage of new classes are linked to higher in-degree classes. Stated in a different way, "important nodes (in software systems) stay important", a fact which is considered as a sign of stability in the design in the work of Paymal et al. (2011). To validate empirically whether Preferential Attachment actually holds for the systems under study, we illustrate in Figure 15 the number of edges attracted by a node versus the in-degree of this node. The number of edges attracted by a node of a particular degree has been calculated as follows:

During the evolution of a system, there are numerous nodes having a particular in-degree at each time point. Moreover, a node having a particular in-degree value at a certain time point *t* might have a different in-degree at a later time point. The average number of edges attracted by nodes having in-degree *k* should be calculated considering all versions (Leskovec et al., 2008). The corresponding formula is:

$$AvgNumOfEdges = \frac{\sum_{forAllVersions} \left| \left\{ e(u,v) : d^{-}(v) = k \right\} \right|}{\sum_{forAllVersions} \left| \left\{ v : d^{-}(v) = k \right\} \right|} \quad (3)$$

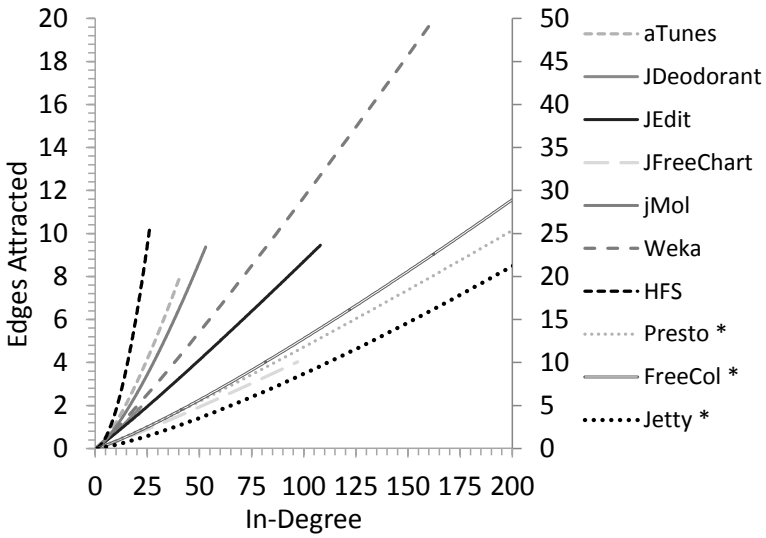
where:

$d^-(v)$ is the in-degree of node *v*

e(*u*, *v*) is an edge from node *u* to node *v*

The numerator represents the number of edges towards nodes of in-degree *k* and the denominator is the number of nodes having in-degree *k*.

As it can be observed from Figure 15 most new edges are attached to classes that have a large in-degree, while low in-degree classes appear to attract a limited number of new edges. Thus, it can be concluded that the preferential attachment mechanism is strongly present in the evolution of the examined software systems.



* Projects Presto, Freecol & Jetty are displayed against secondary y- axis

Figure 15 - Node in-degree vs. number of edges attracted

Based on these observations we adopt the Preferential Attachment model as a core element in the construction of our network prediction model. Once a new node is added in a version and the number of outgoing edges that will be formed is determined (as it will be explained next), the target nodes are selected with a probability that is proportional to the class in-degree. According to Barabási and Albert (1999b) this probability *P* for node *v* depends on the in-degree of the node and is equal to:

$$P(v) = \frac{d^-(v)}{\sum_{i=1}^{|V|} d^-(i)} \tag{4}$$

Where |*V*| corresponds to the cardinality of the vertex set of the graph.

However, the consideration of the preferential attachment mechanism is not sufficient for modeling the evolution of software systems. In evolution models of other domains (biological, social or technological), most systems are represented as networks in a rather 'flat' way, that is, all new nodes are assumed to belong to the same community neglecting distinct sub-communities which might exist. However, in software systems, structure in terms of packages or namespaces heavily influences the way that a network evolves and thus should not be ignored (Li et al., 2013). According to our experiments applying the preferential attachment model considering all system classes as a single community with-out structure, limits significantly the explanatory power of the model. To confront this issue we adopted, on top of the preferential attachment, the so called Duplication Model according to which the behavior of a new node that is introduced to a network copies the behavior of a previously introduced node (Chung et al., 2003; Chung and Lu, 2006; Kleinberg, 2000; Kumar et al., 2000; Valverde and Sole, 2003b).

Following this model, we construct for each analyzed system a data structure holding all previous node behaviors. By the term 'behavior' we mean for each 'new' node that has been introduced in the past, the software packages to which it has created edges, the number of edges that have been created to each package and in which package the node itself has been added. Consequently, when modeling future evolution, we select a previous behavior and copy it.

What remains open is the specific target class of the selected package to which the new class should be attached. For this parameter we rely on the preferential attachment model. As a result, we employ a combination of these two models: we rely on the Duplication model for determining the coarse-grained aspects of package selection (and as a consequence we introduce domain knowledge into the evolution model) and for the more fine-grained process of class selection we employ the preferential attachment model (which in turn ensures the preservation of the power law phenomena).

4.2.2 Modelling node activity

Obviously, an object-oriented system does not evolve simply by adding new classes. The already existing classes also form new relations among them in order to access new functionality or data. Neglecting the evolution of edges emanating from existing nodes

would lead to a highly inaccurate model. To incorporate this aspect in our model, three issues should be addressed:

a) Determination of the number of edges to be created
b) Selection of the source node for each new edge
c) Selection of the target node for each new edge

In order to determine the number of new edges that should be formed by existing nodes, we rely on past data and particularly on the number of new edges created by existing nodes in all previous versions. In Figure 16 the cumulative frequency of the number of new edges (emanating from existing classes) is shown for all examined systems. As an example, point *A* in the plot indicates that (for project FreeCol), in 60% of the cases, up to 200 new edges have been formed by existing nodes. During the simulation of future evolution the number of new edges to be formed by existing nodes is sampled from this distribution through inverse transform sampling (Devroye, 1986).

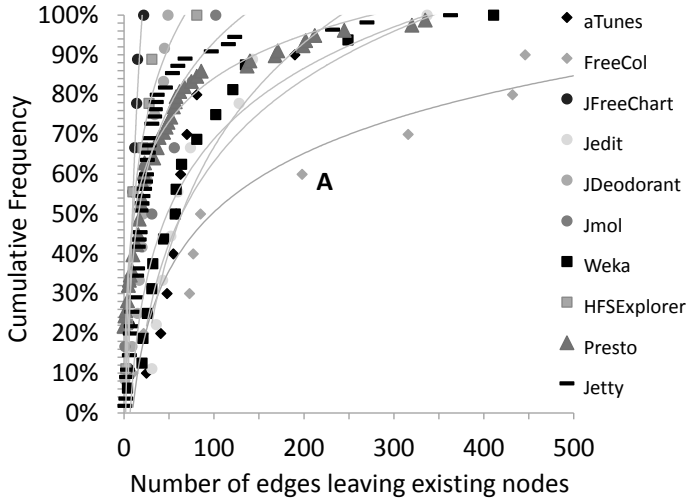


Figure 16 - Distribution of the number of new edges leaving existing nodes

To perform inverse transform sampling from a cumulative distribution function (CDF) the first step is to fit an appropriate regression function *f* on the given data points and obtain its inverse function f^{-1}. Next, a random *y* value in the range [0, 1) is chosen and applied to the inverse function yielding the corresponding *x* value. In our case, the obtained *x* value indicates how many edges should be formed by existing classes for a particular simulation step.

The source node for each new edge is selected by employing the previously mentioned Duplication Model since even for edges created by existing nodes, the system's

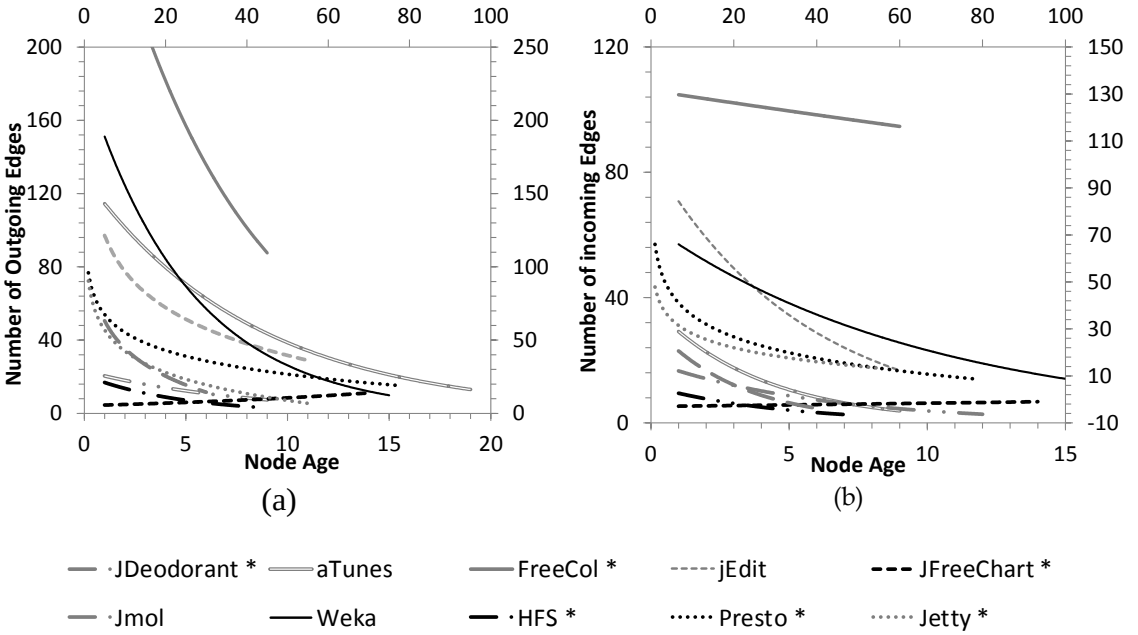
structure in terms of packages should be respected. As in the previous subsection, the duplication model determines also the target packages to which the source node will form edges. The target node for each new edge is selected according to the Preferential Attachment model as previously, since large classes in terms of incoming relations act as attractors also in this case.

When new classes are introduced to an object-oriented system they form relationships to existing (older) classes, but very often these new classes also collaborate with each other. This is reasonable, since classes are not added as individual modules but as pieces of code offering certain functionality and thus they are exchanging messages among them. Quite often, the new classes are added in the form of an entire new package with a complete architecture as it has also been observed in the work of Li et al. (2013). For these reasons, a prediction model should also consider the edges that are formed between new nodes, i.e. leave a new (source) class and reach a new (target) class. For the determination of the number of edges between new nodes, we follow the already explained strategy by sampling from the distribution of past data concerning edge creation among new classes in each version.

4.2.3 Effect of class age

The study of software evolution in terms of the underlying networks, over several versions and different projects, reveals another interesting feature regarding node activity: The majority of edges are departing from "younger" nodes, with their age measured in number of versions. This remark is in agreement with similar observations that have been made for the evolution of software systems in other studies (Bhattacharya et al., 2012; Zheng et al., 2008). The creation of outgoing edges is equivalent to the access of functionality and data offered by other classes, in order to accomplish the functions that the source class should deliver. Evidence from the systems that have been examined clearly support this viewpoint: Figure 6a displays, for all examined systems, the total number of new outgoing edges created from nodes which at any time point of their history had the corresponding age (the lines correspond to exponential fits on the actual data). It becomes clear that among the created edges, most edges are departing from "young" nodes. This fact is taken into account in the prediction model, by incorporating the node age in the Duplication Model applied for the selection of the source nodes. As a result when a node

has to be selected as the source of a new edge, the model does not simply rely on past node behaviors but restricts the selection among nodes with the required age, which is obtained by inverse transform sampling. In a similar manner the age of a class affects its probability of being the target node for a new edge. As already explained this is mainly determined by the preferential attachment model, according to which classes with a large in-degree act as 'attractors' for new edges. However, we have observed that the raw application of the Barabási and Albert model (Barabási and Albert, 1999a) leads to an excessive loading of large classes with additional incoming edges. To mitigate this issue and extract a more accurate model we do take into account the age of the target node as well. Fig. 6(b) illustrates the total number of incoming edges versus the age of the target node, for all past data of the examined projects.



Projects JDeodorant, FreeCol, JFreeChart, HFS, Presto and Jetty are displayed against secondary x- & y- axis

Figure 17 - Distributions based on node age.
(a) Total number of outgoing edges vs. node age and
(b) Total number of incoming edges vs. node age

Among the created edges, most incoming edges reach "young" nodes, but this tendency is less intense than for outgoing edges. It should be noted that this observation is not in contrast to the Preferential Attachment mechanism. The large number of incoming edges attracted by "young" nodes (which usually have low in-degree) occurs because we take into account edges in the entire history of a project. By employing inverse transform sampling on the corresponding cumulative distribution, we obtain for each new edge to be

created the desired age for the target node. Consequently, in the proposed software evolution model both the appropriate in-degree (according to the preferential attachment model) as well as the appropriate age (according to inverse transform sampling) of the target node are considered.

4.2.4 Edge removal

Our empirical investigation indicated that during the evolution of the examined systems, apart from the introduction of new nodes and the addition of new edges, a significant number of edges have also been removed. Table 6 shows the number of edges that have been added and removed during the evolution of each examined system (over the versions that have been taken into account).

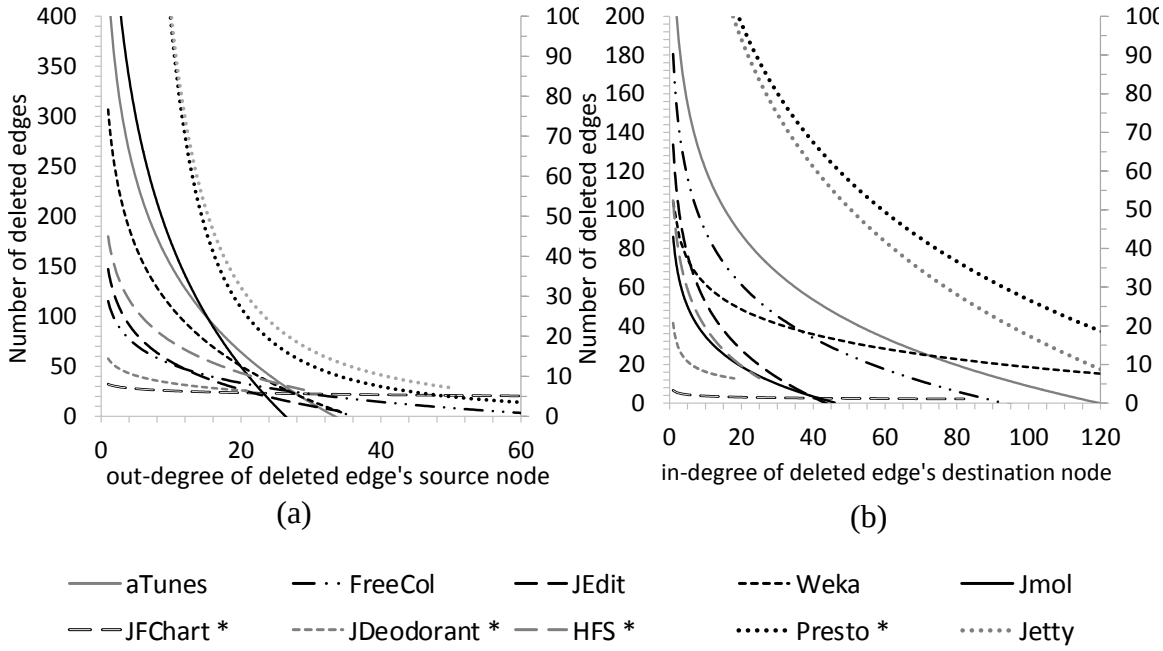
Table 6 - Number of added and removed edges for the examined projects and versions

Project	Edges Added	Edges Removed
aTunes	5621	3344
FreeCol	4834	1591
JDeodorant	765	169
JEdit	2590	1347
JFreeChart	522	134
Jmol	1406	888
Weka	9727	2735
HFS	1470	361
Presto	15346	7190
Jetty	12570	5216

It is apparent that the removal of edges cannot be neglected during the construction of a software evolution model. On the contrary, the number of removed nodes is insignificant (for all projects the percentage of deleted nodes versus the number of nodes in the last version ranges around 2-3%).

In order to simulate an edge removal one should decide on the source and destination node of the edge to be removed. In the context of our model, the parameters that could have been considered in such a selection are the nodes' age and degree. The history of all examined systems revealed that there is no clear relationship between the age of the source and destination node, whenever an edge is removed. On the other hand, there appears to exist a relatively strong relation between the out-degree of the source node (and the in-degree of the destination node, respectively) and the number of removed edges.

Figure 18 illustrates the number of deleted edges versus the (in/out) degree of the deleted edge's source node and target nodes, for all examined software projects.



*Projects JFreeChart, JDeodorant, HFS Explorer, Presto and Jetty are displayed against secondary y- axis

Figure 18 - Distributions based on the number of deleted edges.
(a) out-degree of deleted edge's source node and (b) in-degree of deleted edge's destination node. Each curve corresponds to an examined project.

It appears that as the out-degree of a node becomes higher (possibly implying the accumulation of functionality or data in the corresponding class), fewer edges are removed from that class (Figure 18a)). The same holds for the relation between the number of deleted edges and the in-degree of the destination class (Figure 18(b)): the largest percentage of removed edges seem to reach nodes that have a relatively low in-degree. Once again, this past information is taken into account in the proposed model, by converting these distributions into cumulative form and then by applying the aforementioned inverse transform sampling methodology to extract for each edge to be deleted the sought degree of the destination and target class. Next, an edge is removed for a pair of nodes that satisfy the required degrees. In case the removal of an edge leads to an unconnected node (orphan node), the corresponding node is also removed. To determine the number of edges to be removed in each simulated version, as already applied in other aspects of the model, we sample from the cumulative distribution of past edge removals.

4.2.5 Introduction of domain knowledge

Willinger et al. (2009) vividly showed that interpreting phenomena at the network level, such as the presence of scale-free properties, without proper handling of domain-specific issues might be a cause for enormous confusion. For example they have debunked the myth of scale-free internet formed by interconnected routers. Numerous researchers observed an apparently striking common characteristic, according to which their node connectivities follow a scale-free power-law distribution (Barabási and Albert, 1999a). However, this conclusion stems from "unhealthy" data: For the case of router networks, neglecting domain specific issues, such as IP aliases and the inherent inability of the traceroute tool to reveal the actual node degree might generate misleading results. As a result, the excitement generated by straight-line behavior in log-log plots of degree vs. frequency as evidence for power-law phenomena might be based on unreliable data sets. The authors suggest *to rely on domain knowledge and exploit the details that matter when dealing with highly engineered systems* (Willinger et al., 2009).

Obviously, object-oriented designs are also engineered systems where particular tradeoffs have been considered during their construction. These tradeoffs are exactly what differentiates them from social networks. As an example, in a social setting, there is no cost in creating a friendship relation to another person and in most cases, the more friend connections the better. On the other hand, in an object-oriented system, forming a relation between two classes increases coupling and the associated impact on the architecture is usually taken into account.

Consequently, to improve the accuracy of the prediction models for future software evolution trends, domain knowledge should be incorporated. Here, we do not aim to provide a full coverage of rules that govern the evolution of software. This would be a great challenge given that different development teams set different priorities and follow distinct principles. However, it will be shown that even by considering one common practice in OO design can largely improve the forecasting accuracy.

The Dependency Inversion Principle (Martin, 2003) (and in part the Open Closed Principle) in object-oriented design states that details in the design (i.e. concrete classes containing implemented methods) should depend on abstractions (such as abstract classes and interfaces) rather than having abstractions depend upon details. The second formulation of the Dependency Inversion Principle states this explicitly, as "*Abstractions*

should not depend upon details" (Martin, 2003). This well-known principle promotes the separation of interface and implementation, by having concrete subclasses extend the abstractions, achieving increased flexibility and reusability.

Moreover, abstractions are considered stable (especially in the case of pure abstract classes or interfaces) in the sense that they are depended upon by many other classes (Martin, 2003) and thus there is a good reason for keeping them unmodified. Under this perspective, designers should strive to preserve the stability of abstract classes so that changes in them do not ripple up to their clients causing them to change as well. In terms of relations among classes, the conformance to the stability principle dictates that abstract classes should not often form relations to other concrete modules causing a change in their interface. This is not to claim that abstract classes are not associated to other classes but rather to emphasize that once an abstraction has been introduced to the system, the outgoing relations to other modules should not change drastically in later versions. In the context of the proposed model this design rule (Rule 1) implies that creating new edges that leave abstract classes (and reach other system classes) should be avoided. The same holds for Java interfaces whose stability is even more important (an outgoing edge from an interface would occur by having another class as parameter in one of its methods).

An even more clear violation of this principle occurs whenever a superclass contains a reference to any of its subclasses. According to Riel's design heuristics, "*base classes should not know anything about their derived classes*" (Arthur J Riel, 1996) as this would force an abstraction to become dependent upon a concrete implementation (detail) canceling out the benefit of inheritance. In the context of the proposed model this rule (Rule 2) implies that new edges leaving abstract classes and reaching their subclasses should not be allowed.

The aforementioned two rules, as applied in the proposed model, are depicted graphically in Figure 19. The model rejects edges that correspond to the marked cases. Practically, during the construction of networks for future software versions it is being checked whether the following domain rules are satisfied. Whenever an edge that violates these rules is created, it is being rejected and another source or destination node is being sought.

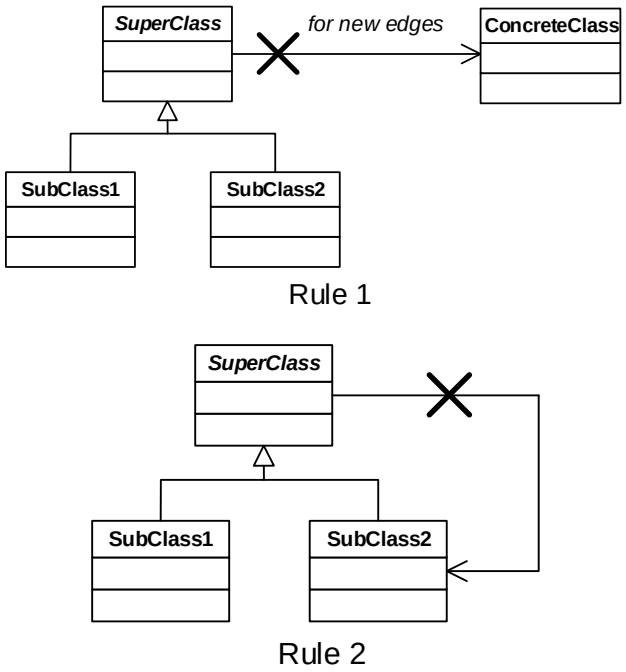


Figure 19 - Graphical depiction of applied domain rules

The validity of these particular domain rules can be tested by examining the occurrences in which the rules have been violated throughout the history of the examined projects. We recorded separately the number of times that a superclass created an edge to another system class and the number of times when a superclass created an edge to one of its subclasses. In particular, for the first rule, we calculated the percentage of edges throughout the history of each project that emanate from super classes and end up in another system class (not subclasses). For the second rule, we calculated the percentage of edges that link a superclass to one of its subclasses. The results are displayed in Table 7.

Table 7 - Percentage of actual rule violations in the history of examined projects

#	Name	Rule 1 violations	Rule 2 violations
1	aTunes	10%	0%
2	FreeCol	6%	0%
3	JDeodorant	5%	0%
4	JEdit	8%	0%
5	JFreeChart	10%	0%
6	Jmol	15%	1%
7	Weka	6%	0%
8	HFS Explorer	20%	2%
9	Presto	8%	0.1%
10	Jetty	10%	0.1%

As it can be observed, although the violation of Rule 1 is not common, the percentage is not negligible. As stated in Section 4.1.2 , the proposed prediction model allows, via a model parameter, for a certain degree of randomness in every decision that has to be made. Consequently, a certain percentage of rule violations can be permitted. By examining these cases in detail, we have found out that such new edges (forming new links between an abstract superclass and another system class) is always the consequence of enhancing the public interface of the superclass. As an example, the introduction of a new method signature, containing a parameter of a system class type, in the list of methods of an interface, leads to an admissible violation of rule 1. On the contrary there are almost no violations of rule 2 as it would be a major design flaw to make a superclass dependent upon its descendants. The effect of the consideration of these rules on the accuracy of the proposed model will be demonstrated in Section 4.4.1 where the results are discussed.

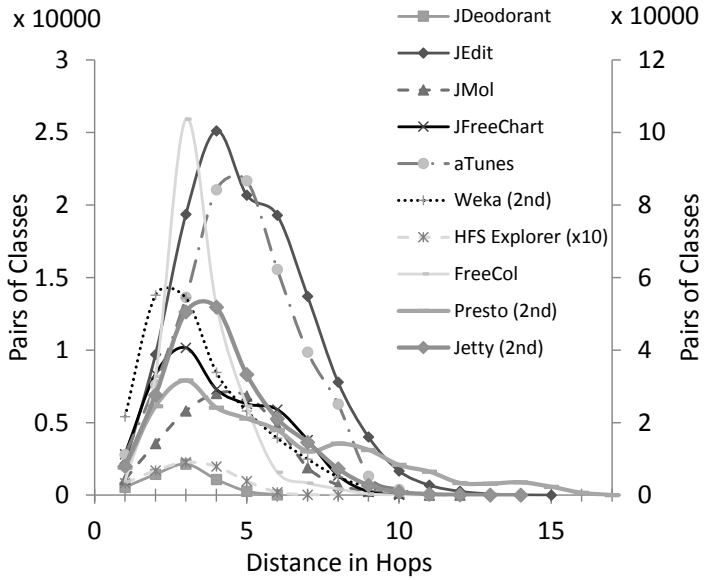
4.2.6 Small world phenomena

Social psychologist Stanley Milgram conducted in the 1960s a famous experiment which suggested that the entire human society is characterized by relatively short paths among its members. Popularly known as six degrees of separation, this experiment concluded that any two people on this planet can be linked via an average number of six intermediate acquaintances (Easley and Kleinberg, 2010; Kleinberg, 2000). A similar property has been observed in several technological and social networks (Watts and Strogatz, 1998). A network is characterized by the *small-world phenomenon* if any two nodes have a high probability of being associated through a short path of intermediate nodes (Easley and Kleinberg, 2010).

The presence of the small-world phenomenon implies that the diameter of the corresponding graph is relatively small. According to Easley and Kleinberg (2010) the small-world phenomenon can be attributed to the existence of homophily (the tendency of nodes to connect to similar nodes) and the presence of weak ties (edges that link distant nodes). In object-oriented systems both of these properties are reasonable to expect. Up to a certain degree, classes tend to form relationships with other classes that have conceptual similarities (such as classes residing in the same package) and at the same time classes often create associations to distant classes residing in other packages. Therefore, we should

expect that the majority of classes in networks representing object-oriented systems are connected by paths of a relatively small length.

To investigate the presence of the small world phenomenon we employ hop plots, depicting graphically the number of class pairs which are *h* hops apart. Figure 20 illustrates the hop plot for the last version of all examined systems. The most striking observation is the spike around 3 and 4 hops, indicating that in almost all systems, the majority of class pairs have a distance of 3 or 4 hops. Moreover, even for the systems with a larger diameter, very few classes are more than 9 hops apart. This initial evidence implies the presence of a sort of small-world phenomenon in the networks of classes. However, its intensity varies from one project to the other. It should be noted that if the main goal is to test whether a network exhibits small-world phenomena, a more systematic approach could be employed, such as the comparison of topological parameters (e.g. clustering coefficient) to a null model (Hosseini and Kesler, 2013).



*Projects Weka, Presto and Jetty are displayed against secondary y- axis

Figure 20 - Hop Plots for the last version of all examined systems
(curves are smooth for aesthetic reasons: distance can assume only integer values)

In order to investigate whether the decisions that have been incorporated in the proposed model (such as the preferential attachment and duplication models, the sampling of past distributions and the introduction of domain knowledge) preserve the validity of the small-world phenomenon, we consider it important to assess the accuracy of the proposed model employing the aforementioned hop-plots. We believe that these diagrams

reflect in a representative manner the structure of the underlying graph and as a result can reveal whether the forecasting power of the proposed model is high or low. Therefore, we have used hop plots that compare the predicted network structure to that of the actual target network in the evaluation section (Section 4.4.1).

4.3 Overview of the Software Evolution Model

Since the proposed model encompasses a number of phases and parameters, a graphical overview is provided in Figure 21. First, the project versions under investigation are parsed and mapped to their corresponding network representation (step 1) and these representations are analyzed to extract the aforementioned past distributions (step 2). Next, a sequence of steps is repeated for all versions that have to be simulated, in order to perform changes on the virtually evolved network. All actions that have to be performed are depicted in the shown steps along with the parameters that have to be considered.

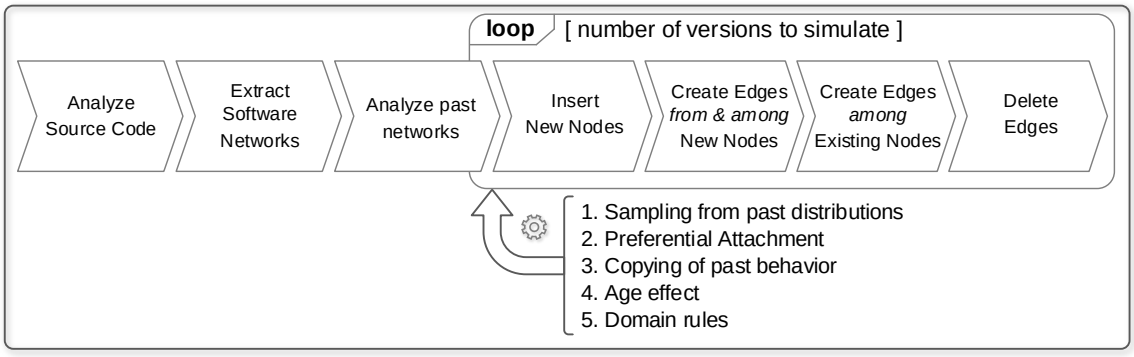


Figure 21 - Overview of the proposed model

All aspects of the proposed model have been implemented as an Eclipse plugin. The plugin can be downloaded from a dedicated webpage (2014) that also provides access to the network data for all projects examined in this paper.

4.4 Evaluation

Since the proposed model aims at predicting future trends in software evolution, in terms of its network representation, evaluation consists in determining its forecasting power. In order to check the efficiency of the proposed model we employ the approach that has been used for similar purposes in the domain of social networks. A set of past version data (in our case 50-60% of the available software versions for each project) is employed as training dataset in order to obtain the distributions of network properties that dictate the model parameters.

The rest of the versions are used as test dataset and eventually we perform the evaluation along two axes:

Network Perspective: we test network properties de-rived from the application of the proposed model, against the values of the corresponding properties for the final available software version, and

Software Perspective: in order to demonstrate the potential for practical exploitation of such a prediction model, we investigate the forecasting power when predicting the evolution of class fan-in as well as the afferent and efferent package coupling.

4.4.1 Evaluation from a network perspective

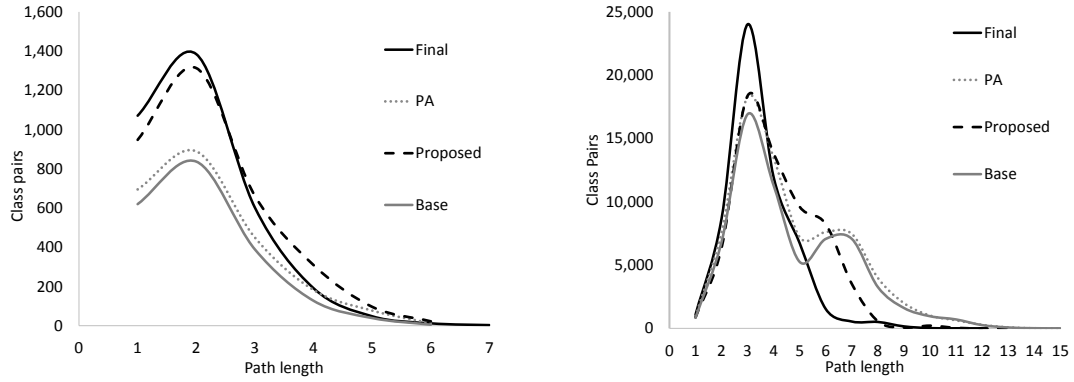
The properties that are put under investigation provide a representative picture of the corresponding network topology (Bhattacharya et al., 2012; Leskovec et al., 2008, 2005; Pfeiffer III et al., 2012) and thus can serve as indicators of how much the predicted network resembles the graph corresponding to the actual software system of the final version. These properties are:

a) the distribution of the distance between all class pairs (captured by the number of class pairs versus the distance in number of hops). For this analysis we employ the aforementioned hop plots.
b) the in-degree distribution of the corresponding graph expressed by the exponent (α) of the power function.
c) the diameter of the corresponding networks (i.e. the longest shortest path between any two nodes, where shortest paths are calculated according to Dijkstra's algorithm (1959)).
d) size properties of the corresponding networks (number of nodes, edges and the resulting density) as representative for the accuracy of the proposed model to estimate the growth of the simulated software system.

To provide insight into the benefit of enhancing the simple Preferential Attachment model with the additional parameters and rules shown earlier, the results are given for the last version of the training set (***Base***), for the network derived when applying the Preferential Attachment model (***PA***), for the network derived when applying all parameters

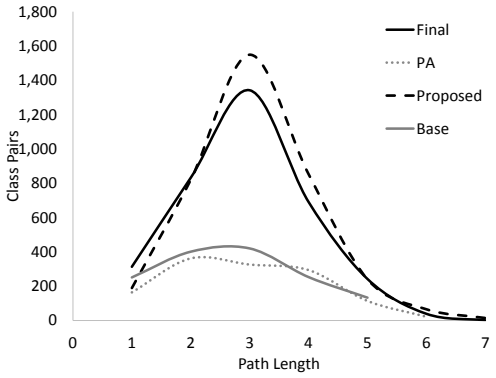
and rules of the proposed model (***Proposed***) and for the actual network corresponding to the last version of the test dataset (***Final***).

The results are shown in Fig. 11 for the distribution of the distance in terms of hop plots and in Table 6 for the numerical values of the power exponent (α), the diameter and the size properties, for all examined software projects. Obviously, the goal of the simulation is to reproduce the final version as accurately as possible.

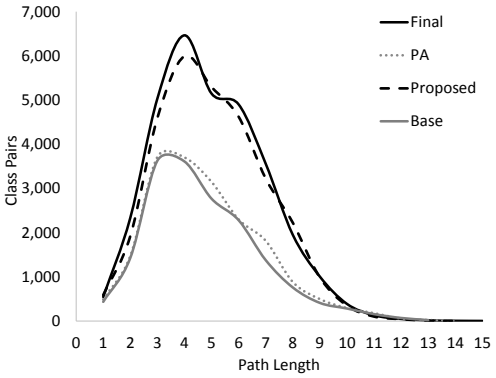


aTunes

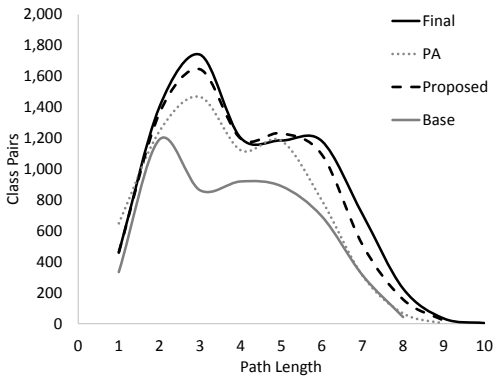
FreeCol



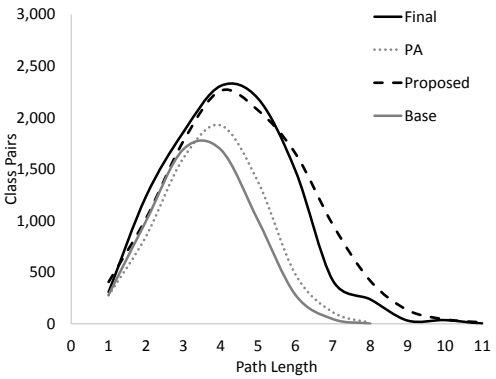
JDeodorant



JEdit



JFreeChart



Jmol

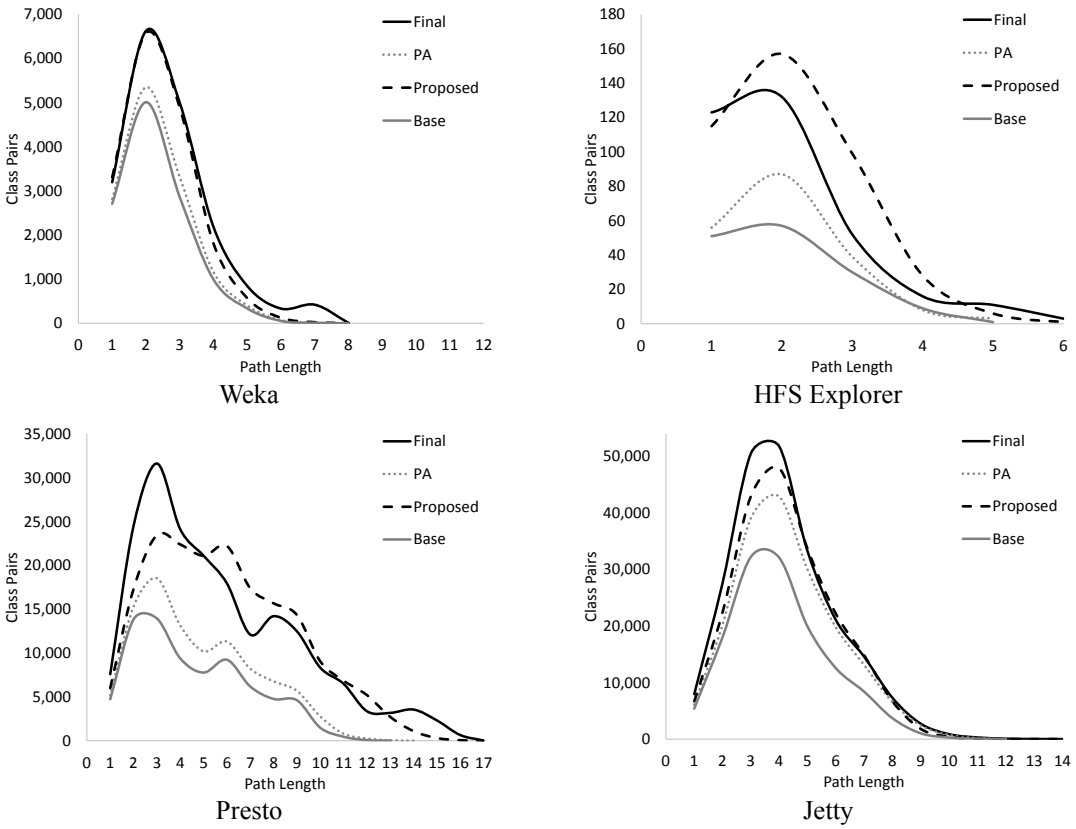


Figure 22 - Comparison of predicted and actual hop plots
(curves are smooth for aesthetic reasons: path length can assume only integer values)

As it can be observed from the hop plots, in almost all cases the proposed model increases significantly the forecasting power over the simple Preferential Attachment model and bridges the largest part of the distance between the Base and the Final version. From Figure 22 it becomes clear that the distribution of the distance between class pairs predicted by the proposed model, approaches the distribution for the final version more efficiently than the simple preferential attachment model, for the entire spectrum of distances.

Table 8 - Comparison of predicted and actual network properties

		δ	**|V|**	**|E|**	**Density**	**α**
	Base	6	438	1004	0.0052	-1.283
	PA	7	473	1237	0.005541	-1.244
aTunes	% error	-22%	-43%	-37%	94%	12%
	Proposed	11	702	1719	0.003493	-1.276
	% error	22%	-16%	-13%	22%	15%
	Final	9	832	1979	0.00286	-1.114
	Base	15	548	2362	0.00788	-1.11645
FreeCol	PA	15	631	3336	0.008392	-1.17921
	% error	25%	-18%	-16%	26%	15%

FreeCol	Proposed	13	820	4313	0.006422	-1.15214
	% error	8%	6%	9%	-4%	13%
	Final	12	772	3975	0.006678	-1.0218
JDeodorant	Base	5	84	264	0.037866	-1.1589
	PA	6	141	472	0.023911	-1.1329
	% error	-14%	-38%	-39%	60%	-22%
	Proposed	7	238	710	0.012587	-1.3285
	% error	0%	4%	-9%	-16%	-8%
	Final	7	229	779	0.01492	-1.44615
Jedit	Base	13	662	1499	0.003426	-1.38723
	PA	14	693	1638	0.003416	-1.38888
	% error	-7%	-28%	-28%	38%	0%
	Proposed	14	1042	2290	0.002111	-1.385
	% error	-7%	9%	1%	-14%	0%
	Final	15	960	2273	0.002469	-1.38898
JFreeChart	Base	8	532	1733	0.006135	-1.19137
	PA	9	584	2111	0.0062	-1.13429
	% error	-10%	-43%	-27%	122%	-9%
	Proposed	9	776	2058	0.003422	-1.18378
	% error	-10%	-24%	-29%	23%	-5%
	Final	10	1018	2892	0.002793	-1.24448
Jmol	Base	8	409	845	0.005064	-1.33827
	PA	8	461	971	0.004579	-1.3152
	% error	-33%	-16%	-14%	21%	-8%
	Proposed	13	573	1184	0.004635	-1.4512
	% error	8%	5%	5%	22%	1%
	Final	12	547	1130	0.003784	-1.43146
Weka	Base	8	864	3293	0.004416	-1.03944
	PA	9	1296	5198	0.003097	-1.0032
	% error	-18%	-16%	-12%	26%	-5%
	Proposed	12	1704	6010	0.002071	-1.0902
	% error	9%	10%	2%	-16%	3%
	Final	11	1550	5884	0.002451	-1.05372
HFS Explorer	Base	5	109	213	0.018094	-1.71211
	PA	5	114	232	0.01801	-1.73183
	% error	-29%	-72%	-81%	150%	28%
	Proposed	8	382	1183	0.006493	-1.28257
	% error	14%	-8%	-4%	-10%	-5%
	Final	7	414	1233	0.007211	-1.3511
Presto	Base	13	1380	4745	0.00249	-1.19893
	PA	14	1812	5177	0.00154	-1.23873
	% error	-18%	-18%	-32%	-16%	-2%
	Proposed	17	1812	6011	0.001578	-1.31144
	% error	0%	-18%	-21%	-14%	4%
	Final	17	2219	7595	0.00183	-1.25875

Jetty	Base	12	2188	5396	0.001128	-0.29392
	PA	12	2902	6110	0.000726	-1.35826
	% error	-14%	-9%	-23%	-6%	4%
	Proposed	12	2902	6754	0.0008	-1.33193
	% error	-14%	-9%	-15%	3%	2%
	Final	14	3201	7952	0.000776	-1.30299

A comparison of the proposed model and the Preferential Attachment model on the basis of the other network properties is shown in Table 8. To enable an evaluation of the relative error, as well as the improvement over the last examined version which served as the baseline for the prediction, the network properties are also shown for the corresponding graphs (Final, Base).

For each case, the graph diameter δ, the number of nodes $|V|$, the number of edges $|E|$, the graph density and the slope α of the degree distribution, are predicted with higher accuracy by the proposed model (the obtained value is closer to that of the final version) than by the simple Preferential Attachment model.

To investigate whether the difference between the proposed and the Preferential Attachment model is statistically significant, we compared their ability to capture the hop plot of the final system. To this end, we compared the errors resulting from each approach by employing a paired samples t-test and a related samples Wilcoxon Test depending on whether the differences of errors are normally distributed or not (Lehmann and Romano, 2005; McDonald, 2009). The paired samples t-test is ideal for the comparison of a pair of variables distinguished by a Boolean characteristic (e.g. Before, After) (Lehmann and Romano, 2005; McDonald, 2009) . However, this test can be applied only if the differences of the pairs are normally distributed. If they are not, the most appropriate test is the Wilcoxon. Consequently, at first we ran a Kolmogorov Smirnov (K-S) test for normality in the differences of the errors and then, depending on the K-S test result we applied either the paired samples t-test or the Wilcoxon test.

The corresponding null hypothesis of the t-test (Wilcoxon test) is that the difference of means (medians) between paired observations is equal to zero, i.e. H_0: $\mu_1 - \mu_2 = 0$. The results are shown in **Table 9**. As it can be observed for seven out of the ten projects the proposed model is superior to the PA model since the resulting error is lower and the difference is statistically significant. For three projects where the errors are relatively similar, the results are not statistically significant.

Table 9 - Statistical comparison of errors between the proposed and the PA model

Project Name	Error (PA)	Error (Proposed)	Sig. (2-tailed)	Method
aTunes	0.22	0.02	0.031*	t-test
FreeCol	0.18	0.12	0.007*	t-test
JDeodorant	0.005	0.005	0.345	Wilcoxon
JEdit	0.08	0.02	0.05*	t-test
JFreeChart	0.11	0.03	0.012*	Wilcoxon
jMol	0.15	0.07	0.033*	t-test
Weka	0.17	0.02	0.02*	t-test
HFS	0.024	0.016	0.893	Wilcoxon
Presto	0.1385	0.0837	0.007*	t-test
Jetty	0.0337	0.0302	0.216	t-test

* implies that the result is statistically significant at the 0.05 level

A large portion of the improvement over the PA model is due to the consideration of new edges between existing nodes, and new edges between new nodes. Edges of this kind are quite frequent during the evolution of several systems and thus should not be neglected. Moreover, it appears that the consideration of edge removals, in combination with the introduction of edges between existing nodes, helps in modeling accurately the diameter of the predicted network (compared to the base version). It should be mentioned that the percentage of simulated node deletions caused by the removal of edges leading to unconnected nodes is between 0.5% and 1.5%.

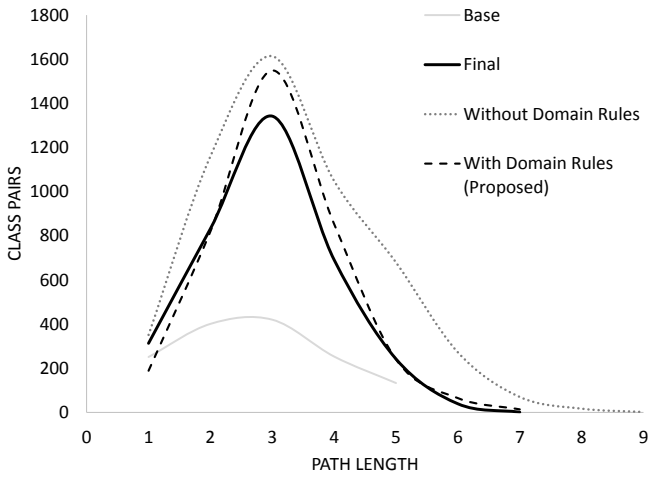


Figure 23 - Impact of domain knowledge application.

We have also observed the impact of the application of the domain rules, which in terms of distance distribution helps in following the actual distribution more accurately. In Figure 23 the impact of considering domain rules on the hop distance among class pairs is

shown for project JDeodorant. It becomes evident that when the domain rules are taken into account, the corresponding curve moves closer to that of the final version and moreover the diameter of the final graph is correctly predicted. The simulation revealed that the domain rules regarding the handling of edges emanating from superclasses have been applied many times, which means that otherwise, edges leaving superclass and edges reaching subclasses of the same hierarchy, would have been added throughout the simulated network of classes.

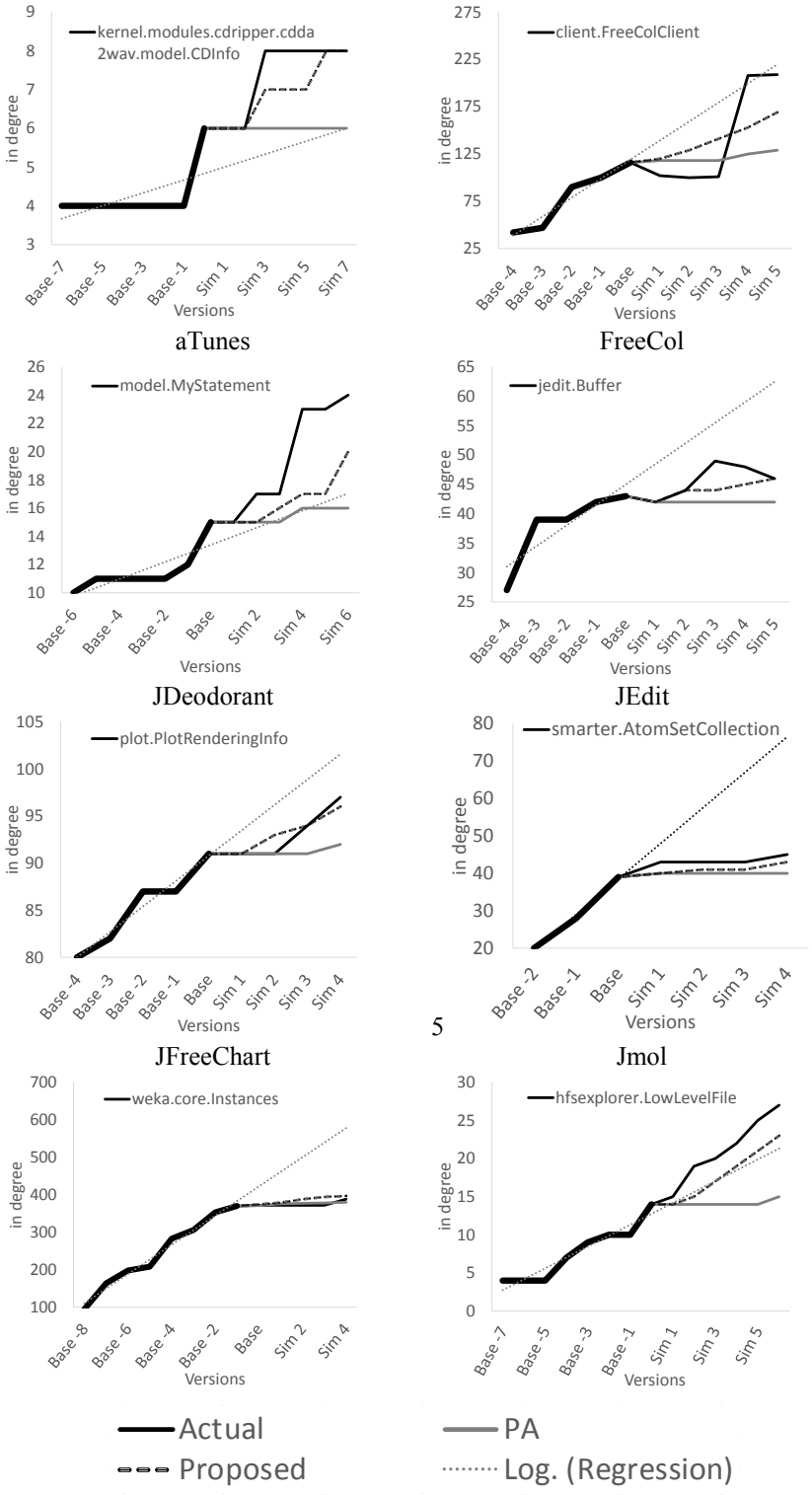
4.4.2 Evaluation from a software perspective

Software prediction models based on the network representation would be valuable if their results could be exploited by software developers or maintainers. As an example, the prediction of future software evolution can indicate which system classes might become overloaded, based on their expected in-degree (fan-in). This information might be useful for anticipating possible future "God" or highly-coupled classes.

To investigate how accurately the proposed model can predict the classes with the largest in-degree in a future version, we compare the predictions against the actual evolution. Moreover, we compare the predicted afferent and efferent coupling at the package level to the actual values of the last examined version.

In Figure 24 we illustrate for all projects, the evolution of the in-degree for the class that exhibited the largest variation between the first and last examined version (the same classes as in Figure 13). Each figure shows the actual evolution of the in-degree (continuous line, where the thicker part corresponds to the training period), the evolution predicted by the proposed model (dashed line), the PA model (grey line) and the regression-based model (dotted line). The three models employed for training all versions up to the version denoted as Base.

As it can be observed, despite the numerous factors that affect the growth of a software system, the variation of the in-degree is captured to a large extent by the proposed model offering significant improvement over the PA and the regression models. The improved performance of the proposed graph-based prediction model, justifies the use of graphs as means for the study of evolution, since it becomes evident that simple regression is inadequate. Moreover, the improved results compared to the ones of the preferential attachment model are due to the consideration of the various parameters in our model.


kernel.modules.cdripper.cdda 2wav.model.CDInfo
in degree
Versions
Base -7
Base -5
Base -3
Base -1
Sim 1
Sim 3
Sim 5
Sim 7
aTunes
client.FreeColClient
Base -4
Base -3
Base -2
Base -1
Base
Sim 1
Sim 2
Sim 3
Sim 4
Sim 5
FreeCol
model.MyStatement
Base -6
Base -4
Base -2
Base
Sim 2
Sim 4
Sim 6
JDeodorant
jedit.Buffer
JEdit
plot.PlotRenderingInfo
Base -4
Base -3
Base -2
Base -1
Base
Sim 1
Sim 2
Sim 3
Sim 4
JFreeChart
smarter.AtomSetCollection
Base -2
Base -1
Base
Sim 1
Sim 2
Sim 3
Sim 4
Jmol
weka.core.Instances
Base -8
Base -6
Base -4
Base -2
Base
Sim 2
Sim 4
hfsexplorer.LowLevelFile
Base -7
Base -5
Base -3
Base -1
Sim 1
Sim 3
Sim 5
Actual
Proposed
PA
Log. (Regression)

5

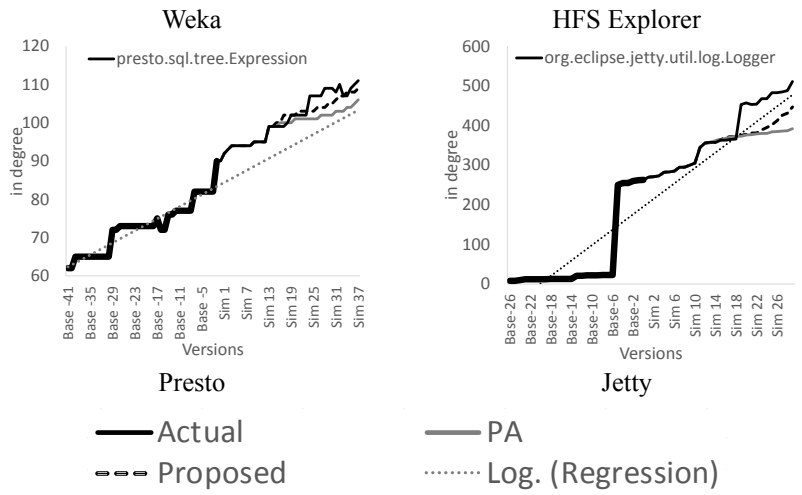


Figure 24 - Comparison of actual and predicted evolution of in-degree.

For the shown classes the average error in the prediction of the actual in-degree is 7.3% for the proposed model, while for the PA and regression models the error is 19.1% and 24%, respectively. Certainly, a prediction model cannot guarantee to reveal all forthcoming changes in software but providing even a relatively accurate prediction might be the basis for further work in this direction.

As already mentioned, in contrast to all existing models, the proposed prediction model considers also edge removals. Such edge removals, although less frequent than edge additions, might result in decreases of the in- and out-degree of particular nodes. We have measured the percentage of classes whose in- and out-degree dropped between the first and last examined version. The percentages vary from 0% to 2.7% for in-degree reductions and from 0.6% to 12% for out-degree reductions. To assess the ability of the proposed model in predicting cases where the degree of a node decreases, Table 10 shows the number of classes whose in- and out-degree actually decreased during the evolution of the examined systems as well as the classes for which the proposed model successfully predicted a degree reduction.

A network-based software prediction model can also be useful in anticipating the evolution of more 'traditional' software properties, such as the afferent and efferent coupling at the package level. In our context, the afferent coupling for a given package refers to the number of other packages that depend upon classes within this package and can be considered as an indicator of the package's responsibility.

Table 10 - Prediction of classes whose in- and out-degree Decreased*

Project Name	Classes whose in-degree decreased		Classes whose out-degree decreased	
	Actual	Predicted	Actual	Predicted
aTunes	13	5	31	28
FreeCol	21	6	11	3
JDeodorant	5	5	5	2
JEdit	17	9	26	16
JFreeChart	8	7	36	36
jMol	8	1	9	1
Weka	1	1	1	1
HFS	0	0	0	0
Presto	52	37	33	28
Jetty	41	29	19	16

* Matching has been performed at the level of identical class names

Efferent coupling refers to the number of other packages that the classes in a given package depend upon and can be considered as an indicator of the package's independence (Martin, 2003). In order to evaluate the accuracy of the proposed model we investigated the predicted afferent and efferent package coupling. Figure 25 depicts graphically the afferent and efferent coupling for a) the last version of the training set (Base), b) the last version of the test dataset (Final) and c) the results predicted by the proposed model (Proposed), for all examined systems. The set of bars on the left-hand side correspond to the afferent coupling and values are measured against the bottom x-axis. The set of bars on the right-hand side correspond to the efferent coupling and values are measured against the top x-axis. The packages correspond to the top ten packages with the highest afferent coupling in the Final version (in case the system contains less than 10 packages, all are shown). In cases where only one bar is shown (representing the Final version), the corresponding package had a zero coupling value in the base version.

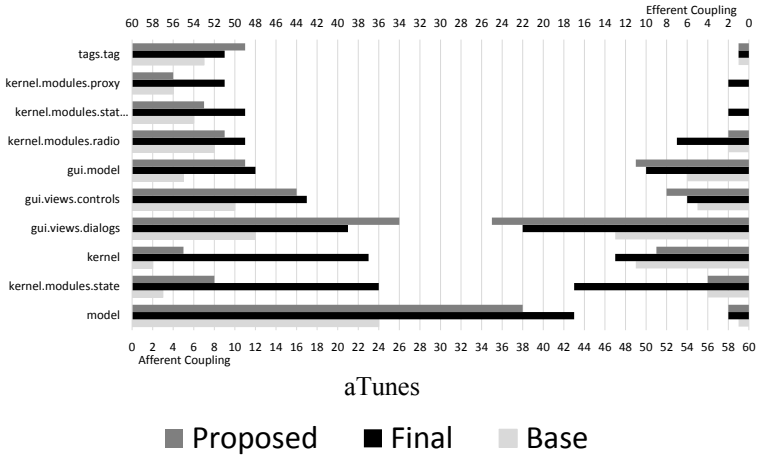
In general, it can be observed that in a large number of cases where the coupling value of the final version is larger than that of the base version, the proposed model correctly identified this increasing trend and predicted a value that is larger than that of the initial state. In particular, changes in the afferent coupling have been correctly predicted in 81% of the cases, while changes in the efferent coupling have been correctly predicted in 60% of the cases.

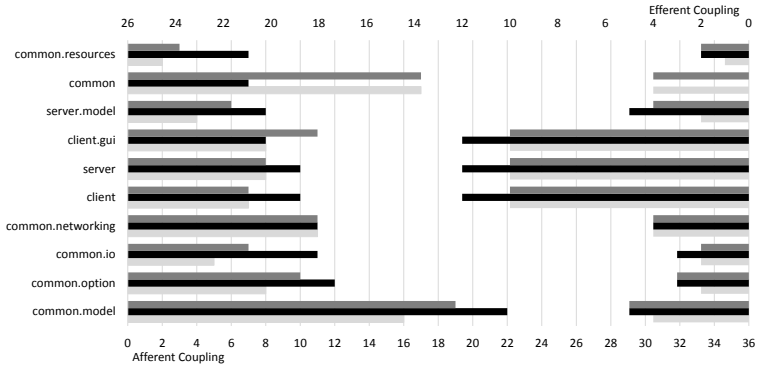
However, as it can be observed, there are several cases where the accuracy of the prediction is limited. This is primarily the case when the set of historical observations is

not sufficient. As an example, consider the case of package `modules.radio` in project aTunes (fourth set of bars on the right of the first figure). The predicted efferent coupling of value 2 is equal to the efferent coupling that the project had at the base version (end of training period) while the actual coupling at the end of the forecasting period has risen up to 7. The reason is that this package was introduced into the system in the last version of the training period thus offering no historical data to guide the prediction. As already mentioned, the quality of a forecast depends heavily on the representativeness of the recorded history.

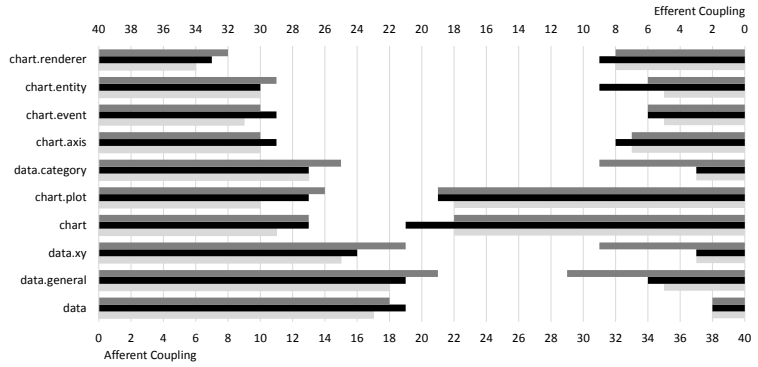
The aforementioned results indicate that a network prediction model with sufficient forecasting power can be valuable in guiding the maintenance efforts of a software project. Anticipating which classes exhibit a tendency to become large in terms of functionality offered to clients or extremely dependent upon other classes, might be helpful to identify system modules that warrant further attention and possibly refactoring.

Similarly, it would be valuable to know in advance which packages might develop excessive coupling among their modules or to other packages and which parts of the architecture appear to be volatile. In any case, representing software systems in terms of networks enables a systematic modeling of their evolution and can provide a glimpse into their future. Furthermore, the output of this kind of analysis can be fed to empirical studies that would investigate the relation among the examined network properties and design qualities of the underlying systems in order to validate the use of graph metrics as indicators of software quality.

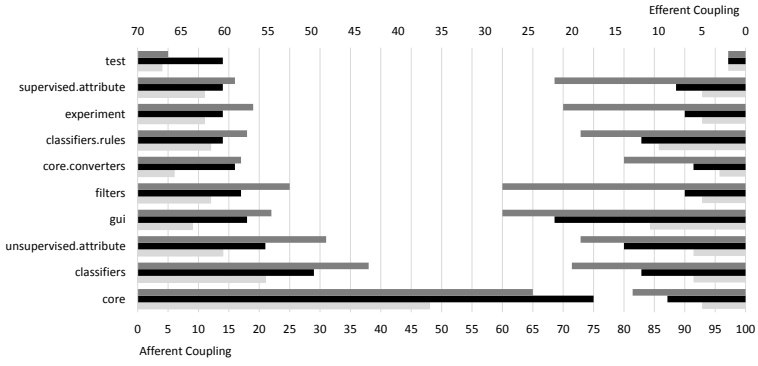




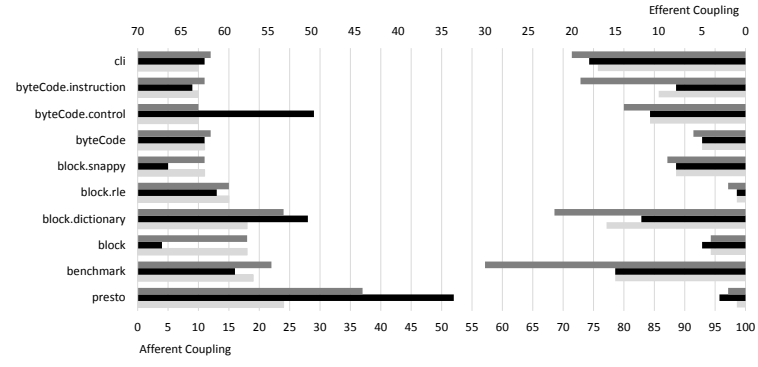
FreeCol



JFreeChart

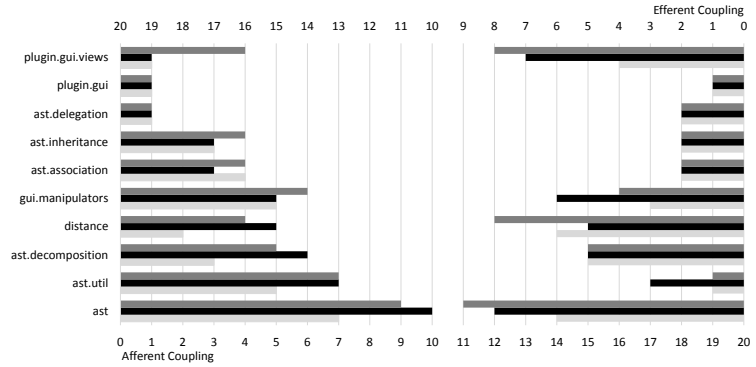


Weka

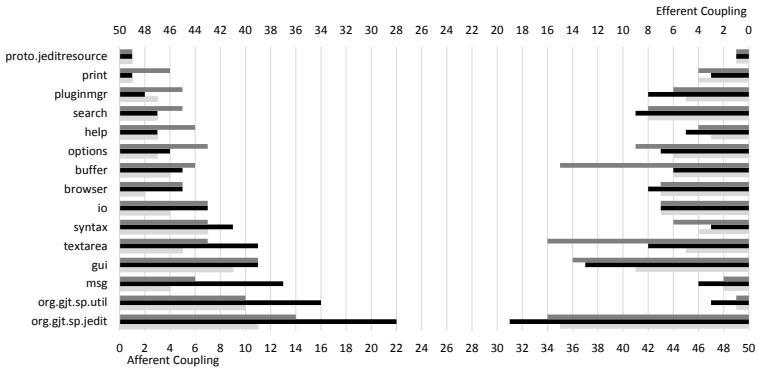


Presto

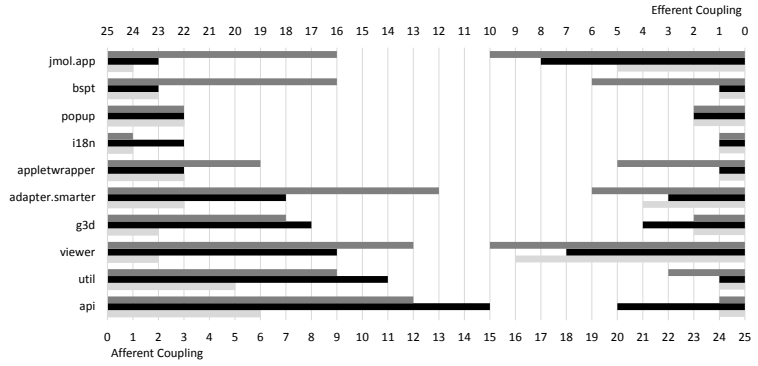
Proposed Final Base



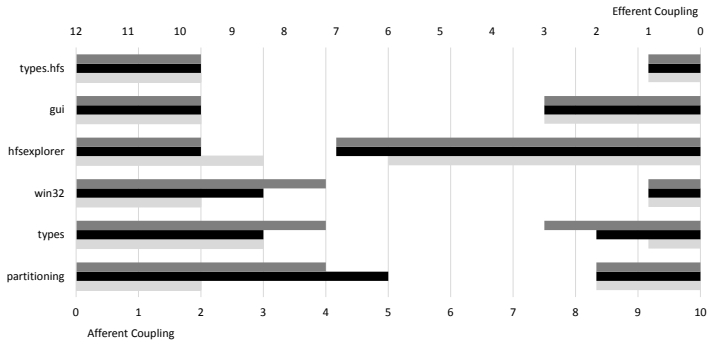
JDeodorant



jEdit



Jmol



HFS Explorer

Proposed Final Base

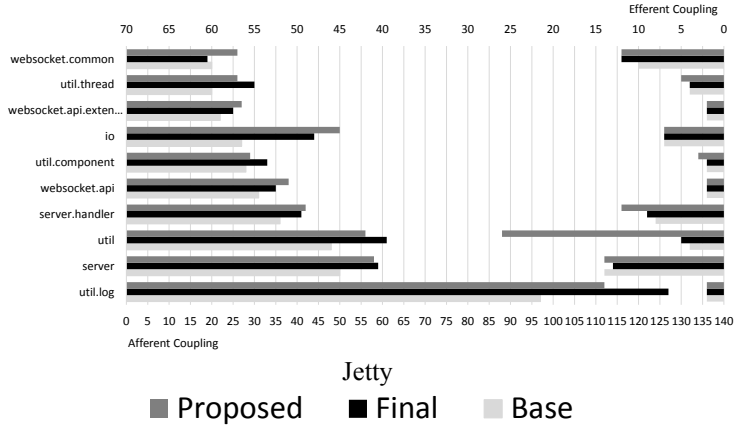


Figure 25 - Comparison of predicted Afferent and Efferent Package Couplings.
Afferent Coupling is illustrated on the left bars (bottom x-axis). Efferent Coupling is illustrated on the right bars (top x-axis)

The accuracy of any prediction model is by definition constrained, especially in the software domain, since future evolution can be affected by numerous, unpredictable factors such as requirements for implementing radically novel functionality, decisions to modify the architecture, changes in the development team etc. The limitations and threats to validity that can be identified are outlined next.

The proposed approach samples from distributions based on past version data to obtain several model parameters. This sampling assumes that the system under study will continue to evolve in the future in a similar manner as in the past. This means that if abrupt changes to the network topology occur due to major architectural modifications for example, the predicted future network evolution might be inaccurate. As another example, if the trend in the number of nodes and edges has been increasing during the entire history of a software project with an exception for the last version (where it could possibly drop), the model would not be able to predict this sudden reduction in the number of nodes and edges. Another source of inaccuracy is related to the fact that in order to derive the model parameters, the distributions are captured by means of curve fitting to the actual data. The corresponding mathematical function might not always have an ideal fit to the actual data points, affecting the accuracy of the extracted parameter.

Apart from the aforementioned limitations, the proposed prediction model suffers from the usual threats to external and internal validity. The fact that the model is evaluated against ten projects, unavoidably limits the possibility to extensively generalize our findings. This threat is related to the observation of general trends which are applicable to

all projects (such as preferential attachment). It is always possible that another set of projects might exhibit different phenomena. A similar threat stems from the fact that all analyzed projects are developed in Java thus limiting the ability to generalize to other object-oriented languages. However, since a large part of the model consists in learning from past versions, we believe that it can be successfully adapted to any software project regardless of its particularities.

Concerning the internal validity (i.e. the parameters that might affect the evolutionary trends that we are trying to predict), it is reasonable to assume that numerous other factors, which affect software growth, might have not been taken into consideration. As an example, a forecasting model could be augmented by considering the co-evolving developer community or dependencies among documented requirements. However, the proposed simulation approach can be considered as incremental in the sense that additional parameters can be easily integrated. The same holds for the incorporation of additional domain rules which might be representative for a particular application domain or the habits of a particular development team.

5 SEAgle: A platform for Software Evolution Analysis

Statistics are like bikinis. What they reveal is suggestive, but what they conceal is vital.

Aaron Levenstein, prof. of business

5.1 Tools for software engineering research

Calculation of metrics that quantify quality characteristics is a process of fundamental importance for every software engineering researcher. There is a number of current tools that offer calculation of various quality metrics and provide interesting reports about the overall characteristics of a software system. But each of these tools suffers from one or more of the following disadvantages:

- difficult and time consuming installation procedure
- static (not evolutionary) analysis of the project
- the need for code-under-analysis to exist in the same storage system with the tool and also to be compilable
- simple presentation of results without the ability compute relations among variables

An ideal tool for the calculation of a number of metrics, in our point of view, should be provided to the community as a platform with a number of characteristics:

+ the platform should be easy to use. To this end we opted for a Web based platform enabling users to analyze a repository by a single click (either selection of an already analyzed project or by providing the git repository URI).
+ software repositories encompass a project's history. As a result, all reported information spans across all available versions, i.e. constitutes a form of software evolution analysis.
+ Any software system has several facets. Therefore, we offer multiple views concerning commit-related metrics, source code metrics and graph based metrics.

+ Empirical studies very often focus on the investigation of relations among variables. To satisfy this need we offer direct correlation analysis between any two monitored variables. (for this reason the x-axis is common on all diagrams and represents software versions)
+ Contemporary software repositories are extremely large in size. To confront this challenge, we optimized the process of extracting commit-related metrics, which are demanding since they involved the analysis of thousands of commits.

5.2 Platform architecture

The architecture of SEAgle is outlined in Figure 26. In the left hand side components offering core services are shown, such as the API taking care of communication with VCS and the API responsible for metrics. For the latter two components the architecture is highly extendible in the sense that a clear separation between abstraction and implementation has been adopted. The Software Evolution Analysis Engine, running in Java EE, exploits services provided by individual components and stores the calculated results in a MySql database. Moreover, the engine provides Web Services (REST), which are accessed by the presentation tier in order to trigger the analyses and retrieve the results which are then displayed in the form of charts and tables.

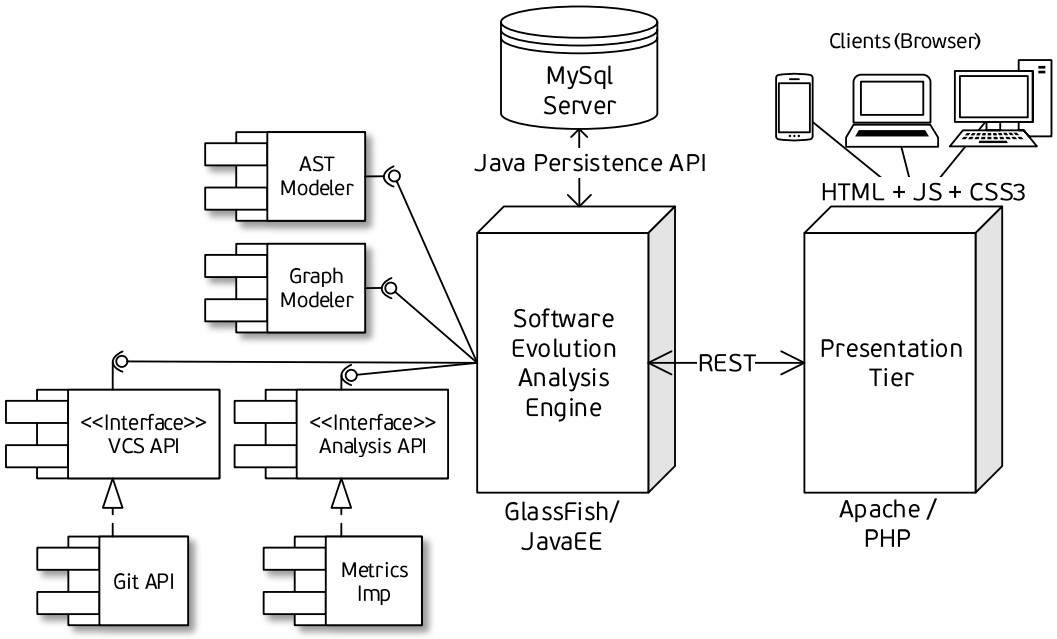


Figure 26 - Architecture of SEAgle platform

The proposed platform employs a plethora of state-of-the-art technologies, which are outlined in Table 11.

It should be mentioned that in order to minimize human intervention versions are automatically determined based on tags explicitly contained within the git repository. This is in alignment with common practices in software development where tags delineate different software releases.

Table 11 - Technologies employed in SEAgle

Core Components	
Java EE 7 / Glassfish	• provides an API and runtime environment to run on a Web Server. We selected the "Payara" distribution of Glassfish to be our application server.
JAX-RS API	• provides support in creating RESTfull Web Services, we used "Jersey", the annotation-based, reference implementation of this API.
Java Persistence API (JPA)	• facilitates object-relational mapping and storage of analysis results to the database. We used the Eclipse Link as an implementation of JPA
R	• for the calculation of statistical measures
JGit	• Library used to access git source code management (SCM) systems
Web sockets	• interaction with presentation tier to provide progress monitoring
Guava	• Extended Java Collections caching
Jung	• Library for Graph Modelling with implementations of Graph based algorithms
Eclise JDT	• Eclipse Java Development Toolkit for AST parsing and object bindings manipulation
Presentation Tier	
HTML5, CSS3	• Local storage, adaptive screen controls, etc.
Bootstrap	• Responsive design
JavaScript	• REST client implementation and client-side light data manipulation
Flot, JQuery, Sparkline	• Chart creation technologies • Data manipulation

5.2.1 Analysis Engine

The overview of the analysis process is depicted in Figure 27. The analysis engine is comprised of the repository handling and downloading module, the version module, the source code parsing module and the module that handles and dispatches requests for metric calculations, the Metrics Manager.

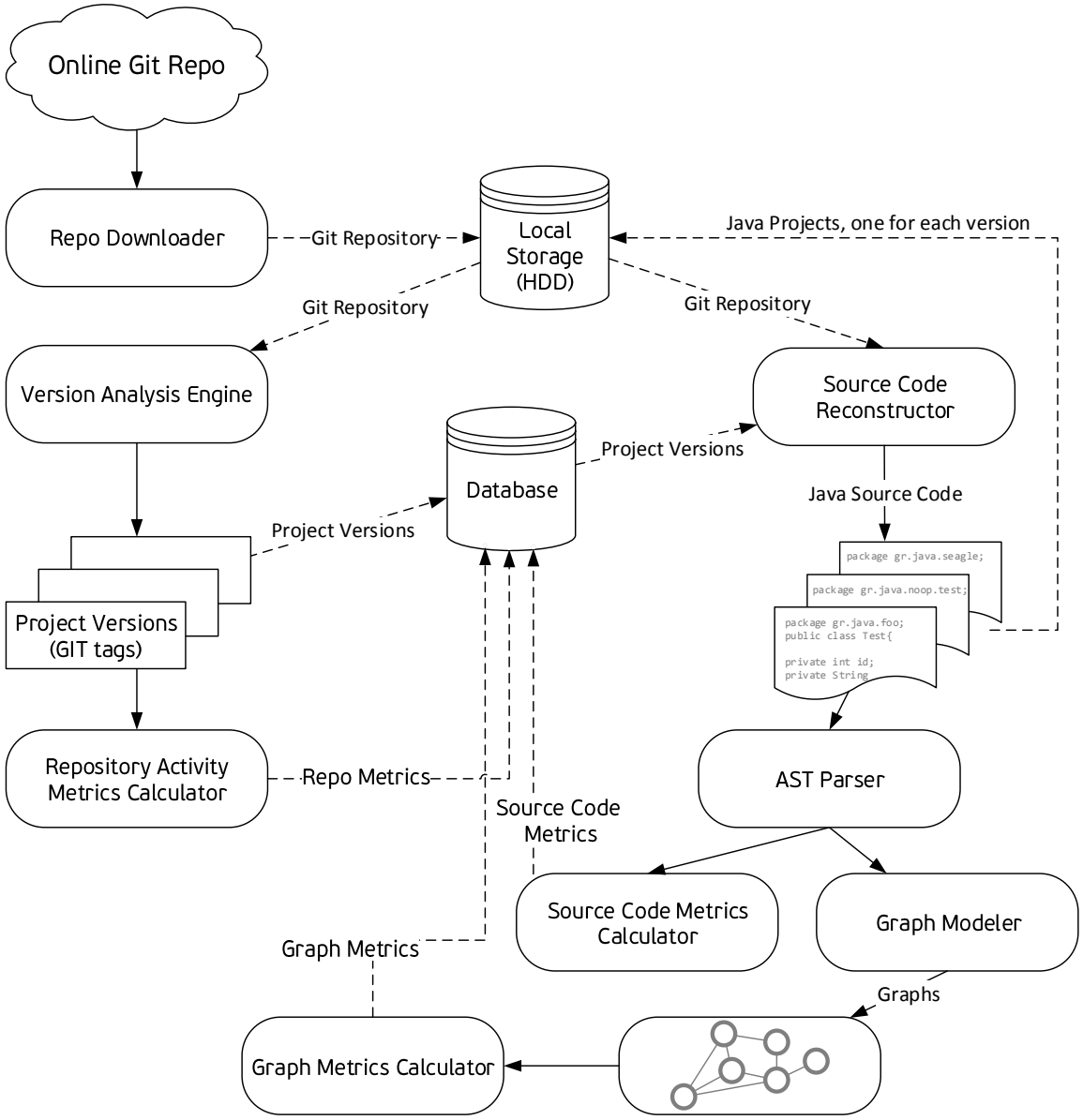


Figure 27 - Overview of analysis process

The analysis process is triggered by the user who enters the URI from a publicly-available git repository as displayed in Figure 29. A necessary precondition is that the repository under analysis should contain at least one git tag, because tags are handled as versions by our analysis engine. The module that handles the repository downloading, clones the given repository and stores it locally in a specific folder in the Hard Disk physical storage.

After the successful cloning, the analysis engine determines the versions of the system and orders them by using a process that was developed specifically for this purpose and is analyzed in the Appendix.

After the determination of the version ordering, the Metrics Manager dispatches the analysis requests to each one of the registered metric calculators. Currently, the implemented metric calculation subsystems are three. The repository metrics calculators, the source code metric calculators and the graph-based metrics calculators.

Special attention should be paid on the graph metrics calculation system since it is the most complex one. The process of modeling each software system as a graph premises the successful syntactic and semantic analysis of the source code. That is, the determination of all possible relations that may exist between 2 classes, by statically analyzing the source code. This challenging task is carried out by an Abstract Syntax Tree (AST) parser, the details of which can be found in Appendix A2.

After the identification of the relations among all classes, the project graph is being built for each version and also, during the same process, the graph metrics are calculated. The calculation of graph metrics just after the creation of the graph and not in an asynchronous way, is imposed by the huge computational burden and amount of memory that the graph modelling demands. Just for a single version of a medium sized project of 2000 classes, the amount of RAM needed for the storage of the AST is about 5 GB. Consequently, a later re-loading of an entire project version would be inefficient.

All calculated metrics (Repo, Graph, Source code) are stored in the system's relational MySQL database where a significant effort was made to minimize information redundancy and maximize scalability. An overview of the database schema is presented in Appendix A4.

5.2.2 Source Code Smells Engine

One useful exploitation of the calculated metric values, is the metric-based identification of code smells. Seagle supports the identification of God Class, Data Class and Feature Envy smells. As an example, an overview of the process that is followed for the identification of the God Class code smell is presented in Figure 28.

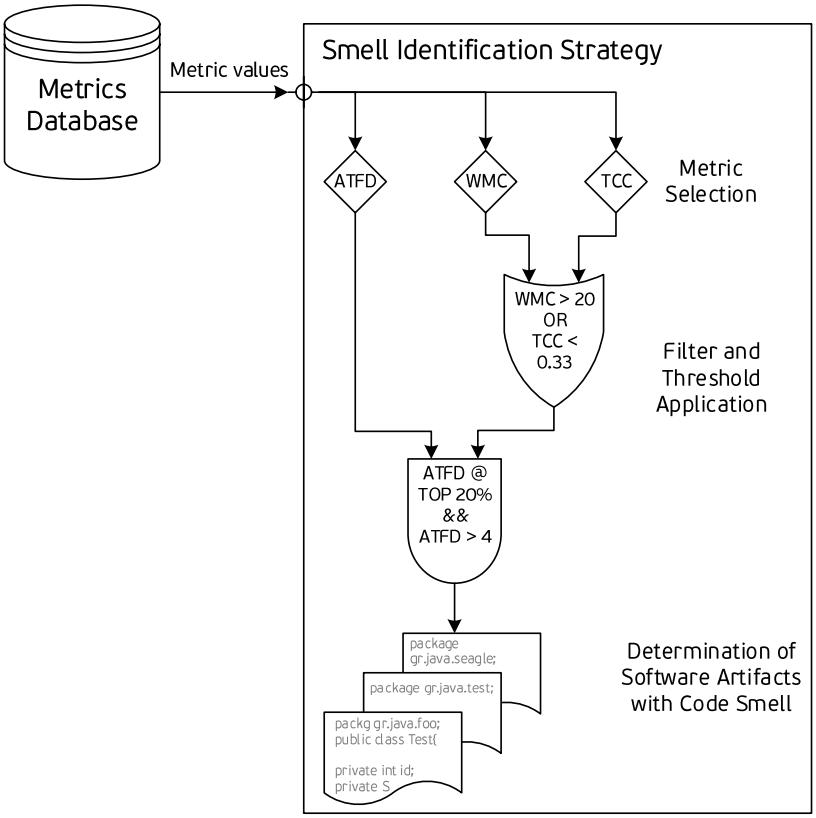


Figure 28 - Overview of the smell detection mechanism

The detection strategy starts with the selection of the retrieval of the appropriate metrics that are involved in the identification. Software artifacts are filtered based on metric values by applying specific criteria (usually thresholds). For each artifact (class, module, method etc.) a combination of metric values is applied in order to determine if it has the smell we are looking for.

5.3 Description of SEAgle's graphical user interface

The homepage of SEAgle, shown in Figure 29, awaits a single user input, which may be either:

- a project name
- a git URI from a public repository

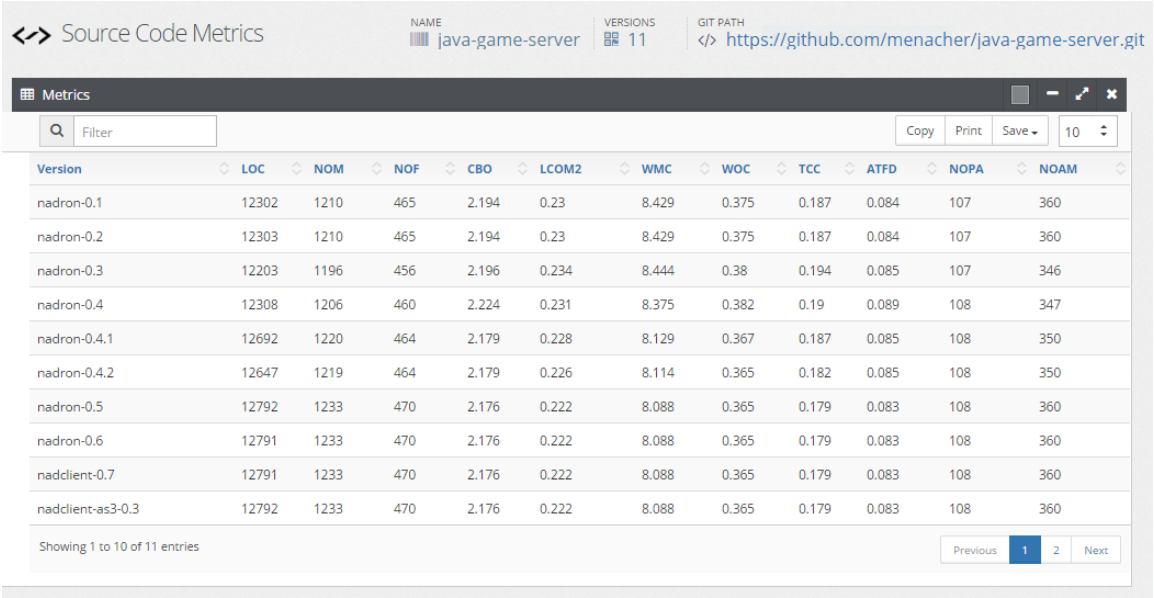
Figure 29 - SEAgle's main search page

In case a project name is entered, it will be looked for in the already analyzed projects for which results are available. If the user types in a git URI, it is also being checked whether the corresponding repository has been analyzed. If not, the user request triggers the analysis.

Since the analysis of large repositories can be time and resource consuming, the user is notified on the progress of processing. Moreover, the system can notify the user by email when the analysis has been completed. The home page offers (on the right hand side) a timeline overview of the recently analyzed projects (shown in Figure 30).

Figure 30 - Timeline of recently analyzed projects

For each analyzed project a dashboard containing a metrics overview is displayed Figure 31.



Version	LOC	NOM	NOF	CBO	LCOM2	WMC	WOC	TCC	ATFD	NOPA	NOAM
nadron-0.1	12302	1210	465	2.194	0.23	8.429	0.375	0.187	0.084	107	360
nadron-0.2	12303	1210	465	2.194	0.23	8.429	0.375	0.187	0.084	107	360
nadron-0.3	12203	1196	456	2.196	0.234	8.444	0.38	0.194	0.085	107	346
nadron-0.4	12308	1206	460	2.224	0.231	8.375	0.382	0.19	0.089	108	347
nadron-0.4.1	12692	1220	464	2.179	0.228	8.129	0.367	0.187	0.085	108	350
nadron-0.4.2	12647	1219	464	2.179	0.226	8.114	0.365	0.182	0.085	108	350
nadron-0.5	12792	1233	470	2.176	0.222	8.088	0.365	0.179	0.083	108	360
nadron-0.6	12791	1233	470	2.176	0.222	8.088	0.365	0.179	0.083	108	360
nadclient-0.7	12791	1233	470	2.176	0.222	8.088	0.365	0.179	0.083	108	360
nadclient-as3-0.3	12792	1233	470	2.176	0.222	8.088	0.365	0.179	0.083	108	360

Figure 31 - Overview of project metrics

Detailed information concering the evolution of metrics for the three examined views (commit, source code and network metrics) can be displayed by selecting “Evolution Analysis” on the left menu. As an example, in Figure 32 the evolution of commit-related metrics over the examined versions of a project are shown. The results are also shown as Tables with columns corresponding to metrics, and rows to examined versions.

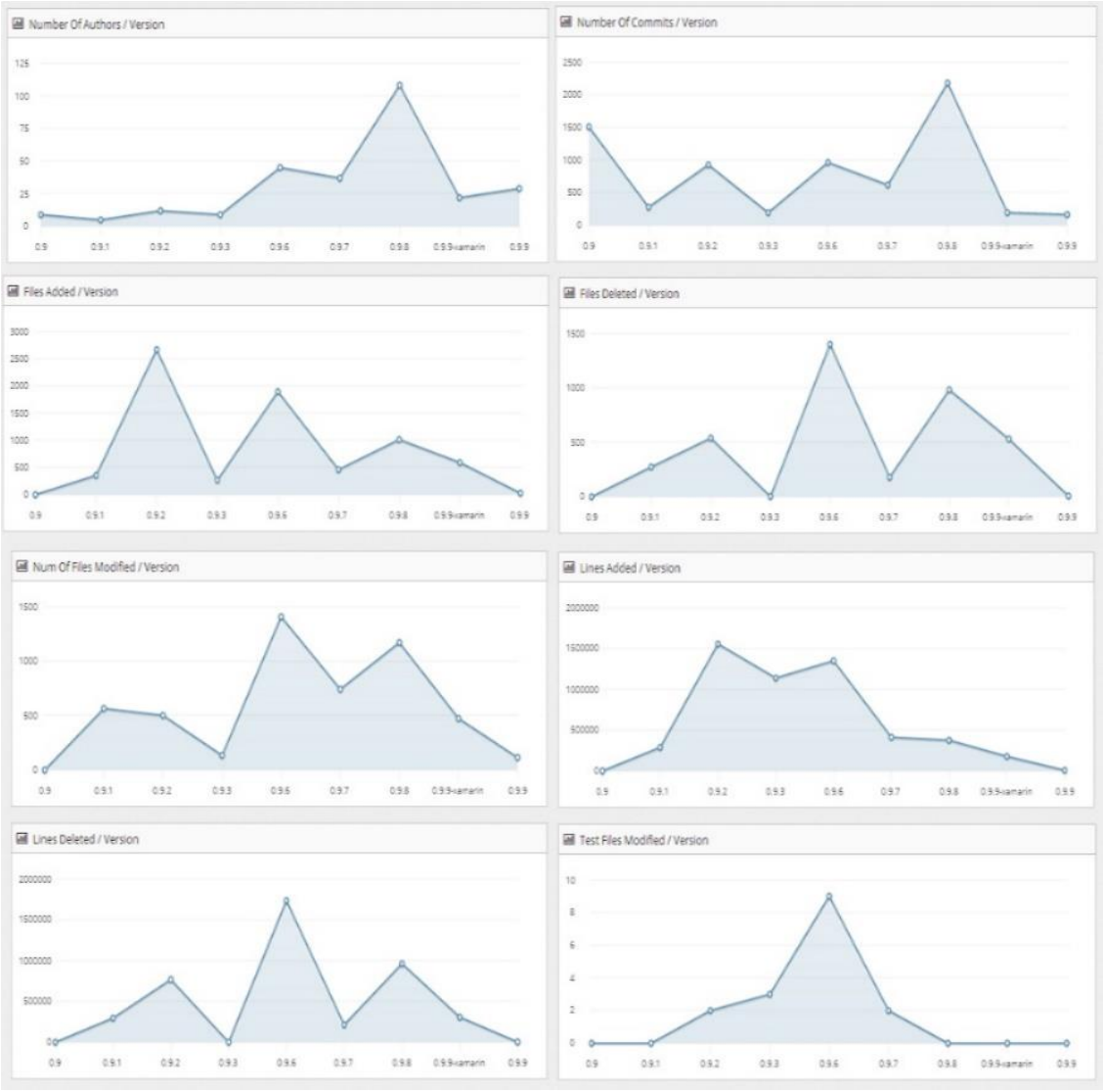


Figure 32 - Diagrams that depict metrics related to repository activity

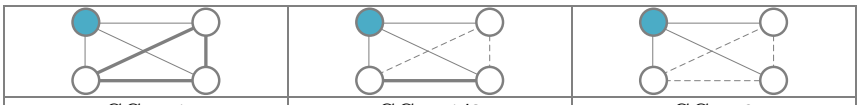
By clicking the "Save" button on every tabular representation, the corresponding data can be exported in CSV, Excel or PDF format to allow further experimentation.

5.4 Implemented Metrics

The implemented metrics can be split in 3 categories. Metrics that reflect the structure of the graph that models the system under analysis, metrics related to the activity of developers and file changes at the level of software repository and finally, source code quality metrics such as Coupling, Cohesion, Complexity end many more. The available metrics of SEAgle are summarized in Table 12

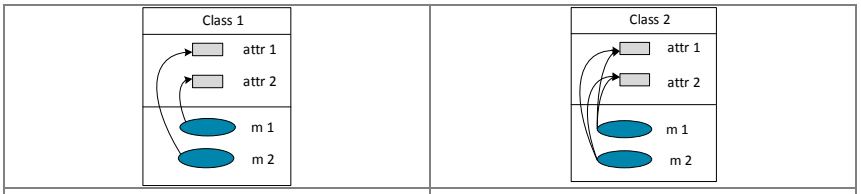
Table 12 - SEAgle metrics

Graph-Based Metrics	
Abbreviation	**Name**
NON	Number of Nodes The number of nodes of the graph that represents the system under analysis. Each node corresponds to a system class and not a file. So a class with a nested one inside her, will be modelled as two nodes with an edge between them
NOE	Number of Edges The number of associations between classes according to Law of Demeter. An edge from a class A to a class B exists if • A contains an attribute of type B • a method in A contains a local variable of type B • a method in A contains a parameter of type B • a method in A returns an object of type B • a method in A accesses a public object of type B
GD	Graph Diameter The average path distance of the graph. A more common way of measurement is the maximum of the shortest paths between any pair of nodes. But this method is susceptible to outliers (Kang et al., 2011)
DEN	Graph Density For directed, simple graphs the graph density is defined as the number of edges divided by the maximum number of edges. $$DEN = \frac{\lvert E \rvert}{\lvert V \rvert \cdot (\lvert V \rvert - 1)}$$ Where $\lvert E \rvert$ is number of edges And $\lvert V \rvert$ is the number of nodes
CC	Clustering Coefficient The Clustering Coefficient (Watts and Strogatz, 1998), measures the extend of connectedness of a node's neighbors, or in other words, it quantifies how close the neighbors of a node are to being a clique (fully connected subgraph). Seagle calculates both the CC value for each and also the average value for the whole network (system).



Commit-Related Metrics	
Abbreviation	**Name**
NATH	Number of Authors
NCOM	Number of Commits
NAF	Number of Added Files
NDF	Number of Deleted Files
NMF	Number of Modified Files
NAL	Number of Added Lines of Code
NDL	Number of Deleted Lines of Code
NATF	Number of Added Test Files
NMTF	Number of Modified Test Files

Source-Code Metrics	
Abbreviation	**Name**
LOC	Lines Of Code Blank lines and comments are omitted
NOM	Number of Methods The total number of methods, without taking into account access modifiers and method body.
NOF	Number of Fields The total number of attributes of class without taking into account access modifiers.
CBO	Coupling Between Objects Defined by Chidamber and Kemerer (1994), the coupling between objects (CBO) metric represents the number of classes coupled to a given class (efferent couplings, Ce). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.
LCOM2	Lack of Cohesion in Methods (2) This metric quantifies the level of class cohesiveness. A cohesive class carries out a single function and therefore all methods use all of the class features. A non-cohesive class implements two or more unrelated features. The first version of LCOM was introduced by Chidamber and Kemerer (Chidamber and Kemerer, 1994), however in the light of various disadvantages that arose, various modification have been proposed over the years. LCOM2 is an alternative measure of class cohesiveness proposed by Henderson-Sellers et al. (1996) and is defined as the percentage of methods that do not access a specific attribute averaged over all attributes in the class. If the number of methods or attributes is zero, LCOM2 is undefined and displayed as zero.
WMC	Weighted Methods per Class The acronym fits also well to "weighted method complexity", which is eventually the real deal with this metric. It is calculated by summing the cyclomatic complexity of all methods of a class. For the whole system level, we report the average aggregation.
WOC	Weight Of Class WOC is the ratio of public methods that are not accessors over the total number all public methods, without considering inherited methods. This metric acts as an indicator as to whether the class interface contains more methods that are data accessors rather than logic implementers. For the whole system level, we report the average aggregation.
TCC	Tight Class Cohesion



Non-Cohesive Class | Cohesive Class

	Tight Class Cohesion is the relative number of methods directly connected. Two public methods are directly connected if they access the same attribute and TCC is the ratio of directly connected methods over all possible public method connections, (i.e. M(M-1)/2, where M is the number of methods). For the whole system level, we report the average aggregation.
ATFD	Access To Foreign Data Access to Foreign Data is the number of attributes of external (with respect to the class hierarchy) classes accessed from the given class, either directly or by accessor methods.
NOPA	Number Of Public Attributes Total number of the attributes of a class that are public. For the whole system level, we report the sum aggregation
NOAM	Number of Accessor Methods Total number of methods that return an attribute of the class. For the whole system level, we report the sum aggregation
FDP	Foreign Data Providers FDP measures the number of external classes that contain ATFD attributes, i.e attributes accessed by the entity. In other words, it quantifies in how many other classes do the used "foreign" attributes belong to. This is an indication of how many external classes an entity is interested in.
LAA	Locality of Attribute Access LAA is the ratio of distinct local attribute accesses (same class attributes) of a method over the total number of variables accessed (including via accessors). Values range from 0 to 1, where 0 indicates that no local data is accessed at all

5.5 Correlation Analysis

One of the objectives of SEAgle is to go beyond simple data presentation and try to provide an inside view of the underlying variable relationships. One simple but very enlightening step toward this, is the quick and easy exploration of possible variable correlations. For example, a researcher may want to follow a hunch that in a specific project two variables correlate. By using SEAgle, he/she can accomplish, with zero configuration, the analysis of the evolution of the project under study, and after the analysis completion, he/she can explore the nature and correlation strength between all possible variable pairs of the provided metrics.

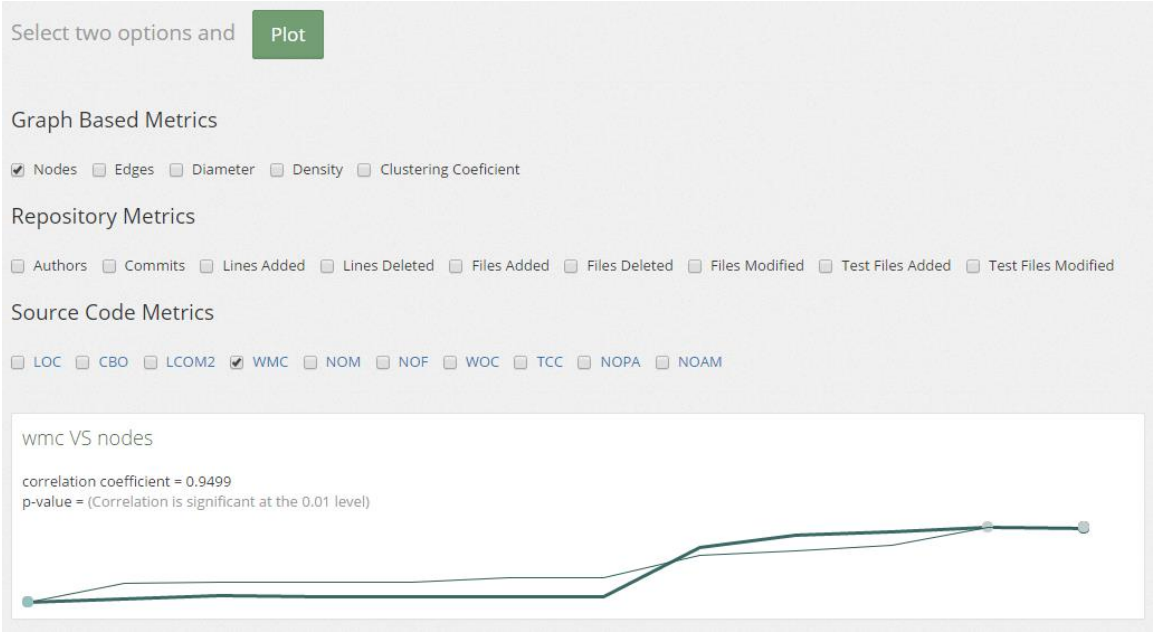


Figure 33 - SEAgle metric correlation analysis engine

"Correlation Analysis" option, which is available through the left-hand menu of the dashboard, enables a simple menu, where the user selects any two variables and plots the corresponding co-evolution graph. In this way, researches can investigate visually the way that the corresponding measures co-evolve over time. Both trends are shown on a common chart (employing a secondary y-axis) for improved readability (Figure 33).

Moreover, the Pearson correlation coefficient as well as the corresponding significance level (p-value) are shown. Currently, the platform is capable of calculating the correlation of 276 different metric pairs.

6 Threats to validity

Good judgment comes from experience, and experience comes from bad judgment.

Frederick Brooks, American computer architect

6.1 Introduction

Any novel approach along with resulting findings is unavoidably exposed to various threats to validity. The identified threats regard the categories of internal, construct and external validity (Wohlin et al., 2000).

Threats to internal validity are the factors that might affect the phenomena under investigation without the researcher's knowledge and therefore they threat the conclusions regarding possible causal relationships between depended and independent variables. Examples of threats to internal validity include wrong subject selection among humans, badly-designed questionnaires or experiments that are repeated in days that are not similar to each-other (e.g. the first in a normal day and the second in a holiday).

Threats to construct validity limit the ability to base the result of the experiment to the concept behind the experiment, in other words, construct validity refers to the degree to which an experiment captures the phenomena that was designed for. Threats of this type can emerge due to experiment design flaws such as limited measurement repetitions of an independent variable, or because of social threats such as the inherent tendency of humans to perform their best when they know they are being evaluated.

Finally, threats to external validity limit our ability to generalize the findings of the research. An example of this threat is the selection of computer science students as the sole subject group in an experiment that measures student's ability to understand programming concepts.

6.2 Description of threats

Regarding the set of techniques and measures that introduced in order to investigate the evolution of feature scattering, the following threats to construct and internal validity (Wohlin et al., 2000) can be identified. Since the application on the four case studies has been performed as an illustration of how the proposed techniques/measures can be employed, threats to external validity are not present.

The entire process is based on the assumption that the number of methods involved in the implementation of a given feature constitutes a valid measure for the quantification of feature scattering. This could potentially impose a threat to construct validity which deals with how well the selected measures or tests can stand in for the concepts of interest. According to the taxonomy by Dit et al. (2013) regarding feature location techniques, a feature's implementation can be traced down to the following measures: (1) files/classes, (2) methods/functions, (3) statements and (4) non-source code artifacts. It appears that methods as an output of feature identification is used much more frequently than any other measure and in particular it has been used in 39 out of 45 feature location approaches. As a result, the selection of methods appears to be the most reliable and sound choice.

Regarding the internal validity of the study, an identified threat is related to the presence of other features which might have not been included in our analysis. However, this threat is valid only for the investigation of reusability among features and its evolution by means of multi-dimensional scaling. The reason is that other ignored features might be interleaved with the features that have been the focus of our study. For example, a feature that exhibits relatively low reuse with other selected features, might share a large number of classes and methods with a feature that has been omitted. To mitigate this threat, anyone who aims at analyzing the degree of reuse among features should be mindful to select all possible features which are conceptually or functionally similar. On the other hand, for the techniques and measures presented in section 2.a – 2.d this threat is not present since the employed measures are not affected by the existence of other features.

Regarding the model of software evolution, it should be acknowledged that the accuracy of any prediction model is by definition constrained, especially in the software domain, since future evolution can be affected by numerous, unpredictable factors. These factors may include requirements for implementing radically novel functionality, decisions to modify the architecture, changes in the development team etc.

The proposed approach samples from distributions based on past version data to obtain several model parameters. This sampling assumes that the system under study will continue to evolve in the future in a similar manner as in the past. This means that if abrupt changes to the network topology occur due to major architectural modifications for example, the predicted future network evolution might be inaccurate. As another example, if the trend in the number of nodes and edges has been increasing during the entire history of a software project with an exception for the last version (where it could possibly drop), the model would not be able to predict this sudden reduction in the number of nodes and edges. Another source of inaccuracy is related to the fact that in order to derive the model parameters, the distributions are captured by means of curve fitting to the actual data. The corresponding mathematical function might not always have an ideal fit to the actual data points, affecting the accuracy of the extracted parameter.

Apart from the aforementioned limitations, the proposed prediction model suffers from the usual threats to external and internal validity. The fact that the model is evaluated against ten projects, unavoidably limits the possibility to extensively generalize our findings. This threat is related to the observation of general trends which are applicable to all projects (such as preferential attachment). It is always possible that another set of projects might exhibit different phenomena. A similar threat stems from the fact that all analyzed projects are developed in Java thus limiting the ability to generalize to other object-oriented languages. However, since a large part of the model consists in learning from past versions, we believe that it can be successfully adapted to any software project regardless of its particularities.

Concerning the internal validity (i.e. the parameters that might affect the evolutionary trends that we are trying to predict), it is reasonable to assume that numerous other factors, which affect software growth, might have not been taken into consideration. As an example, a forecasting model could be augmented by considering the co-evolving developer community or dependencies among documented requirements. However, the proposed simulation approach can be considered as incremental in the sense that additional parameters can be easily integrated. The same holds for the incorporation of additional domain rules which might be representative for a particular application domain or the habits of a particular development team.

7 Conclusions and Future Work

You never finish a program; you just stop working on it.

Anonymous

7.1 Conclusions

By drawing analogies between networks in other domains and software, this thesis proposed graph-based tools and techniques for different aspects of software lifecycle that facilitate the evolutionary analysis of object-oriented systems.

First of all, this thesis introduces a set of techniques for the analysis of the evolution in feature scattering, based on the classes and methods involved in the implementation of high-level, distinct and observable pieces of functionality. The proposed analyses have been applied on several versions of four open-source projects. Based on the results, the applied techniques appear to be promising, since they allow software stakeholders to assess the evolution of feature scattering and gain insight into the associated implications. In particular, the obtained visualizations facilitate the study of feature spreading in terms of the number of classes and methods. A more in-depth analysis can be performed by examining the distribution of methods contributing to the implementation of the examined features and the use of the Gini coefficient to determine whether this distribution tends to become more unbalanced over time. Since similar features usually rely on common methods, the investigation of feature scattering should consider the corresponding degree of method reuse among features, as we have shown by means of (Multidimensional Scaling – MDS) charts. Finally, the impact of three widely used refactorings on feature scattering was studied, leading to the conclusion that generic rules can be employed to assess the circumstances under which a refactoring improves or deteriorates the distribution of feature functionality.

Successful and efficient maintenance of software systems requires thorough system understanding, otherwise the quality of the system will deteriorate. A visual exploration of the modules that implement specific features can assist developers in better understanding the structure of the system, especially in situations where developers that perform maintenance are not the ones that developed the system initially. Furthermore, the systematic observation of features that share common methods can provide significant assistance to maintainers during module refactoring and restructuring. In the same way, a system maintainer can recognize that for example it is not usual for two features with similar functionality to be apart from each other in the MDS chart and if this happens, it surely indicates existence of code clones.

A second major chapter of this thesis introduces a model for software evolution that incorporates findings regarding the growth patterns of software networks, such as the conformance to the preferential attachment and the duplication model. Other crucial model parameters such as the effect of node age and the number of node and edge removals are extracted by sampling from distributions formed by analyzing past versions and therefore the prior evolution is taken into account. One major contribution of the proposed model is that by acknowledging the importance of domain specific characteristics, the model has been enhanced by rules specific to object-oriented design such as the rules that guide the behavior of abstract classes. Evaluation against 10 open-source projects showed that forecasting can provide sufficient insight to the future evolution trends of systems. Software maintenance can benefit from the application of such models by focusing on parts of the network whose properties tend to deteriorate. The proposed model has been implemented as an eclipse plugin and is freely available on the project's web page.

The majority of forecasting approaches consider the modelling and prediction of a single or several software characteristics. However, the structural and conceptual complexity of a software system cannot be reflected in several metric values. Consequently, a process capable of predicting the structure of a software system as a whole, allows maintainers to gain better insight into the future state of modules along with their dependencies. Experimental evaluation revealed that the model is sufficiently capable of predicting the evolution of fan-in and fan-out couplings of packages and therefore, can provide significant assistance to maintainers in their efforts to reduce coupling. In the light of a trend for increased coupling in a package, developers can focus refactoring or other

preventive maintenance activities at an early stage and therefore prevent future costs and reduce future bugs.

Finally, in the context of this thesis, a platform for the facilitation of software engineering research was created. The tool under the name 'SEAgle' is a web-based platform that enables users to analyze any git open source repository that is freely available on the internet. SEAgle calculates a multitude of metrics of three different categories and publishes the results in a free online web page. Categories include graph-based metrics such as number of nodes, clustering coefficient and graph density, repo-based metrics such as number of commits per version and source code metrics such as complexity, coupling and cohesion. Also, by understanding the fact that empirical studies very often focus on the investigation of relations among variables, the tool offers direct correlation analysis between any two monitored variables.

The ambition of SEAgle is to significantly reduce the time needed for the collection and manipulation of metric data regarding object oriented systems. By simplifying and expediting the process of source code collection, metric calculation and data retrieval, SEAgle can assist software practitioners and researchers in their quality assessment efforts thereby improving the quality of delivered software products.

7.2 Future Work

Future work can be divided along two axes. The first one includes the enrichment of the proposed methods and the second one refers to the extension of the SEAgle platform.

The incorporation of additional parameters in the modeling of software systems, such as the investigation of developer collaboration networks, will improve the expressiveness of the corresponding model. It will also enable the parallel study of the co-evolution between software artifacts and networks formed by people. The parallel networks could be enriched by the inclusion of defect networks, where interdependencies among defects will be represented. By taking into account the relations among defects, we can co-analyze the evolution of specific modules along with the evolution of defects that appeared in these modules, create associations between software and defect networks and if possible, predict software modules that exhibit increased error proneness.

Another aspect that will enhance the predictive ability of the model is the deepening of the code analysis to the method level and the inclusion of method-related data in the

calculated distributions. In this way, we can better monitor the specific method invocations of each class and actually measure the extent to which dependencies are used. These measurements will give insight into the amount of foreign dependencies that are actually exploited by the hosting classes.

Furthermore, by considering contemporary software architectures that involve multiple paradigms such as server and client-based programming, database engineering, web design, file management and network bandwidth control in a single software solution, an interesting line of future research would be towards this direction. Modeling such multi-paradigm software systems becomes a challenge due to the non-homogeneous nodes of the corresponding network. The complexity of such systems usually leads to situations where the source of a defect is not clear and may scatter to any of the aforementioned categories. A systematic analysis of modern multi-disciplinary systems will allow the better exploration of the underlying interdependencies in an attempt to identify critical details that affect the stability of these systems.

Regarding SEAgle, a line of feature expanding is the addition of bug repository analysis and mailing list mining capabilities. A second major goal is the integration of a the simulation module for the forecasting of future software trends in the form of a web service under the SEAgle umbrella. These feature additions will not only strengthen the analytical power of the platform, but also will help SEAgle broaden its user base.

Another useful feature addition is the ability to provide analysis of web systems written in JavaScript, which currently dominates the web application development landscape by allowing both client- and server-side programming. The analysis of such systems poses serious challenges due to the multitude of technologies that they employ. Nevertheless, a platform that is capable of analyzing modern JavaScript systems will significantly broaden the toolbox of every maintainer.

Funding

Research conducted in the context of this thesis has been funded by the following sources:

- A 5-year scholarship from the Bodossaki Foundation.
- A national research project that has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) under the research funding program: Thalis – Athens University of Economics and Business - Software Engineering Research Platform.
- A national research project that has been co-financed by the European Union (European Social Fund – ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) under the research funding program: Archimedes III – Alexander Technological Institute of Thessaloniki – Comparative Study of Library, Open-Source and Industrial Software.
- This research has also been co-financed by the Basic Research Project funded by the Research Committee of the University of Macedonia titled: “Investigating historical trends concerning design problems in Object-Oriented Systems”

Publications

The work of this dissertation has been documented in a number of papers that have been published in International Conferences and Journals.

Journals

1. **T. Chaikalis** and A. Chatzigeorgiou, "Forecasting Java Software Evolution Trends employing Network Models", IEEE Transactions on Software Engineering, vol. 41, no 6, June 2015, pp. 582-602.
2. **T. Chaikalis**, A. Chatzigeorgiou and G. Examiliotou, "Investigating the Effect of Evolution and Refactorings on Feature Scattering", Software Quality Journal, vol. 23, no.1, 2015, pp. 79-105

Conferences

1. A. Chatzigeorgiou, **T. Chaikalis**, G. Paschalidou, N. Vesyropoulos, C. K. Georgiadis and E. Stiakakis, "A Taxonomy of Evaluation Approaches in Software Engineering", 7th Balkan Conference in Informatics, Craiova, Romania, September 2-4, 2015, Best Paper Award.
2. **T. Chaikalis**, A. Chatzigeorgiou, A. Ampatzoglou, I. Deligiannis, "Assessing the Evolution of Quality in Software Libraries", 7th Balkan Conference in Informatics, Craiova, Romania, September 2-4, 2015.
3. A. Zaimi, A. Ampatzoglou, N. Triantafyllidou, A. Chatzigeorgiou, A. Mavridis, **T. Chaikalis**, I. Deligiannis, P. Sfetsos, I. Stamelos, "A Case Study on Reusing Third-Party Libraries in Open-Source Software Development", 7th Balkan Conference in Informatics, Craiova, Romania, September 2-4, 2015.
4. **T. Chaikalis**, E.Ligu, G. Melas and A. Chatzigeorgiou, "SEAgle: Effortless Software Evolution Analysis", 30th International Conference on Software Maintenance and Evolution (ICSME'2014), Tool Demonstration Track, Victoria, British Columbia, Canada, Sept. 28 – Oct. 3, 2014.
5. E. Ligu, A. Chatzigeorgiou, **T. Chaikalis** and N. Ygeionomakis, "Identification of Refused Bequest Code Smells", 29th IEEE International Conference on Software Maintenance (ICSM'2013), Eindhoven, Netherlands, September 23-26, 2013.
6. E. Ligu, **T. Chaikalis** and A. Chatzigeorgiou, "BuCo Reporter: Mining Software and Bug Repositories", 6th Balkan Conference in Informatics (BCI'2013), Thessaloniki, Greece, September 19-21, 2013.
7. **T. Chaikalis**, G. Melas and A. Chatzigeorgiou, "SEANets: Software Evolution Analysis with Networks", 28th IEEE International Conference on Software Maintenance (ICSM'2012), Tool Demonstration Track, Riva del Garda, Trento, Italy, September 23rd – 30th, 2012.
8. **T. Chaikalis** and A. Chatzigeorgiou, "Investigating the Evolution of Feature Scattering", 6th International Workshop on Software Quality and Maintainability (SQM 2012), collocated with 16th European Conference on Software Maintenance and Reengineering (CSMR 2012), Szeged, Hungary, March 27, 2012

References

_____, 2014. Forecasting trends in Software Evolution: Supplemental Material [WWW Document]. URL http://se.uom.gr/index.php/software-evolution-model/

Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., Merlo, E., 2002. Recovering Traceability Links Between Code and Documentation. IEEE Trans Softw Eng 28, 970–983. doi:10.1109/TSE.2002.1041053

Antoniol, G., Casazza, G., Di Penta, M., Merlo, E., 2001. Modeling clones evolution through time series, in: IEEE International Conference on Software Maintenance, 2001. Proceedings. Presented at the IEEE International Conference on Software Maintenance, 2001. Proceedings, pp. 273–280. doi:10.1109/ICSM.2001.972740

Arisholm, E., Briand, L.C., 2006. Predicting Fault-prone Components in a Java Legacy System, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06. ACM, New York, NY, USA, pp. 8–17. doi:10.1145/1159733.1159738

aTunes, 2013. . ATunes Audio Play. Organ. URL http://www.atunes.org

Barabási, A.-L., Albert, R., 1999a. Emergence of Scaling in Random Networks. Science 286, 509–512. doi:10.1126/science.286.5439.509

Barabási, A.-L., Albert, R., 1999b. Emergence of Scaling in Random Networks. Science 286, 509–512. doi:10.1126/science.286.5439.509

Bartholomew, D.J., Steele, F., Moustaki, I., Galbraith, J., 2008. Analysis of Multivariate Social Science Data, Second Edition, 2 edition. ed. Chapman and Hall/CRC, Boca Raton.

Basili, V.R., Briand, L.C., Melo, W.L., 1996. A validation of object-oriented design metrics as quality indicators. IEEE Trans. Softw. Eng. 22, 751–761. doi:10.1109/32.544352

Bevan, J., Whitehead,Jr., E.J., Kim, S., Godfrey, M., 2005. Facilitating Software Evolution Research with Kenyon, in: Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13. ACM, New York, NY, USA, pp. 177–186. doi:10.1145/1081706.1081736

Bhattacharya, P., Iliofotou, M., Neamtiu, I., Faloutsos, M., 2012. Graph-based analysis and prediction for software evolution, in: 2012 34th International Conference on Software Engineering (ICSE). Presented at the 2012 34th International Conference on Software Engineering (ICSE), pp. 419–429. doi:10.1109/ICSE.2012.6227173

Biggerstaff, T.J., Mitbander, B.G., Webster, D.E., 1994. Program Understanding and the Concept Assignment Problem. Commun ACM 37, 72–82. doi:10.1145/175290.175300

Boehm, B.W., 1981. Software Engineering Economics, 1 edition. ed. Prentice Hall, Englewood Cliffs, N.J.

Chaikalis, T., Melas, G., Ligu, E., Chatzigeorgiou, A., 2014. Seagle: Effortless Software Evolution Analysis, in: Tool Demonstration Track. Presented at the 30th International Conference on Software Maintenance and Evolution (ICSME'2014), Victoria, British Columbia, Canada, pp. 581–584.

Chang, C.K., Jiang, H., Di, Y., Zhu, D., Ge, Y., 2008. Time-line based model for software project scheduling with genetic algorithms. Inf. Softw. Technol. 50, 1142–1154. doi:10.1016/j.infsof.2008.03.002

Chaturvedi, K.K., Sing, V.B., Singh, P., 2013. Tools in Mining Software Repositories, in: 2013 13th International Conference on Computational Science and Its Applications (ICCSA). Presented at the 2013 13th International Conference on Computational Science and Its Applications (ICCSA), pp. 89–98. doi:10.1109/ICCSA.2013.22

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Trans. Softw. Eng. 20, 476–493. doi:10.1109/32.295895

Choi, S., Cha, S., 2010. A survey of Binary similarity and distance measures. J. Syst. Cybern. Inform. 43–48.

Chung, F., Lu, L., Dewey, T.G., Galas, D.J., 2003. Duplication models for biological networks. J. Comput. Biol. J. Comput. Mol. Cell Biol. 10, 677–687. doi:10.1089/106652703322539024

Chung, F.R.K., Lu, L., 2006. Complex graphs and networks. American Mathematical Society, Providence, RI.

Coleman, D., Ash, D., Lowther, B., Oman, P., 1994. Using metrics to evaluate software system maintainability. Computer 27, 44–49. doi:10.1109/2.303623

Concas, G., Marchesi, M., Pinna, S., Serra, N., 2007. Power-Laws in a Large Object-Oriented Software System. IEEE Trans. Softw. Eng. 33, 687–708. doi:10.1109/TSE.2007.1019

Conejero, J.M., Figueiredo, E., Garcia, A., Hernández, J., Jurado, E., 2009. Early Crosscutting Metrics as Predictors of Software Instability, in: Oriol, M., Meyer, B. (Eds.), Objects, Components, Models and Patterns, Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, pp. 136–156.

Cox, M.A.A., Cox, T.F., 2008. Multidimensional Scaling, in: Handbook of Data Visualization, Springer Handbooks Comp.Statistics. Springer Berlin Heidelberg, pp. 315–347.

Cytoscape: An Open Source Platform for Complex Network Analysis and Visualization [WWW Document], 2014. URL http://www.cytoscape.org/ (accessed 4.11.16).

D'Ambros, M., Lanza, M., 2008. A Flexible Framework to Support Collaborative Software Evolution Analysis, in: 12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008. Presented at the 12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008, pp. 3–12. doi:10.1109/CSMR.2008.4493295

Devroye, L., 1986. Non-Uniform Random Variate Generation. Springer-Verlag, New York.

Dijkstra, E.W., 1959. A note on two problems in connexion with graphs. Numer. Math. 1, 269–271. doi:10.1007/BF01386390

Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D., 2013. Feature location in source code: a taxonomy and survey. J. Softw. Evol. Process 25, 53–95. doi:10.1002/smr.567

Eaddy, M., Aho, A., Murphy, G.C., 2007. Identifying, Assigning, and Quantifying Crosscutting Concerns, in: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques, ACoM '07. IEEE Computer Society, Washington, DC, USA, p. 2–. doi:10.1109/ACOM.2007.4

Eaddy, M., Zimmermann, T., Sherwood, K.D., Garg, V., Murphy, G.C., Nagappan, N., Aho, A.V., 2008. Do Crosscutting Concerns Cause Defects? IEEE Trans. Softw. Eng. 34, 497–515. doi:10.1109/TSE.2008.36

Easley, D., Kleinberg, J., 2010. Networks, crowds, and markets reasoning about a highly connected world. Cambridge University Press, New York.

Eisenbarth, T., Koschke, R., Simon, D., 2003. Locating features in source code. IEEE Trans. Softw. Eng. 29, 210–224. doi:10.1109/TSE.2003.1183929

Faloutsos, M., Faloutsos, P., Faloutsos, C., 1999. On Power-law Relationships of the Internet Topology, in: Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '99. ACM, New York, NY, USA, pp. 251–262. doi:10.1145/316188.316229

Filho, F.C., Cacho, N., Figueiredo, E., Maranhão, R., Garcia, A., Rubira, C.M.F., 2006. Exceptions and Aspects: The Devil is in the Details, in: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '06/FSE-14. ACM, New York, NY, USA, pp. 152–162. doi:10.1145/1181775.1181794

Fischer, M., Gall, H., 2004. Visualizing feature evolution of large-scale software based on problem and modification report data. J. Softw. Maint. Evol. Res. Pract. 16, 385–403. doi:10.1002/smr.302

Fortuna, M.A., Bonachela, J.A., Levin, S.A., 2011. Evolution of a modular software network. Proc. Natl. Acad. Sci. doi:10.1073/pnas.1115960108

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., Gamma, E., 1999. Refactoring: Improving the Design of Existing Code, 1 edition. ed. Addison-Wesley Professional, Reading, MA.

FreeCol, 2013. . Free. - Colon. Am. URL http://www.freecol.org/

Ganter, B., Wille, R., 1999. Formal Concept Analysis. Springer Berlin Heidelberg, Berlin, Heidelberg.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A., 2005. Modularizing Design Patterns with Aspects: A Quantitative Study, in: Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD '05. ACM, New York, NY, USA, pp. 3–14. doi:10.1145/1052898.1052899

Genero, M., Olivas, J., Piattini, M., Romero, F., 2001. Using Metrics to Predict OO Information Systems Maintainability, in: Dittrich, K.R., Geppert, A., Norrie, M.C. (Eds.), Advanced Information Systems Engineering, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 388–401.

Gephi - The Open Graph Viz Platform [WWW Document], 2014. URL https://gephi.org/ (accessed 4.11.16).

Ghezzi, C., Jazayeri, M., Mandrioli, D., 2003. Fundamentals of software engineering. Prentice Hall, Upper Saddle River, N.J.

Gibbs, C., Liu, C.R., Coady, Y., 2005. Sustainable System Infrastructure and Big Bang Evolution: Can Aspects Keep Pace?, in: Black, A.P. (Ed.), ECOOP 2005 - Object-Oriented Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 241–261.

Gini, C., 1921. Measurement of Inequality of Incomes. Econ. J. 31, 124–126. doi:10.2307/2223319

Gîrba, T., Ducasse, S., 2006. Modeling History to Analyze Software Evolution: Research Articles. J Softw Maint Evol 18, 207–236. doi:10.1002/smr.v18:3

Gîrba, T., Ducasse, S., Lanza, M., 2004. Yesterday"s Weather: Guiding Early Reverse Engineering Efforts by Summarizing the Evolution of Changes., in: ICSM. IEEE Computer Society, pp. 40–49.

Godfrey, M.W., German, D.M., 2008. The past, present, and future of software evolution, in: Frontiers of Software Maintenance, 2008. FoSM 2008. Presented at the

Frontiers of Software Maintenance, 2008. FoSM 2008., pp. 129–138. doi:10.1109/FOSM.2008.4659256

Goeminne, M., Mens, T., 2011. Evidence for the pareto principle in open source software activity, in: In the Joint Porceedings of the 1st International Workshop on Model Driven Software Maintenance and 5th International Workshop on Software Quality and Maintainability. pp. 74–82.

González-Torres, A., Therón, R., García-Peñalvo, F.J., Wermelinger, M., Yu, Y., 2011. Maleku: An evolutionary visual software analysis tool for providing insights into software evolution, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM). Presented at the 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 594–597. doi:10.1109/ICSM.2011.6080838

Gotel, O.C.Z., Finkelstein, C.W., 1994. An analysis of the requirements traceability problem, in: , Proceedings of the First International Conference on Requirements Engineering, 1994. Presented at the , Proceedings of the First International Conference on Requirements Engineering, 1994, pp. 94–101. doi:10.1109/ICRE.1994.292398

Granger, C.W.J., Newbold, P., 1977. Forecasting economic time series. Academic Press.

Greenwood, P., Bartolomei, T., Figueiredo, E., Dosea, M., Garcia, A., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A., 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, in: Ernst, E. (Ed.), ECOOP 2007 – Object-Oriented Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 176–200.

Greevy, O., Ducasse, S., Gîrba, T., 2006. Analyzing software evolution through feature views. J. Softw. Maint. Evol. Res. Pract. 18, 425–456. doi:10.1002/smr.340

GUESS: The Graph Exploration System [WWW Document], 2014. URL http://graphexploration.cond.org/ (accessed 4.11.16).

Henderson-Sellers, B., Constantine, L.L., Graham, I.M., 1996. Coupling and cohesion (toward a valid metrics suite for object-oriented analysis and design). Object-Oriented Syst. 3, 143–158.

HFS Explorer, 2013. . HFS Explor. URL http://www.catacombae.org/hfsx.html

Holmes, R., Begel, A., 2008. Deep intellisense: a tool for rehydrating evaporated information. ACM Press, p. 23. doi:10.1145/1370750.1370755

Hosseini, S.M.H., Kesler, S.R., 2013. Influence of Choice of Null Network on Small-World Parameters of Structural Correlation Networks. PLoS ONE 8, e67354. doi:10.1371/journal.pone.0067354

IEEE Std 14764, 2006. . ISOIEC 147642006 E IEEE Std 14764-2006 Revis. IEEE Std 1219-1998 0_1-46. doi:10.1109/IEEESTD.2006.235774

igraph [WWW Document], 2014. URL http://igraph.org/redirect.html (accessed 4.11.16).

Java Profiler - JProfiler [WWW Document], 2012. URL http://www.ej-technologies.com/products/jprofiler/overview.html (accessed 4.5.16).

JDeodorant, 2014. . JDeodorant - Qual. Matters Most. URL http://www.jdeodorant.com/

JEdit, 2013. . JEdit - Program. Text Ed. URL http://www.jedit.org/

Jenkins, S., Kirk, S.R., 2007. Software Architecture Graphs As Complex Networks: A Novel Partitioning Scheme to Measure Stability and Evolution. Inf Sci 177, 2587–2601. doi:10.1016/j.ins.2007.01.021

Jetty, 2014. . Jetty - Servlet Engine HTTP Serv. URL http://www.eclipse.org/jetty/

JFreeChart, 2013. . JFreeChart. URL http://www.jfree.org/jfreechart

Jiang, T., Tan, L., Kim, S., 2013. Personalized defect prediction, in: 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE). Presented at the 2013 IEEE/ACM 28th International Conference on Automated Software Engineering (ASE), pp. 279–289. doi:10.1109/ASE.2013.6693087

Jmol, 2013. . Jmol Open-Source Java Viewer Chem. Struct. 3D. URL http://www.jmol.org/

Jørgensen, M., 2007. Forecasting of software development work effort: Evidence on expert judgement and formal models. Int. J. Forecast. 23, 449–462. doi:10.1016/j.ijforecast.2007.05.008

Jun, E.S., Lee, J.K., 2001. Quasi-optimal case-selective neural network model for software effort estimation. Expert Syst. Appl. 21, 1–14. doi:10.1016/S0957-4174(01)00021-5

Kagdi, H., 2007. Improving Change Prediction with Fine-grained Source Code Mining, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07. ACM, New York, NY, USA, pp. 559–562. doi:10.1145/1321631.1321742

Kang, U., Tsourakakis, C.E., Appel, A.P., Faloutsos, C., Leskovec, J., 2011. HADI: Mining Radii of Large Graphs. ACM Trans Knowl Discov Data 5, 8:1–8:24. doi:10.1145/1921632.1921634

Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A., 2007. Predicting Faults from Cached History, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07. IEEE Computer Society, Washington, DC, USA, pp. 489–498. doi:10.1109/ICSE.2007.66

Kleinberg, J., 2000. The Small-World Phenomenon: An Algorithmic Perspective, in: In Proceedings of the 32nd ACM Symposium on Theory of Computing. pp. 163–170.

Kleinberg, J.M., 1999. Authoritative Sources in a Hyperlinked Environment. J ACM 46, 604–632. doi:10.1145/324133.324140

Kontogiannis, K.A., Demori, R., Merlo, E., Galler, M., Bernstein, M., 1996. Pattern matching for clone and concept detection. Autom. Softw. Eng. 3, 77–108. doi:10.1007/BF00126960

Koschke, R., Quante, J., 2005. On Dynamic Feature Location, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05. ACM, New York, NY, USA, pp. 86–95. doi:10.1145/1101908.1101923

Kothari, J., Denton, T., Mancoridis, S., Shokoufandeh, A., 2006. On Computing the Canonical Features of Software Systems, in: 13th Working Conference on Reverse Engineering, 2006. WCRE '06. Presented at the 13th Working Conference on Reverse Engineering, 2006. WCRE '06, pp. 93–102. doi:10.1109/WCRE.2006.39

Kuhn, A., Loretan, P., Nierstrasz, O., 2008. Consistent Layout for Thematic Software Maps, in: 15th Working Conference on Reverse Engineering, 2008. WCRE '08. Presented at the 15th Working Conference on Reverse Engineering, 2008. WCRE '08, pp. 209–218. doi:10.1109/WCRE.2008.45

Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tomkins, A., Upfal, E., 2000. Stochastic models for the Web graph, in: 41st Annual Symposium on Foundations of Computer Science, 2000. Proceedings. Presented at the 41st Annual Symposium on Foundations of Computer Science, 2000. Proceedings, pp. 57–65. doi:10.1109/SFCS.2000.892065

Lehman, M.M., 1980. Programs, life cycles, and laws of software evolution. Proc. IEEE 68, 1060–1076. doi:10.1109/PROC.1980.11805

Lehmann, E.L., Romano, J.P., 2005. Testing Statistical Hypotheses. Springer.

Leskovec, J., Backstrom, L., Kumar, R., Tomkins, A., 2008. Microscopic Evolution of Social Networks, in: Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '08. ACM, New York, NY, USA, pp. 462–470. doi:10.1145/1401890.1401948

Leskovec, J., Kleinberg, J., Faloutsos, C., 2005. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations, in: In KDD. ACM Press, pp. 177–187.

Li, H., Zhao, H., Cai, W., Xu, J.-Q., Ai, J., 2013. A modular attachment mechanism for software network evolution. Phys. Stat. Mech. Its Appl. 392, 2025–2037. doi:10.1016/j.physa.2013.01.035

Li, J., Ruhe, G., Al-Emran, A., Richter, M.M., 2007. A flexible method for software effort estimation by analogy. Empir. Softw. Eng. 12, 65–106. doi:10.1007/s10664-006-7552-4

Li, L., Alderson, D., Doyle, J.C., Willinger, W., 2005. Towards a theory of scale-free graphs: Definition, properties, and implications. Internet Math. 2, 4.

Li, Y.F., Xie, M., Goh, T.N., 2009. A study of project selection and feature weighting for analogy based software cost estimation. J. Syst. Softw. 82, 241–252. doi:10.1016/j.jss.2008.06.001

Lieberherr, K.J., Holland, I.M., 1989. Assuring good style for object-oriented programs. IEEE Softw. 6, 38–48. doi:10.1109/52.35588

Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P., 2009. Sourcerer: mining and searching internet-scale software repositories. Data Min. Knowl. Discov. 18, 300–336. doi:10.1007/s10618-008-0118-x

Liu, D., Marcus, A., Poshyvanyk, D., Rajlich, V., 2007. Feature Location via Information Retrieval Based Filtering of a Single Scenario Execution Trace, in: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07. ACM, New York, NY, USA, pp. 234–243. doi:10.1145/1321631.1321667

Lorenz, M.O., 1905. Methods of Measuring the Concentration of Wealth. Publications of the American Statistical Association.

Louridas, P., Spinellis, D., Vlachos, V., 2008. Power Laws in Software. ACM Trans Softw Eng Methodol 18, 2:1–2:26. doi:10.1145/1391984.1391986

Ma, Y., He, K., Liu, J., 2008. Network Motifs in Object-Oriented Software Systems. ArXiv08083292 Cs.

Marcus, A., Maletic, J.I., 2003. Recovering Documentation-to-source-code Traceability Links Using Latent Semantic Indexing, in: Proceedings of the 25th International Conference on Software Engineering, ICSE '03. IEEE Computer Society, Washington, DC, USA, pp. 125–135.

Marcus, A., Sergeyev, A., Rajlich, V., Maletic, J.I., 2004. An Information Retrieval Approach to Concept Location in Source Code, in: Proceedings of the 11th Working Conference on Reverse Engineering, WCRE '04. IEEE Computer Society, Washington, DC, USA, pp. 214–223.

Martin, R.C., 2003. Agile software development: principles, patterns, and practices. Prentice Hall, Upper Saddle River, N.J.

McDonald, J., 2009. Handbook of Biological Statistics. Sparky House Publishing.

Meneely, A., Williams, L., Snipes, W., Osborne, J., 2008. Predicting Failures with Developer Networks and Social Network Analysis, in: Proceedings of the 16th

ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16. ACM, New York, NY, USA, pp. 13–23. doi:10.1145/1453101.1453106

Mens, T., 2008. Introduction and Roadmap: History and Challenges of Software Evolution, in: Software Evolution. Springer Berlin Heidelberg, pp. 1–11.

Myers, C.R., 2003. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68, 46116. doi:10.1103/PhysRevE.68.046116

Nagappan, N., Ball, T., 2005. Use of Relative Code Churn Measures to Predict System Defect Density, in: Proceedings of the 27th International Conference on Software Engineering, ICSE '05. ACM, New York, NY, USA, pp. 284–292. doi:10.1145/1062455.1062514

Nagappan, N., Ball, T., Zeller, A., 2006. Mining Metrics to Predict Component Failures, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06. ACM, New York, NY, USA, pp. 452–461. doi:10.1145/1134285.1134349

Nagwani, N.K., Bhansali, A., 2010. A Data Mining Model to Predict Software Bug Complexity Using Bug Estimation and Clustering, in: 2010 International Conference on Recent Trends in Information, Telecommunication and Computing (ITC). Presented at the 2010 International Conference on Recent Trends in Information, Telecommunication and Computing (ITC), pp. 13–17. doi:10.1109/ITC.2010.56

Naseem, R., Maqbool, O., Muhammad, S., 2011. Improved Similarity Measures for Software Clustering, in: 2011 15th European Conference on Software Maintenance and Reengineering (CSMR). Presented at the 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), pp. 45–54. doi:10.1109/CSMR.2011.9

NetMiner 4 - Social Network Analysis Software [WWW Document], 2014. URL https://sites.google.com/site/netminer4/ (accessed 4.11.16).

Networks / Pajek [WWW Document], 2014. URL http://vlado.fmf.uni-lj.si/pub/networks/pajek/ (accessed 4.11.16).

NetworkX [WWW Document], 2014. URL https://networkx.github.io/documentation/latest/overview.html (accessed 4.11.16).

Ohloh [WWW Document], 2014. . Ohloh Open Source Netw. URL https://www.ohloh.net/ (accessed 6.30.14).

Page, L., Brin, S., Motwani, R., Winograd, T., 1999. The PageRank Citation Ranking: Bringing Order to the Web. [WWW Document]. URL http://ilpubs.stanford.edu:8090/422/ (accessed 5.27.16).

Parnas, D.L., 1994. Software Aging, in: Proceedings of the 16th International Conference on Software Engineering, ICSE '94. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 279–287.

Paymal, P., Patil, R., Bhowmick, S., Siy, H., 2011. Measuring disruption from software evolution activities using graph-based metrics, in: 2011 27th IEEE International Conference on Software Maintenance (ICSM). Presented at the 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 532–535. doi:10.1109/ICSM.2011.6080825

Pendharkar, P.C., Subramanian, G.H., Rodger, J.., 2005. A probabilistic model for predicting software development effort. IEEE Trans. Softw. Eng. 31, 615–624. doi:10.1109/TSE.2005.75

Pfeiffer III, J.J., La Fond, T., Moreno, S., Neville, J., 2012. Fast Generation of Large Scale Social Networks with Clustering. ArXiv12024805 Phys.

Pinzger, M., Gall, H., Fischer, M., Lanza, M., 2005. Visualizing Multiple Evolution Metrics, in: Proceedings of the 2005 ACM Symposium on Software Visualization, SoftVis '05. ACM, New York, NY, USA, pp. 67–75. doi:10.1145/1056018.1056027

Pinzger, M., Nagappan, N., Murphy, B., 2008. Can Developer-module Networks Predict Failures?, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, SIGSOFT '08/FSE-16. ACM, New York, NY, USA, pp. 2–12. doi:10.1145/1453101.1453105

Poshyvanyk, D., Gueheneuc, Y.G., Marcus, A., Antoniol, G., Rajlich, V., 2007. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. IEEE Trans. Softw. Eng. 33, 420–432. doi:10.1109/TSE.2007.1016

Poshyvanyk, D., Marcus, A., 2007. Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code, in: 15th IEEE International Conference on Program Comprehension, 2007. ICPC '07. Presented at the 15th IEEE International Conference on Program Comprehension, 2007. ICPC '07, pp. 37–48. doi:10.1109/ICPC.2007.13

Potanin, A., Noble, J., Frean, M., Biddle, R., 2005. Scale-free Geometry in OO Programs. Commun ACM 48, 99–103. doi:10.1145/1060710.1060716

Presto, 2014. . Presto Distrib. SQL Query Engine Big Data. URL http://prestodb.io/

Raja, U., Tretter, M.J., 2012. Defining and Evaluating a Measure of Open Source Project Survivability. IEEE Trans. Softw. Eng. 38, 163–174. doi:10.1109/TSE.2011.39

Revelle, M., Gethers, M., Poshyvanyk, D., 2011. Using structural and textual information to capture feature coupling in object-oriented software. Empir. Softw. Eng. 16, 773–811. doi:10.1007/s10664-011-9159-7

Riel, A.J., 1996. Object-Oriented Design Heuristics, 1 edition. ed. Addison-Wesley Professional, Reading, Mass.

Riel, A.J., 1996. Object-oriented design heuristics. Addison-Wesley Pub. Co., Reading, Mass.

Robillard, M.P., Murphy, G.C., 2007. Representing Concerns in Source Code. ACM Trans Softw Eng Methodol 16. doi:10.1145/1189748.1189751

Shang, W., Adams, B., Hassan, A.E., 2010. An Experience Report on Scaling Tools for Mining Software Repositories Using MapReduce, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE '10. ACM, New York, NY, USA, pp. 275–284. doi:10.1145/1858996.1859050

Sharp, A., 1997. Smalltalk by Example: The Developer's Guide. Mcgraw-Hill, New York.

Shibata, K., Rinsaka, K., Dohi, T., Okamura, H., 2007. Quantifying Software Maintainability Based on a Fault-Detection/Correction Model, in: 13th Pacific Rim International Symposium on Dependable Computing, 2007. PRDC 2007. Presented at the 13th Pacific Rim International Symposium on Dependable Computing, 2007. PRDC 2007, pp. 35–42. doi:10.1109/PRDC.2007.46

Simpson, G., 1960. Notes on the measurement of faunal resemblance 258, 300–311.

Singh, K., 2007. Quantitative Social Research Methods. SAGE Publications Pvt. Ltd, Los Angeles.

SNAP: Stanford Network Analysis Project [WWW Document], 2014. URL http://snap.stanford.edu/ (accessed 4.11.16).

SonarQube™ - Open source platform to manage code quality [WWW Document], 2014. URL http://www.sonarqube.org/ (accessed 6.23.14).

Stecklein, J.M., Dabney, J., Dick, B., Haskins, B., Lovell, R., Moroney, G., 2004. Error Cost Escalation Through the Project Life Cycle. Presented at the 14th Annual International Symposium, Toulouse, France.

Sun, S., Xia, C., Chen, Z., Sun, J., Wang, L., 2009. On Structural Properties of Large-Scale Software Systems: From the Perspective of Complex Networks, in: Sixth International Conference on Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09. Presented at the Sixth International Conference on Fuzzy Systems and Knowledge Discovery, 2009. FSKD '09, pp. 309–313. doi:10.1109/FSKD.2009.635

Swanson, E.B., 1976. The Dimensions of Maintenance, in: Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 492–497.

Taube-Schock, C., Walker, R.J., Witten, I.H., 2011. Can We Avoid High Coupling?, in: Mezini, M. (Ed.), ECOOP 2011 – Object-Oriented Programming, Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 204–228.

Trifu, M., 2010. Tool-supported identification of functional concerns in object-oriented code.

Turnu, I., Concas, G., Marchesi, M., Pinna, S., Tonelli, R., 2011. A modified Yule process to model the evolution of some object-oriented system properties. Inf. Sci. 181, 883–902. doi:10.1016/j.ins.2010.10.022

Turnu, I., Concas, G., Marchesi, M., Tonelli, R., 2013. The fractal dimension of software networks as a global quality metric. Inf. Sci., Statistics with Imperfect Data 245, 290–303. doi:10.1016/j.ins.2013.05.014

UCINET Software [WWW Document], 2014. URL https://sites.google.com/site/ucinetsoftware/home (accessed 4.11.16).

USC-CSE, 1997. COCOMO II Model Definition Manual. Cent. Softw. Eng. Comput. Sci. Dep. South. Calif. Los Angel. CA.

Valverde, S., Sole, R.V., 2003a. Hierarchical Small Worlds in Software Architecture. ArXivcond-Mat0307278.

Valverde, S., Sole, R.V., 2003b. Hierarchical Small Worlds in Software Architecture. ArXivcond-Mat0307278.

van Koten, C., Gray, A.R., 2006. An application of Bayesian network for predicting object-oriented software maintainability. Inf. Softw. Technol. 48, 59–67. doi:10.1016/j.infsof.2005.03.002

Vasa, R., Lumpe, M., Branch, P., Nierstrasz, O., 2009. Comparative analysis of evolving software systems using the Gini coefficient, in: IEEE International Conference on Software Maintenance, 2009. ICSM 2009. Presented at the IEEE International Conference on Software Maintenance, 2009. ICSM 2009, pp. 179–188. doi:10.1109/ICSM.2009.5306322

Vasa, R., Schneider, J.-G., Nierstrasz, O., 2007. The Inevitable Stability of Software Change, in: IEEE International Conference on Software Maintenance, 2007. ICSM

2007. Presented at the IEEE International Conference on Software Maintenance, 2007. ICSM 2007, pp. 4–13. doi:10.1109/ICSM.2007.4362613

Vinay Kumar, K., Ravi, V., Carr, M., Raj Kiran, N., 2008. Software development cost estimation using wavelet neural networks. J. Syst. Softw. 81, 1853–1867. doi:10.1016/j.jss.2007.12.793

Wang, L., Wang, Z., Yang, C., Zhang, L., Ye, Q., 2009. Linux kernels as complex networks: A novel method to study evolution, in: IEEE International Conference on Software Maintenance, 2009. ICSM 2009. Presented at the IEEE International Conference on Software Maintenance, 2009. ICSM 2009, pp. 41–50. doi:10.1109/ICSM.2009.5306348

Watts, D.J., Strogatz, S.H., 1998. Collective dynamics of "small-world" networks. Nature 393, 440–442. doi:10.1038/30918

Weka, 2013. . Weka Data Min. Softw. Java. URL http://www.cs.waikato.ac.nz/ml/weka/

Wettel, R., 2009. Visual exploration of large-scale evolving software, in: 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. Presented at the 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009, pp. 391–394. doi:10.1109/ICSE-COMPANION.2009.5071029

Wettel, R., Lanza, M., Robbes, R., 2011. Software Systems As Cities: A Controlled Experiment, in: Proceedings of the 33rd International Conference on Software Engineering, ICSE '11. ACM, New York, NY, USA, pp. 551–560. doi:10.1145/1985793.1985868

Wheeldon, R., Counsell, S., 2003. Power law distributions in class relationships, in: Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003. Proceedings. Presented at the Third IEEE International Workshop on Source Code Analysis and Manipulation, 2003. Proceedings, pp. 45–54. doi:10.1109/SCAM.2003.1238030

Wilde, N., Scully, M.C., 1995. Software Reconnaissance: Mapping Program Features to Code. J. Softw. Maint. 7, 49–62. doi:10.1002/smr.4360070105

Wilkie, F.G., Kitchenham, B.A., 2000. Coupling measures and change ripples in C++ application software. J. Syst. Softw. 52, 157–164. doi:10.1016/S0164-1212(99)00142-9

Willinger, W., Alderson, D., Doyle, J.C., 2009. Mathematics and the Internet: A Source of Enormous Confusion and Great Potential. Not. Am. Math. Soc. 56, 586–599.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers, Norwell, MA, USA.

Wong, W.E., Gokhale, S.S., Horgan, J.R., 2000. Quantifying the closeness between program components and features. J. Syst. Softw., Special Issue on Software Maintenance 54, 87–98. doi:10.1016/S0164-1212(00)00029-7

Wong, W.E., Gokhale, S.S., Horgan, J.R., Trivedi, K.S., 1999. Locating program features using execution slices, in: 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings. Presented at the 1999 IEEE Symposium on Application-Specific Systems and Software Engineering and Technology, 1999. ASSET '99. Proceedings, pp. 194–203. doi:10.1109/ASSET.1999.756769

Yazdi, H.S., Mirbolouki, M., Pietsch, P., Kehrer, T., Kelter, U., 2014. Analysis and Prediction of Design Model Evolution Using Time Series, in: Iliadis, L.,

Papazoglou, M., Pohl, K. (Eds.), Advanced Information Systems Engineering Workshops, Lecture Notes in Business Information Processing. Springer International Publishing, pp. 1–15.

Ying, A.T., Murphy, G.C., Ng, R., Chu-Carroll, M.C., 2004. Predicting source code changes by mining change history. IEEE Trans. Softw. Eng. 30, 574–586. doi:10.1109/TSE.2004.52

Yu, P., Systa, T., Muller, H., 2002. Predicting fault-proneness using OO metrics. An industrial case study, in: Sixth European Conference on Software Maintenance and Reengineering, 2002. Proceedings. Presented at the Sixth European Conference on Software Maintenance and Reengineering, 2002. Proceedings, pp. 99–107. doi:10.1109/CSMR.2002.995794

Zanetti, M.S., Schweitzer, F., 2012. A Network Perspective on Software Modularity. ArXiv12013771 Nlin Physicsphysics.

Zhang, H., Gong, L., Versteeg, S., 2013. Predicting Bug-fixing Time: An Empirical Study of Commercial Software Projects, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13. IEEE Press, Piscataway, NJ, USA, pp. 1042–1051.

Zhao, W., Zhang, L., Liu, Y., Luo, J., Sun, J., 2003. Understanding how the requirements are implemented in source code, in: Software Engineering Conference, 2003. Tenth Asia-Pacific. Presented at the Software Engineering Conference, 2003. Tenth Asia-Pacific, pp. 68–77. doi:10.1109/APSEC.2003.1254359

Zhao, W., Zhang, L., Liu, Y., Sun, J., Yang, F., 2004. SNIAFL: towards a static non-interactive approach to feature location, in: 26th International Conference on Software Engineering, 2004. ICSE 2004. Proceedings. Presented at the 26th International Conference on Software Engineering, 2004. ICSE 2004. Proceedings, pp. 293–303. doi:10.1109/ICSE.2004.1317452

Zheng, X., Zeng, D., Li, H., Wang, F., 2008. Analyzing open-source software systems as complex networks. Phys. Stat. Mech. Its Appl. 387, 6190–6200. doi:10.1016/j.physa.2008.06.050

Zhou, Y., Leung, H., 2007. Predicting object-oriented software maintainability using multivariate adaptive regression splines. J. Syst. Softw., The Impact of Barry Boehm's Work on Software Engineering Education and Training 80, 1349–1361. doi:10.1016/j.jss.2006.10.049

Zimmermann, T., Nagappan, N., 2008. Predicting Defects Using Network Analysis on Dependency Graphs, in: Proceedings of the 30th International Conference on Software Engineering, ICSE '08. ACM, New York, NY, USA, pp. 531–540. doi:10.1145/1368088.1368161

Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting Defects for Eclipse, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, PROMISE '07. IEEE Computer Society, Washington, DC, USA, p. 9. doi:10.1109/PROMISE.2007.10

Zou, X., Settimi, R., Cleland-Huang, J., 2009. Improving automated requirements trace retrieval: a study of term-based enhancement methods. Empir. Softw. Eng. 15, 119–146. doi:10.1007/s10664-009-9114-z

Appendix

A1 - AST Parsing

An AST is a tree representation of the source code that maintains the total structure and logic of the system. An illustrative example will surely be helpful in understating the concept of ASTs. Figure a1.1 shows a method with the name `method` in the class `AstTest`.

```
0  import java.util.ArrayList;
1  public class AstTest {
2    public void method(int A[]){
3       int sum = 0;
4       ArrayList<String> list = new ArrayList<String>();
5       for(int i=0; i<A.length; i++){
6          sum += A[i];
7          list.add("Value of object number "+i+" is "+A[i]);
8       }
9    }
10 }
```

Figure a1.1 - Dummy code that is used in the explanation of Abstract Syntax Tree

In the following paragraphs, the transformation of the code in Figure a1.1 to an AST will be explained. The corresponding AST structure as it is in memory, along with number annotations that will be referenced in text, appears in Figure a1.2. This AST refers to a single java file that also constitutes a compilation unit (4). At the top left area, we notice the package that this compilation unit lies (1) and the array of imports that have been declared (2). The types declared in this compilation unit follow (3). The main object that contains the information of a type (class) is called Type Declaration (3.1) which is the most crucial and interesting part of our AST. The Type Declaration begins with the decomposition of line 1, of the source code, which is the line where the class is defined. It contains the modifiers list, the superclass types and the names of the implemented interfaces.

At the right column, the extended BODY_DECLARATIONS node is shown. We have added some annotations along with color lines for easier understanding of the code-to-tree mappings. Body declarations contain either Method Declarations, Field Declarations or other Type Declarations in the case of nested classes.

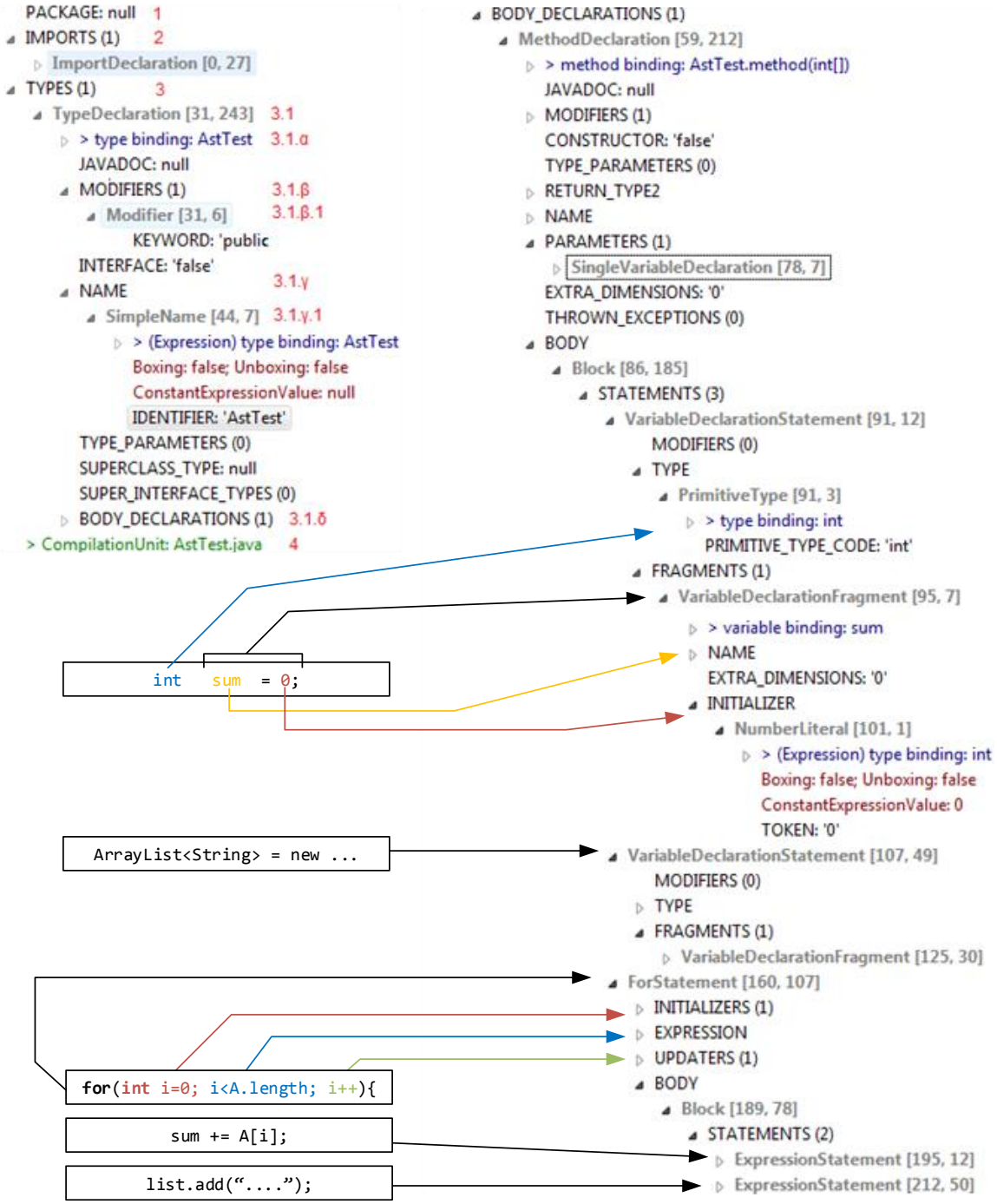


Figure a1.2 - The AST representation of the code of Figure a1.1 as it appears on Java Debugger of Eclipse along with some helpful code tags.

A2 - Seagle version order determination

The creation of a process for the handling of version ordering was considered mandatory by considering that the git tagging scheme may lead to situations where the evolution of the version names does not correspond to the evolution of the code base. A characteristic example is displayed in Figure a2.1, where a typical workflow is depicted.

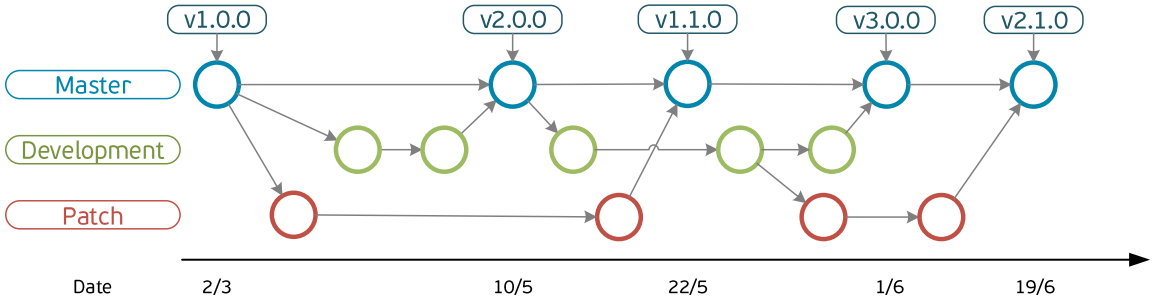


Figure a2.1 - A typical git workflow that may lead to disarranged version names

According to the example, version 1.0.0 is released at the 2nd of March, but after this, a patch branch is initiated. In parallel with the patch, the development team continues the work on the development branch. Just after version 2.0.0, the patch of the first version is ready for release, so it receives the tag 1.1.0. A second patch is needed for version 2.0.0, so this situation is repeated one more time and the patch 2.1.0 is released at June 19th, just after the launching of version 3.0.0.

A date-based ordering of the versions of this system will lead to an order like:

v1.0.0 → v2.0.0 → v1.1.0 → v3.0.0 → v2.1.0

which is clearly wrong. Instead, a version-number-based ordering should be applied along with the date ordering so that the final order becomes:

v1.0.0 → v1.1.0 → v2.0.0 → v2.1.0 → v3.0.0

This process, although in this example seems effortless, could become challenging considering the multitude of naming schemes that each project manager may use.

A3 – Concept Lattices - Full Versions

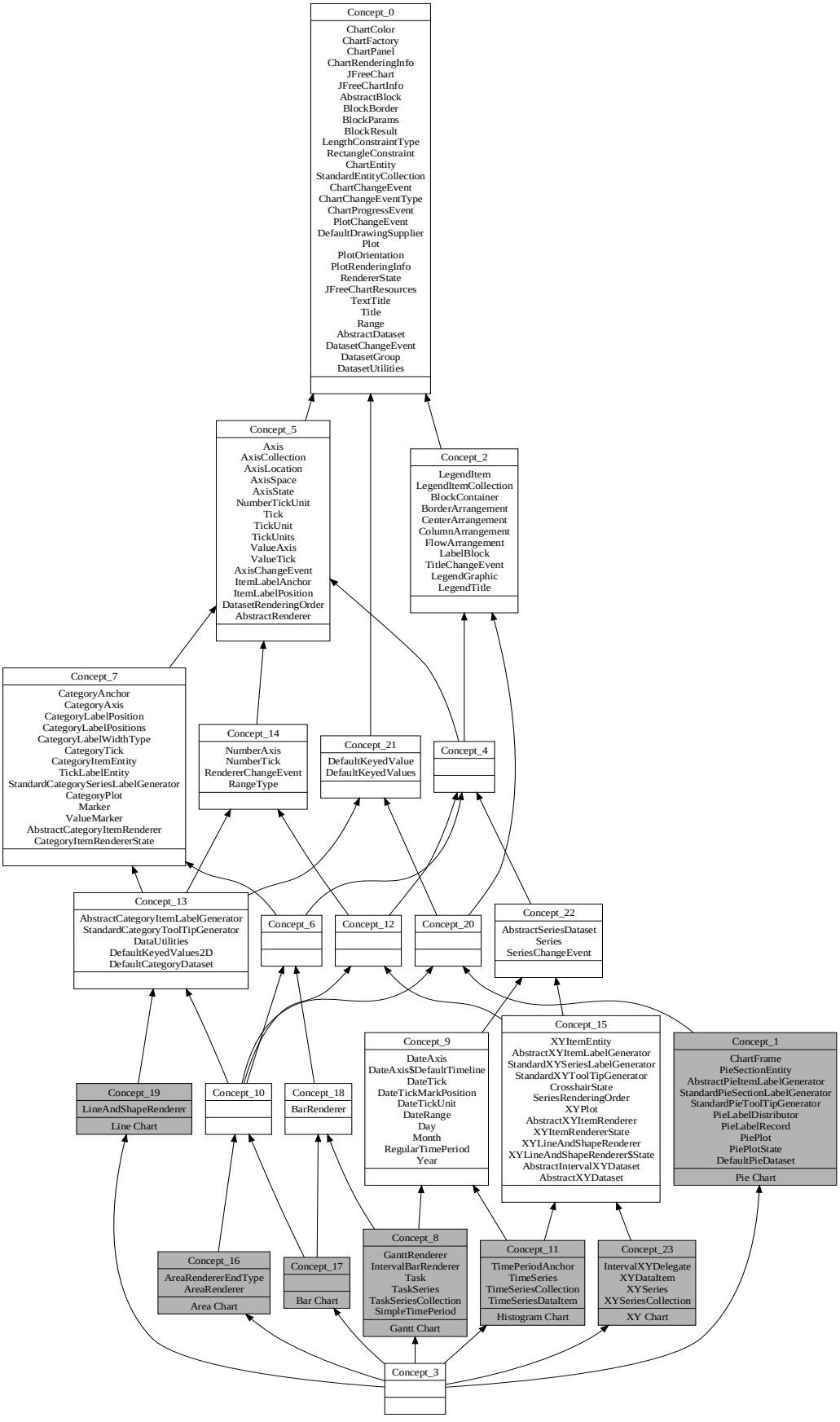


Figure a3.1 – Concept Lattice from JFreeChart ver. 1.0.0

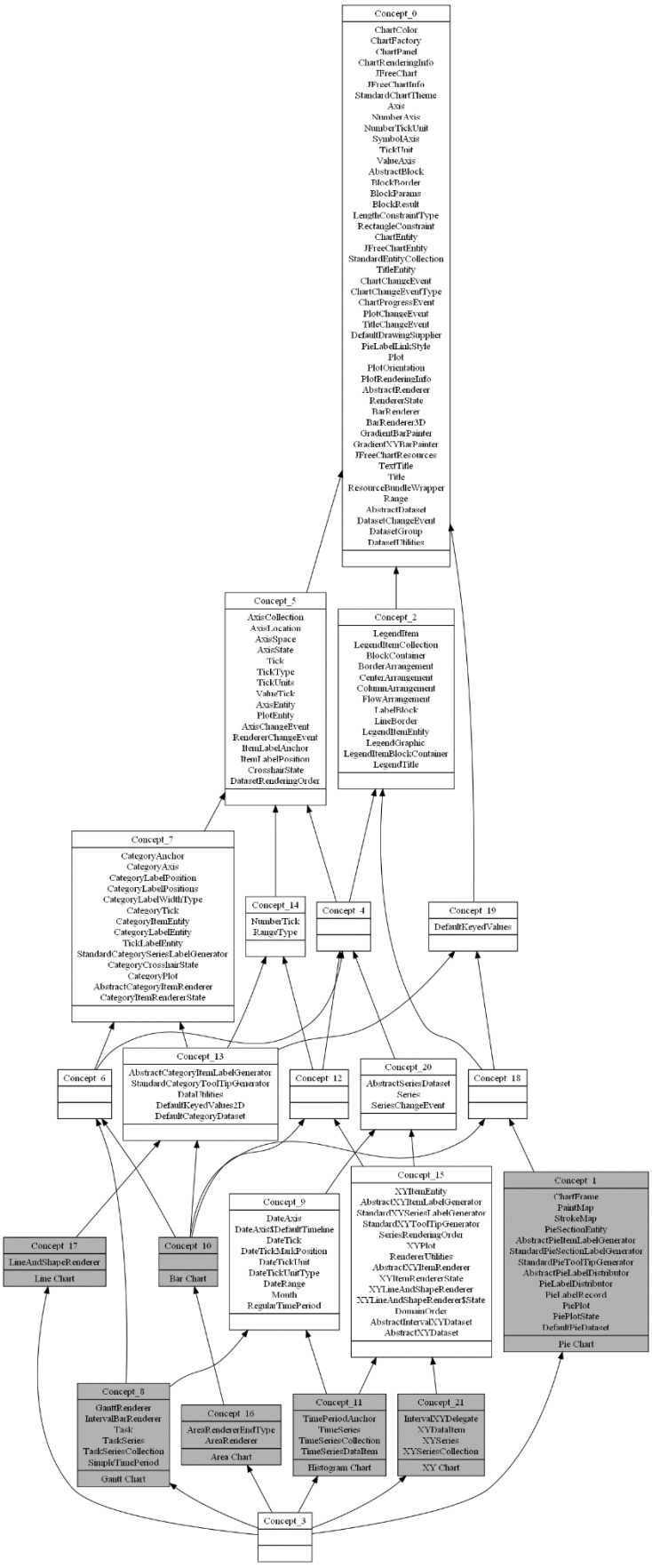


Figure a3.2 – Concept Lattice from JFreeChart ver. 1.0.13

A4 – Seagle Database schema