UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE

CENTRO DE CIÊNCIAS EXATAS E DA TERRA

PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

AN AUTOMATED APPROACH FOR PERFORMANCE DEVIATION ANALYSIS OF EVOLVING SOFTWARE SYSTEMS

FELIPE ALVES PEREIRA PINTO

Ph.D. THESIS

NATAL-RN

NOVEMBER, 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
CENTRO DE CIÊNCIAS EXATAS E DA TERRA
PROGRAMA DE PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO

FELIPE ALVES PEREIRA PINTO

An Automated Approach for Performance Deviation Analysis of Evolving Software Systems

Tese submetida ao Programa de Pós-Graduação em Sistemas e Computação, do Centro de Ciências Exatas e da Terra, da Universidade Federal do Rio Grande do Norte, como parte dos requisites para obtenção do título de Doutor em Ciência da Computação.

Advisor:

Prof. Dr. Uirá Kulesza

NATAL-RN

NOVEMBER, 2015

Catalogação da Publicação na Fonte
Universidade Federal do Rio Grande do Norte - UFRN
Sistema de Bibliotecas - SISBI

Pinto, Felipe Alves Pereira.
An automated approach for performance deviation analysis of evolving software systems / Felipe Alves Pereira Pinto. - Natal, 2015.
154f: il.

Orientador: Prof. Dr. Uirá Kulesza.

Tese (Doutorado) - Universidade Federal do Rio Grande do Norte. Centro de Ciências Exatas e da Terra. Programa de Pós-Graduação em Sistemas e Computação.

1. Evolução de software. 2. Atributos de qualidade. 3. Análise de variação de desempenho. 4. Análise dinâmica. 5. Mineração de repositório de software. 6. Modelo de regressão. 7. Software evolution. 8. Quality attributes. 9. Performance deviation analysis. 10. Dynamic analysis. 11. Software repository mining. 12. Regression model. I. Kulesza, Uirá. II. Título.

RN/UF/CCET CDU 004.416.6

FELIPE ALVES PEREIRA PINTO

An Automated Approach for Performance Deviation Analysis of Evolving Software Systems

Esta Tese foi julgada adequada para a obtenção do título de doutor em Ciência da Computação e aprovado em sua forma final pelo Programa de Pós-Graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte.

Prof. Dr. Uirá Kulesza – UFRN
Orientador

Profª. Drª. Elizabeth Ferreira Gouvêa Goldbarg – UFRN
Vice-coordenadora do Programa

Banca Examinadora

Prof. Dr. Uirá Kulesza – UFRN
Presidente

Prof. Dr. Nélio Alessandro Azevedo Cacho – UFRN
Examinador

Profª. Drª. Thais Vasconcelos Batista – UFRN
Examinador

Prof. Dr. Eduardo Santana de Almeida – UFBA
Examinador

Prof. Dr. Marcelo de Almeida Maia – UFU
Examinador

Novembro, 2015

ABSTRACT

The maintenance and evolution of software systems have become a critical task over the last years due to the diversity and high demand of features, devices and users. The ability to understand and analyze how newly introduced changes impact the quality attributes of the architecture of those software systems is an essential prerequisite for avoiding the deterioration of the engineering quality of them during their evolution. This thesis proposes an automated approach for the deviation analysis of the quality attribute of performance in terms of execution time (response time). It is implemented by a framework that adopts dynamic analysis and mining software repository techniques to provide an automated way to revel potential sources – commits and issues – of performance deviation in scenarios of an evolving software system. The approach defines four phases: (i) preparation – choosing the scenarios and preparing the target releases; (ii) dynamic analysis – determining the performance of scenarios and methods by calculating their execution time; (iii) deviation analysis – processing and comparing the results of the dynamic analysis for different releases; and (iv) repository mining – identifying development issues and commits associated with performance deviation. Empirical studies have been developed to assess the approach from different perspectives. An exploratory study analyzed the feasibility of applying the approach in systems from different domains to automatically identifying source code assets with performance deviation and the changes that have affected them during an evolution. This study was performed using three systems: (i) SIGAA – a web academic management system; (ii) ArgoUML – an UML modeling tool; and (iii) Netty – a network application framework. Another study has performed an evolutionary analysis of applying the approach to multiple releases of Netty, and the web frameworks Wicket and Jetty. It has analyzed twenty-one releases (seven releases of each system) and addressed a total of 57 scenarios. Overall, we have found 14 scenarios with significant performance deviation for Netty, 13 for Wicket, and 9 for Jetty. In addition, the feedback obtained from an online survey with eight developers of Netty, Wicket and Jetty is also discussed. Finally, in our last study, we built a performance regression model in order to indicate the properties of code changes that are more likely to cause performance degradation. We mined a total of 997 commits, of which 103 were retrieved from degraded code assets, 19 from optimized, while 875 had no impact on execution time. Number of days before release and day of week were the most relevant variables of commits that cause performance degradation in our model. The receiver operating characteristic (ROC) area of our regression model is 60%, which means that deciding if a commit will cause performance degradation or not by using the model is 10% better than a random guess.

Keywords: software evolution, quality attributes, performance deviation analysis, dynamic analysis, software repository mining, regression model.

RESUMO

A manutenção e evolução de sistemas de software tornou-se uma tarefa bastante crítica ao longo dos últimos anos devido à diversidade e alta demanda de funcionalidades, dispositivos e usuários. Entender e analisar como novas mudanças impactam os atributos de qualidade da arquitetura de tais sistemas é um pré-requisito essencial para evitar a deterioração de sua qualidade durante sua evolução. Esta tese propõe uma abordagem automatizada para a análise de variação do atributo de qualidade de desempenho em termos de tempo de execução (tempo de resposta). Ela é implementada por um *framework* que adota técnicas de análise dinâmica e mineração de repositório de software para fornecer uma forma automatizada de revelar fontes potenciais – *commits* e *issues* – de variação de desempenho em cenários durante a evolução de sistemas de software. A abordagem define quatro fases: (i) preparação – escolher os cenários e preparar os *releases* alvos; (ii) análise dinâmica – determinar o desempenho de cenários e métodos calculando seus tempos de execução; (iii) análise de variação – processar e comparar os resultados da análise dinâmica para *releases* diferentes; e (iv) mineração de repositório – identificar *issues* e *commits* associados com a variação de desempenho detectada. Estudos empíricos foram realizados para avaliar a abordagem de diferentes perspectivas. Um estudo exploratório analisou a viabilidade de se aplicar a abordagem em sistemas de diferentes domínios para identificar automaticamente elementos de código fonte com variação de desempenho e as mudanças que afetaram tais elementos durante uma evolução. Esse estudo analisou três sistemas: (i) SIGAA – um sistema web para gerência acadêmica; (ii) ArgoUML – uma ferramenta de modelagem UML; e (iii) Netty – um *framework* para aplicações de rede. Outro estudo realizou uma análise evolucionária ao aplicar a abordagem em múltiplos releases do Netty, e dos *frameworks* web Wicket e Jetty. Nesse estudo foram analisados 21 *releases* (sete de cada sistema), totalizando 57 cenários. Em resumo, foram encontrados 14 cenários com variação significante de desempenho para Netty, 13 para Wicket e 9 para Jetty. Adicionalmente, foi obtido *feedback* de oito desenvolvedores desses sistemas através de um formulário online. Finalmente, no último estudo, um modelo de regressão para desempenho foi desenvolvido visando indicar propriedades de *commits* que são mais prováveis a causar degradação de desempenho. No geral, 997 *commits* foram minerados, sendo 103 recuperados de elementos de código fonte degradados e 19 de otimizados, enquanto 875 não tiveram impacto no tempo de execução. O número de dias antes de disponibilizar o release e o dia da semana se mostraram como as variáveis mais relevantes dos *commits* que degradam desempenho no nosso modelo. A área de característica de operação do receptor (ROC – *Receiver Operating Characteristic*) do modelo de regressão é 60%, o que significa que usar o modelo para decidir se um *commit* causará degradação ou não é 10% melhor do que uma decisão aleatória.

Palavras-chave: evolução de software, atributos de qualidade, análise de variação de desempenho, análise dinâmica, mineração de repositório de software, modelo de regressão.

SUMMARY

FIGURE INDEX

TABLE INDEX

1. INTRODUCTION

Nowadays, software systems play a very important role in our society. Many people from different social levels are using software every day on computers and mobile devices. It means there are very complex software systems running in different platforms, such as computers, smartphones, tablets, smart TVs, cars and others. They also have a wide variety of goals, including entertainment software as media players and games, technical software as text editors, spreadsheet editors and computer aided design software. Some of them have quite critical requirements to quality attributes such as security, robustness, scalability and performance.

This chapter introduces this work. **Section 1.1** explains the context of this work and the main limitations of current approaches. **Section 1.2** presents the general research questions, while **Section 1.3** discusses general and specific goals. **Section 1.4** summarizes the proposed solution. The main contributions of this work is presented in **Section 1.5**. Finally, **Section 1.6** summarizes the thesis organization.

1.1. Problem Statement and Limitation of Current Approaches

Due to the diversity and high demand of functionalities, devices and users, the maintenance and evolution of software systems have become a critical and hard task over the last years. Developers are very often under pressure to deal with the complexity of managing and adapting software systems to achieve new customer requirements and technologies. When evolving software systems, developers have to maintain their quality attributes according to architectural and design decisions taken during the development process to avoid undesired side effects. The ability to understand and analyze how newly introduced changes impact the quality attributes of software systems when evolving them is an essential prerequisite for avoiding issues related to software erosion, which is the overall deterioration of the engineering quality of a software system during its evolution [1]. Modifications responsible for software erosion can violate software architecture principles [2] [3] and degrade the quality attributes of the system (e.g., performance), which happens when a new release of a software system exhibits inferior measurements of quality attributes compared to previous releases.

Well-known approaches for quality attribute analysis are the software architecture evaluation methods, such as SAAM (*Software Architecture Analysis Method*) and ATAM

(*Architecture Tradeoff Analysis Method*) [4] [8]. The architecture of a software system can be seen as result of a set of design decisions made about the system [9], which can affect many of its modules, including its structure and quality attributes. Many of these evaluation methods explore architectural analysis of quality attributes based on scenarios [4] [8] [10] [11] [12]. In this context, a scenario is a high-level action that represents the way in which the stakeholders expect the system to be used [4]. However, these methods only perform early evaluation, which happens before system implementation and involves manual reviews of the architectural decisions, scenarios and quality attributes [4]. Thus, such methods are not suitable to evaluate multiple evolutions of a system because it can involve multiple manual reviews. In addition to that, the analysis of quality attributes, such as performance, needs to be automated in order to promote the continuous evaluation and the discovering of code assets responsible for affecting such quality attribute during the system evolution.

Other architectural approaches with code analysis support focused mainly on the structural compliance of the systems [13] [14] [15] [16], but without support for quality attributes, such as performance. Some existing approaches use mathematical models for predicting quality attributes [17] [18], which represents an approximation of the real impact of the evolution, but they do not focus on source code evolution changes. In addition, a well-known way to measure properties related to the quality attribute of performance is using benchmarks, which usually consist on monitoring the global resources that a software uses to perform specific tasks. They are particularly useful in helping developers to determine the limits of the system through load tests [5]. Finally, some recent research studies are exploring software repository mining techniques to infer information about performance. For example, exploring how performance bugs are usually discovered, reported and fixed [19], and how repositories of performance regression causes could be used to identify causes of new regressions [20]. Despite the usage of repository mining techniques, these existing approaches do not indicate code assets or changes related to performance degradation.

Thus, there is still a gap to address, since none of the current approaches are able of indicating code changes that have caused degradation in particular quality attributes. This could be achieved through an automatic analysis process of the source code and the introduced changes during the target evolution. The system implementation is a fundamental artifact to consider when automating the performance deviation analysis during the evolution of software systems. A proper source code analysis together with available information from software repositories, such as control version and issue tracking systems, can be useful to help the

evaluation process including the identification of modules and classes that have suffered performance deviation inside particular scenarios. In addition to that, the integration of code analysis and mining software repository technique can also enable the discovering of changes in terms of issues and commits responsible for the performance deviation.

In this context, based on existing methods of software architecture evaluation and on the available information for different releases of a system, the main requirements that an approach for analysis of evolving systems have to achieve in order to identify performance deviations and their potential causes are highlighted below:

- It should automate many parts as possible of the process. Manual techniques might be time-consumption and expensive, and they are not suitable if the evaluation process requires the analysis of many successive evolutions;
- Software systems could be too large to an entire analysis. Thus, it should be able to focus on some selected parts of the system. It is possible to reuse the concept of target scenarios from architecture evaluation methods;
- It should be able of measuring performance of the target scenarios and their methods in order to identify where the deviation has happened.
- It should provide support for source code analysis in order to give detailed feedback about the code assets related to performance deviation;
- It should be able to access data from software repositories, such as control version and issue tracking systems, in order to retrieve changes (commits and issues) related to code assets with performance deviation.

1.2. General Research Questions

Based on the context and limitation of current approaches previously mentioned, four general research questions were proposed in order to guide this work. They are:

RQ1. *How feasible is a scenario-based approach for automating the performance deviation analysis in terms of execution time considering source code assets and changes (commits and issues)?* The absence of such approach and the limitations of current ones may be an indication that such idea is not feasible in practice because it implies several limitations, or even because such evaluation could not be automated. We believe that it is possible by integrating some techniques including static and dynamic code analysis, and software repository mining. In this context, static analysis refers to the extraction

of information from the source code without the need of running the system, while dynamic analysis refers to the extraction of the information from the system while it is running. This integration has achieved some interesting results with strong indication of its feasibility according to results we obtained from our studies presented in chapters 5 and 6.

RQ2. *Did the scenarios that exhibit performance deviation also changed in terms of implementation?* An important concern related with the proposed approach is that it could find degraded or optimized scenarios that had no source code modification. It would be an indicator that scenarios are being deviated because of external factors, for example, different libraries, settings from the runtime environment, or even any measurement problem. The study presented in Chapter 6 shows that most of the scenarios found with performance deviation also have changes related to source code assets responsible for such variation.

RQ3. *Can automated functional tests be used to exercise scenarios of interest in order to measure performance in terms of execution time?* The goal is to make the proposed approach easier to run in a large amount of systems. Functional tests are not designed to find performance regressions. However, many existing systems implement such kind of test. Thus, we have reused functional tests from the systems in order to exercise scenarios of interest and measure the performance in terms of execution time during the process. In this case, some tests should be excluded to ensure a fair comparison between different releases, for example, tests with random inputs. The main motivation for this research question is that many existing systems implement automated tests. Thus, we are concerned about the feasibility of using them for such purpose. Our concern about the tests was reinforced by some developers who answered an online survey presented in Chapter 6 and shared similar concern.

RQ4. *Can we accurately predict when commits may cause performance degradation based on their most influential properties?* In Chapter 7, we reused results from chapters 5 and 6 to build a performance regression model in order to indicate properties of code changes (commits) that are more likely to cause performance degradation. Examples of such properties are number of inserted lines, number of churns, author activity, day of week, and others. The Receiver Operating Characteristic (ROC) area was calculated to evaluate the accuracy of our regression model when used to predict if a commit could cause performance deviation or not. The found ROC area was 60%, which means that

deciding if a commit will cause performance degradation or not by using the model is 10% better than a random guess.

1.3. General and Specific Goals

In order to answer the proposed research questions, the main goal of this thesis is to define and implement an automated scenario-based approach for revealing performance deviations in terms of execution time. The approach takes into account new releases generated during the software evolution. It also proposes to indicate which code assets may cause performance variation based on the mining of commits and development issues responsible for changing them. Dynamic analysis and repository mining techniques are combined to achieve this goal, and we expect that our approach and support framework help developers in identifying ways to optimize the performance of their systems. In this context, the specific goals of this thesis are:

- The proposal of an approach that integrates static and dynamic analysis with software repository mining techniques to automate the performance deviation analysis in terms of execution time for evolving systems. In addition, the approach also provides initial support to quality attribute traceability in source code assets, as security, reliability and performance;
- A framework that implements the proposed approach developed using the Java programming language. It provides support to the performance deviation analysis of Java software systems;
- Empirical studies to evaluate the approach and the support framework when applied to real systems in different domains, for example, network communication framework, web framework, web information systems, and desktop systems;
- Preliminarily, analyzing and discussing some similarities and differences of the proposed approach compared to some related work.

1.4. Proposed Solution

This work presents a scenario-based approach for automating the performance deviation analysis of evolving software systems considering the property of execution time, which also supports traceability of quality attributes, including security, performance and reliability. In this context, performance refers to the responsiveness of the system, i.e., the time required to respond to events [4]. In order to measure performance, a set of different properties,

including memory, disk and CPU usage, can be used for benchmark-based [5] or power consumption [6] [7] approaches. Execution time was chosen here because it is a general and common property for the system responsiveness.

The approach application requires the availability of the source code of the system and relies on extra metadata information added to the source code. The metadata, represented in the approach implementation by Java annotations, is responsible for indicating entry points of relevant system scenarios, which are represented by the method that starts the scenario execution. The approach uses dynamic analysis to monitor the system execution and measure performance in terms of execution time. Static analysis allows identifying particular code assets that were changed by commits and issues. The data extracted from all these analysis provides enough information to discover: (i) system scenarios that have exhibited performance deviation; and (ii) the modified methods inside them responsible for the performance deviation between two releases of a system. Once the methods are known, it is possible to discover which commits and issues were responsible for the changes during the evolution through the mining of the control version and issue tracking systems.

The approach was implemented as an object-oriented framework using the Java programming language and address the evaluation of software systems developed in the Java language. It integrates static and dynamic analysis techniques with software repository mining techniques. The static analysis support was codified using WALA [22] and Eclipse JDT [23]. On the other hand, the dynamic analysis was implemented using the AspectJ language to provide code instrumentation in order to monitor the system execution. The framework is able to compare releases of a system and generates reports including information of the scenarios and their methods with performance deviation, the commits responsible for changing such methods, and the issues related to the commits. In addition, scenarios that have failed and thrown exceptions during the execution are also saved, and it is possible to discover the methods responsible for throwing exceptions and the paths they have followed. All information related to dynamic analysis monitoring are stored in a PostgreSQL [26] database.

As a consequence of such approach, when the implementation of the system evolves, developers can just keep or adjust the metadata that was added in the source code, and run a new performance deviation analysis to compare the new version with the previous one using any strategy to execute the selected scenarios, for example, running the test suite of the system. If the past version was already analyzed, there is no need of running the dynamic analysis again

because all its data was previously stored. The generated reports are particularly interesting to find unexpected deviations, since some deviations are expected and cannot be avoided, for example, when extending the scenarios to include new validations or functionalities. In such cases, the developers stay aware of the impact of such changes to the system scenarios and they can decide if the variation is acceptable or not. The primary use case of the approach is when it finds deviations which developers do not expect. This allows them to investigate the deviations further. In case of expected deviations, it can confirm which code assets were affected.

1.5. Main Contributions

The main contributions of this thesis are summarized below:

- The definition of a scenario-based approach for performance deviation analysis in terms of execution time during a system evolution, which integrates static and dynamic analysis, and software repository mining techniques;
- A framework that implements and automates such approach for systems developed with the Java programming language. It currently provides support for the mining of *Subversion* [24] and *Git* [25] version control systems, and *Issuezilla*, *IProject*, *Bugzilla*, *Github*, and *Jira* issue tracking systems;
- An exploratory study with three real systems from different domains, being an academic management system (SIGAA [27]), a network application framework (Netty [28]), and an UML modeling tool (ArgoUML [29]);
- An evolutionary empirical study which have analyzed multiple evolutions of Netty, and the web application frameworks Wicket [30] and Jetty [31];
- The analysis of the data from such studies to define a regression model with the aim of indicating which commit properties are more likely to lead to changes related to performance deviation.

1.6. Thesis Organization

This thesis is organized as follows:

- **Chapter 2** presents the background with essential concepts applied during the thesis development;
- **Chapter 3** presents the proposed approach organized in steps with a set of input and output artifacts;

- **Chapter 4** shows the proposed framework architecture and discusses how each step of the proposed approach was implemented;
- **Chapter 5** presents and discusses the results of an exploratory study with SIGAA, Netty and ArgoUML. The main goal in this case was to assessing the approach and framework feasibility to analyze software systems from different domains. Results from this study has provided initial feedback for RQ1 and RQ3;
- **Chapter 6** presents and discusses the results of an evolutionary case study with Netty, Wicket and Jetty. Seven releases of each system were analyzed. The goal was to assess the capacity of the proposal to identify performance issues over multiple evolutions of existing systems and to check if the developers of these systems were aware of that. Results from this study together with the previous one has provided enough feedback to answer RQ2 and complement the answers of RQ1 and RQ3;
- **Chapter 7** shows how the data collected from the previous studies can be used to define a regression model with the aim of suggesting which commit properties are more likely to lead to changes related to performance deviations. Limitations and threats for such model are also discussed. The results from this study were used to answer the RQ4;
- **Chapter 8** discusses some related work, presenting limitations, differences and similarities with the proposed approach;
- **Chapter 9** presents the conclusion of the work with a review of the main results and the general research questions. It also discusses limitations of the proposed approach. In addition, it also points out to future work.

2. BACKGROUND

This chapter revisits some essential concepts and techniques in order to provide a better understanding of this work, as well as some relevant research in the same context. **Section 2.1** presents some concepts related to software architecture and quality attributes, including the scenario definition (**Subsection 2.1.1**), early and later evaluation process (**Subsection 2.1.2**), risks, sensitivity and tradeoffs points (**Subsection 2.1.3**), and how the proposed work is related to each one of these concepts (**Subsection 2.1.4**). **Section 2.2** gives an overview of static and dynamic code analysis. Mining software repositories concepts are shown in **Section 2.3**, followed by a discussion of how the approach is related to these concepts (**Subsection 2.3.1**). Finally, **Section 2.4** introduces a classification for approaches that deal with software evolution and erosion, and it shows how this work could be categorized in such classification (**Subsection 2.4.1**).

2.1. Architectural Concepts and Evaluation of Quality Attributes

Approaches for scenario-based evaluation of quality attributes incorporate concepts common to software architecture evaluation methods, which are vastly reused in this work. This section introduces the concepts directly related to such methods. In this context, a traditional definition says that “the software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them” [32]. Additionally, according to Taylor et al. [9], a software architecture embed the main design decisions taken about the system by the architects.

Which components belong to the software architecture, which services or even properties of such components will be externally visible, and how they are related among them are all resulting of design decisions that should be taken before the system implementation. Ideally, such design decisions and the system implementation, which concretize the software architecture, should keep compliance during all life-cycle of the system. As important as that, is to realize that the software architecture is fundamental for the system in order to achieve its requirements related to quality attributes [4], including security, reliability, performance, and others. It happens because design decisions may impact quality attributes. For example, deciding about parallelizing current operations and increasing the number of bits used to

encrypt messages over the Internet may directly impact performance and security. It could be analyzed with a software architecture evaluation method, such as ATAM [4] [33].

The quality attributes that will be analyzed in an evaluation process depend on the context and domain of the system. A common strategy is addressing the most critical quality attributes for the target system. Different methods of software architecture evaluation focus on different attributes. For example, SAAM [4] [11] focused on modifiability, while ATAM allows architects to define the desired quality attributes in the initial steps of the process. Some common quality attributes are defined below, from Clements et al. [4] originally based on Bass et al. [32], when talking about the quality attributes that can be evaluated:

- **Performance** refers to the responsiveness of the system, the time required to respond to events or the number of events processed in some interval of time;
- **Reliability** is the ability of the system to keep operating over time. It is usually measured by mean time to failure;
- **Security** is a measure of the system's ability to resist to unauthorized attempts at usage and denial of service while still providing its services to legitimate users;
- **Portability** is the ability of the system to run under different computing environments;

Many other quality attributes are defined by Clements et al. [4] and Bass et al. [32]. However, our approach and implementation have focused mainly on the quality attribute of performance, in terms of execution time. Scenarios are fundamental for the evaluation of quality attributes. The goal is to exercise the scenarios in order to determine if the architecture is suitable for a set of quality attributes. It means that a software architecture evaluation process will not tell us if the architecture is good or not, or if performance was achieved or not, but it will tell us where potential risks are.

A good example is when we want to evaluate how easy is to modify a particular feature of a system. For example, a network communication system could dynamically change protocols by reading a configuration file at runtime. On the other hand, it needs to be recompiled in order to change its encryption algorithm. In this context, can we say that the communication mechanism of this system is modifiable? If we consider the possibility of changing the protocol, the answer is yes, but it might be no in case of the encryption algorithm. However, whether the communication mechanism of this software is modifiable in the way that is important for the

stakeholders, and dynamically changing the protocol is enough, while changing the encryption algorithm does not matter, we can say it is suitable for this quality goal.

2.1.1. Scenario-based Evaluation

One of the first phases of a software architecture evaluation is to define what points of the system will be the target of the evaluation process. A common strategy, which was also reused by this work, is to use scenarios from the system. Several evaluation methods have adopted this concept in order to guide the process for years [4] [8] [10] [11] [12] [33].

A scenario is defined by the interaction between stakeholders and the system. They are particularly useful to help the architects to understand how the quality attributes could be addressed. Stakeholders may have different perspectives of the scenarios. For example, a user could imagine a scenario as a task that she/he needs to do in the system. On the other hand, a developer, who is going to implement the scenario, may focus on an architectural view, and use the software architecture to guide the development process.

Thus, scenarios are truly useful when the architects and developers need to get a better understanding about quality attributes since they specify all types of operations the system will have to execute to meet particular functionalities. In this context, a detailed analysis of the way how such operations will be implemented, executed, or even if they can fail or not, helps evaluators to extract important information about many quality attributes, for example, performance. According to Clements et al. [4], when describing ATAM, three kinds of scenarios could be addressed in the evaluation process: use case scenario, growth scenarios, and exploratory scenario.

Use case scenarios describe the user's intended interaction with the completed running system. An example with performance requirement is: remote user requests a database report via the Web during peak period and receives it within five seconds. Growth and exploratory scenarios are related to changes, which represent future expected or possible changes. Thus, growth scenarios represent typical anticipated changes to a system in moderate way, for example, add a new message type for a system, migrate the system for new releases of operational systems, or to another one in the same family. On the other hand, exploratory scenarios have the goal of stressing and expose the limits of the current design of the system, for example, migrate the system from Windows to UNIX, add new critical performance requirements, and others. Due to the generic use of metadata and annotations, as we will present

in Chapters 3 and 4, the proposed approach has no limitations related to the kind of target scenarios.

2.1.2. Early and Late Evaluation

It is another concept taken from software architecture evaluation methods. The classical application of them occurs when the architecture has been specified but before implementation has begun [4]. However, the evaluation for quality attributes, such as performance, should be performed in a continuous way during all the system's life-cycle, even for evolutions. Considering this, it is possible to highlight two moments to conduct the evaluation process: early and late.

According to Clements et al. [4], in the early process, the evaluation does not need wait the architecture be fully specified. It can be used at any stage during the architecture creation process to examine the design decisions that were already made, or even to decide about those ones pending. On the other hand, the late process takes place when not only the architecture is finished but the implementation is completed as well. This case occurs, for example, when an organization inherits some sort of legacy system. In this case, an evaluation will help the new owners to understand the legacy system and if it meets its quality attribute requirements.

The perspective of this work defends that the evaluation process for quality attributes, such as performance, should be performed in a continuous way, starting up with an early approach, for example, a method for software architecture evaluation, while the system is not ready, and following up with a late approach, for example, our proposal, when system's implementation is available. This strategy allows the evaluation of the system at any stage of its life cycle, even when the system's implementation evolves. Of course, early evaluation approaches could still be used if there is the need of adding new design decision or change current ones. After that, a late evaluation approach could be used to validate the source code changes, check structural compliance, and ensure that the quality attributes are suitable.

2.1.3. Risks, Sensitivity Points and Tradeoff Points

An evaluation is not capable of saying if the software architecture or the implementation is good or not, but they can provide information about where the main risks are. Classical applications of methods for evaluation of quality attributes produce reports as output of the process with such information. Some methods, for example, ATAM for software

architecture evaluation is also able of producing information related to sensitivity points and tradeoff points.

In the context of these methods [4] [33], risks are potentially problematic architectural decisions, which usually represent decisions that impact important quality attributes for the correct working of the system. Some examples include, choosing schedule algorithms, using a relational or object-oriented database, or how many bits use to encrypt messages over the Internet. Risks can also represent situations where the decision was already made, but the consequences are not fully understood yet, for example, whether the development team will include a new layer to improve portability, but they do not know for sure how functionalities will be included or how it could impact the general performance of the system.

Sensitivity points [4] [33] are design decisions involving one or more components that are critical for achieving particular quality attributes. They receive this name because the quality attribute is sensitive to changing the decision. For example, the level of confidentiality in a virtual private network might be sensitive to the number of bits of encryption, impacting directly the security of the system. Sensitivity points represent very important parts of the system where developers and architects should pay particular attention in order to understand how design decisions impact the behavior of the quality attributes, and how such decisions could even block the achievement of some quality goals.

Finally, a tradeoff point [4] [33] is an architectural decision that affects more than one attribute and is a sensitivity point for more than one attribute, in other words, it is a decision that is a sensitivity point for more than one quality attribute at the same time. Tradeoff points usually affect the quality attributes in opposite ways. For example, reusing the previous example, changing the level of encryption could have a significant impact on both security and performance. Increasing the level of encryption improves the security, but requires more processing time to prepare the message. If the time to processing a message has a hard real-time latency requirement, the level of encryption could be a tradeoff point for security and performance. Tradeoff points are one of the most critical decisions and need to be carefully addressed.

Finding out where the risks are by analyzing how sensitivity points and tradeoff points can impact the desired quality attributes is fundamental to a successful evaluation. For example, a useful information in this case is to identify which are the potential scenarios that contain

tradeoff points, or which code assets or components from these scenarios, considering a late evaluation process, were responsible for inferior measurements of particular quality attributes after an evolution.

2.1.4. Architectural Concepts and the Proposed Approach

This subsection justifies why the presented architectural concepts in this chapter are important for the proposed approach. Such discussion is necessary because the proposed evaluation process have many features that are inspired on software architecture evaluation methods, even that our approach is mainly focused on the implementation of the system, and not in architectural components or relationships among them. This work focuses in late evaluation. We are going to answers the follow conceptual questions [4] in this section related to the proposed approach: **(Q1)** What can the approach evaluate? **(Q2)** When can the approach be applied? **(Q3)** What are the outputs of the evaluation process?

The approach can evaluate particular quality attributes from selected target scenarios **(Q1)**. First, about the target quality attributes, as said before on Chapter 1, the approach is focused on the quality attribute of performance measured in terms of execution time. Other quality attributes are also supported, such as security and reliability, but they are not measured and are only used for documentation purposes although we can infer some simple information from them by using source code annotations, static code analysis and traceability. In this case, for documentation purposes, the approach is not dependent on particular quality attributes and provide extension mechanisms to include new attributes as will be shown in Chapters 3 and 4.

Second, related to the types of scenarios: use case scenarios, growth scenarios, and exploratory scenarios (subsection 2.1.1). The approach is scenario-based, it means that the evaluation process is guided by them. However, it is not dependent on the type of scenario, despite we have used general use case scenarios exercised by JUnit tests in our studies. The developers, for example, could select which parts of the system will be analyzed by marking entry points of scenarios in the source code. Annotations are used for this.

About the second question **(Q2)**, there are two possibilities: early and late evaluation (subsection 2.1.2). The approach requires the availability of the source code of the system. Thus, it ideally should be applied after the implementation is complete what characterize a late evaluation approach. It is also possible to apply the proposed evaluation approach during the development process to analyze parts of the implementation, but the main purpose is to

automate the performance evaluation in terms of execution time for particular evolutions of a system.

About risks, sensitivity points and tradeoff points (subsection 2.1.3), the framework can report potential risks in scenarios with sensitivity and tradeoff points, as well as the potential changes responsible for performance deviation in terms of execution time **(Q3)**. The framework that implements the proposed approach has support to static code analysis, what allow developers to annotate methods that implement, or are involved, in particular design decisions by identifying the quality attributes that they could affect in the source code. Based on this information, the framework is able to identify potential risks in scenarios with sensitivity and tradeoff points. In addition, the dynamic code analysis is used to identify methods of the scenarios that had significant variation during an evolution in order to perform automated performance deviation analysis in terms of execution time, what is important to discover the potential changes responsible for such variation.

2.2. Dynamic and Static Analysis

The proposed approach uses static and dynamic analysis to assembly call graphs used during the evaluation process. Static analysis refers to the extraction of information directly from the source code of the system or from its binary files. In this kind of analysis, there is no need to run the system, but the availability of the source code or binary files is required. On the other hand, dynamic analysis is used to extract information from the system while it is running by monitoring and intercepting calls to routines (instrumentation/profiling process). Both strategies can be used to assembly call graphs of systems.

According to Graham et al. [34], in the assembled call graph, the nodes represent routines and directed arcs represent calls, and is also possible to distinguish among three different call graphs for a program, which the definition is adapted below:

- The **static call graph** includes all routines and all possible arcs. It usually keeps only one node to represent each declared routine. This kind of call graph can be assembled by using static analysis;
- The **dynamic call graph** includes only those routines and arcs traversed by the profiled execution of the program. It might keep more than one node to represent a routine declaration in order to difference executions, since the same routine may run many times. This kind of call graph can be assembled by using dynamic analysis;

- The **complete call graph** incorporates all routines and all potential arcs. This kind of call graph contains the other two graphs as subgraphs.

Assembling static call graphs for software implemented with object-oriented programming languages [35] [36] has several limitations that may result in substantial loss of accuracy to represent the way the system will be used, since the static analysis process cannot discover all method executions that will happen at runtime, and not every captured method will be truly executed [37]. In this context, the support of object-oriented languages to concepts as interfaces, inheritance and polymorphism difficult this kind of analysis. In addition, the system's implementation may contain reflective calls to methods, calls configured with XML (eXtensible Markup Language) or another strategy usually used by framework and middleware, what makes it even more difficult.

On the other hand, dynamic analysis does not have such limitations. Call graphs assembled with this strategy can represent the execution path in the exact way it happened. In general, most of the approaches that use dynamic analysis use progressive assembling of the call graph through the instrumentation of the system, intercepting relevant calls to be stored in the call graph. It is common these approaches make use of aspect-oriented programming [37] [38]. Even that every execution is accurately represented, a known limitations of dynamic call graphs include the dependency of the graph to the set of execution profiled. It means that if some very important part of the system was not executed, it will not be in the graph. Another issue is that instrument the system may introduce some interference to the original execution, for example, increasing the latency [39].

In order to illustrate such differences and limitations between a static and a dynamic call graph, Figure 1 shows three different call graphs extracted from the illustrative source code sample in Figure 2. Imagine that the `Root()` method is responsible for starting the execution of a scenario of interest which we want to evaluate. In this case, the call graphs for such scenario – from this method – have to be generated.

We can observe that the static call graph (Figure 1A) represents every possible call for each method. In this case, the *Root* node will be the `Root()` method and it could call methods `A()`, `B()` and `C()`. Methods `B()` and `C()` call `D()` and `E()`, respectively. On the other hand the dynamic call graphs (Figure 1B and Figure 1C) represent the exact way the system was executed. For example, considering $n = 0$, in the `Root()` method, the `C()` method will not be

executed inside the *for-loop*, while for $n = 2$, the `C()` method will be executed twice inside the *for-loop*. It is a good example to understand some differences between a call graph generated with static and dynamic analysis. In the first case, we usually represent every possible call and each method has a single node representation. In the second case, each node represents a different execution of the method and the call graph has a dependency of the way the system was executed. These reasons explain why the dynamic call graphs in Figure 1B and Figure 1C are different and why the `C()` method execution generates two *C* nodes for $n = 2$.

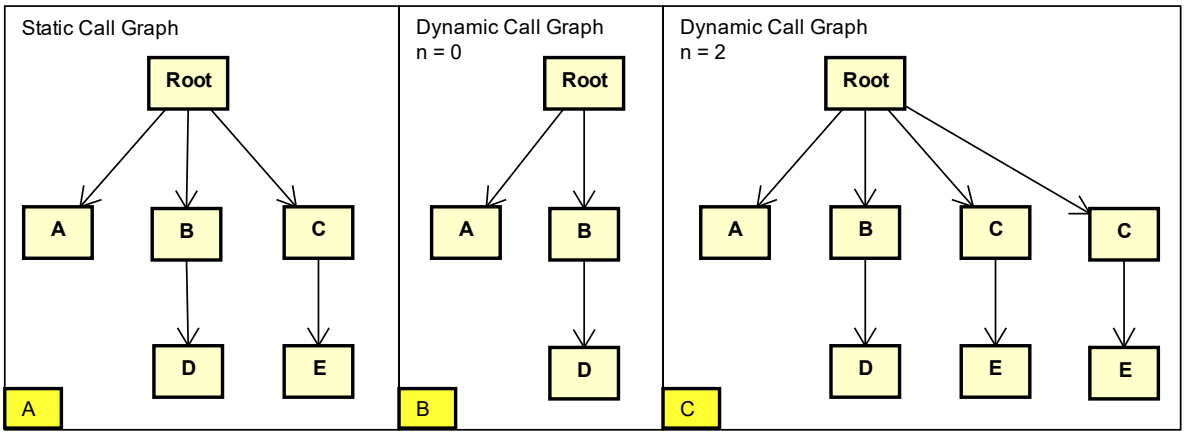


Figure 1. Illustrative call graph example.

```
void Root(int n) {
      A();
      B();

      for (int i = 0; i < n; i++)
            C();
}

void A() {  }
void B() { D(); }
void C() { E(); }
void D() {  }
void E() {  }
```

Figure 2. Illustrative example of method calls.

The framework that implements the proposed approach provides support for both static and dynamic call graphs. They are respectively assembled using static analysis with WALA [22], and dynamic analysis with an aspect-oriented strategy for instrumentation implemented with AspectJ language programming. The static call graph is used to trace code assets of interest for particular documented quality attributes, while the dynamic call graph is used to determine the exact way scenarios are executed and to measure their performance in terms of execution time. Details of these processes will be presented in Chapter 3 and 4.

2.3. Mining Software Repositories

Mining Software Repositories (MSR) has been proposed and explored over the last years in various domains to build predictive models, pattern identification tools and techniques to mine large repositories of data. In this context, the software development and maintenance process generates a very huge amount of data [40]. Chaturvedi et al. [40] present a classification for tools used in mining tasks, including newly developed tools, traditional data mining tools, prototypes, and scripts. According to them, these tools are used mainly for data extraction, required data filtering, pattern finding, learning and prediction. Most of the researchers use already developed tools for data mining tasks in software repositories, while only a few of them, which need customized data mining, develop their own tools or scripts.

The terms mining software repositories, repository mining, or simply data mining have been applied to describe a broad class of approaches that analyze software repositories with different purposes. Here, all of these terms are used as synonyms. Thus, software repositories refer to artifacts that are produced and archived during software evolution, including the information stored in source code version control systems (e.g., *Git*, *Subversion*), issue tracking systems (e.g., *Bugzilla*, *Jira*, and others), and communication archives (e.g., *e-mail*) [41].

According to Hassan [42], mining software repository analyzes the rich data available in software repositories to uncover interesting and actionable information about software systems and projects. In this context, he considers as example of types of repositories:

- **Historical repositories** such as source control repositories, bug repositories and archived communications. They record several information about the evolution and the progress of a project;
- **Runtime repositories** such as deployment logs, which contain information about the execution and the usage of an application at a single or multiple deployment sites;
- **Code repositories** such as *Sourceforge.net* and *Google code*, which contain the source code of various applications developed by several developers.

Kagdi et al. [41] present a survey of mining approaches that examine multiple versions of software artifacts or other temporal information in the context of software evolution. They also derived a taxonomy from the analysis of this literature to present such works via four dimensions: the type of software repositories mined, the purpose, the methodology used, and the evaluation method. According to them, the taxonomy is expressive enough to represent a

wide spectrum of MSR investigations, and effective enough to facilitate comparisons among MSR investigations. A more recent survey was prepared by Kwan & Damian [43]. Their literature was classified according to the area of application, including:

- **Identifying and Predicting Software Quality:** in general, the research wants to investigate two questions: "How many defects will be in this module?" and "In which modules do defects occur?" Some of these techniques also develop prediction models;
- **Identifier Analysis and Traceability:** it usually applies static analysis to trace information of interest from code artefacts;
- **Clone Detection:** it is the investigation of duplicated sections of source code, usually through the copy and paste function in an editor;
- **Process Improvement and Software Evolution:** approaches that focus on how source code changes over time;
- **Social Aspects in Software Development:** it focuses on social and human factors that could be attributed to issues in software engineering;
- **Recommender Systems and Interactive Systems:** these techniques are usually used as interactive tools that respond to the user's actions and use the information available in a repository to assist the user with decisions.

Next, we provide extra information with the goal of classifying the current approach.

2.3.1. Classifying the Approach According to Mining Software Repository

Based on the Hassan's examples of repositories [42], the proposed approach mines both **runtime** and **historical** repositories. The framework that automates the proposed approach has implemented its own customized strategy for mining software repositories. It examines control version systems and issue tracking systems to retrieve information related to changes which could be associated to code assets with performance deviation in a software evolution. Before mining these changes, the framework performs a deviation analysis process to mine databases that contain logged data from the system execution as result of dynamic analysis.

On the other hand, based on the Kwan & Damian [43] classification, our current purpose for mining software repositories, we considered that it fits mainly in the group **Process Improvement and Software Evolution**. This kind of work is particularly interested in identifying common problems that may happen during an evolution in order to contribute to improve the understanding on how the implementation evolves over time. In this context, the

approaches can have very particular goals, for example, identifying who should fix a bug [44], or identifying classification techniques that are able to predict refactoring activity [45]. Robles et al. [46] use source code extraction to examine changes over time in Debian, including the size of the source code, the number and size of the packages, and the use of various programming languages.

Our proposed framework, which automates the evaluation approach, analyzes how the source code associated with elements of performance degradation or optimization is changing over time, but some features from the group **Identifying and Predicting Software Quality** can also be seen because we could be interested in some similar questions, and the framework is also able to calculate some complexity metrics, which we used to build a preliminary regression model (Chapter 7).

2.4. Categorizing Software Erosion Approaches

Silva & Balasubramaniam [1] present a survey with techniques proposed over the last years to deal with software erosion in evolution of systems. All research work described in their survey are classified in three main categories. Software erosion happens when the implementation of the system is not according the original specification after an evolution. This process can happen in different levels, such as degradation of quality attributes and architectural compliance, and due to many reasons related to the way systems evolve, for example, the inclusion of new requirements, migration to new technologies, or just because of the regular maintenance process over time.

Such classification categorizes the approaches in three group: minimize, prevent and repair software erosion. Each one has some subcategories that will be summarized as follow. The goal here is to provide enough information about the classification in order to make us able to classify the proposed approach and analyze some research work from the same category. In this context, it is important to clarify that is not a goal of our work to deal with software erosion. However, analyzing different releases of a system in order to determine performance deviation during an evolution is directed related to the definition of software erosion.

The first category is minimize, and contains those approaches which are focused on limiting the occurrence and impact of erosion but may not be effective in eliminating it. The subcategories are:

- **Process-oriented architecture conformance** includes software engineering processes that ensure architecture conformance during system development and maintenance;
- **Architecture evolution management** covers methods available for managing the evolution of architectural specifications in parallel with implementation artefacts;
- **Architecture design enforcement** incorporates methods and tools available for transforming architectural models into implementation.

The second category is prevent, and include approaches that aim to completely eradicate erosion. The subcategories are:

- **Architecture to implementation linkage** includes mechanisms that associate or embed architectural models within source code and support the ability to monitor architectural compliance at runtime;
- **Self-adaptation** technologies enable systems to reconfigure themselves to align with their architectures after a change has been made to either the implementation or the runtime state.

Finally, the third category is repair, and contain a set of approaches that attempt to repair damages caused by erosion and to reconcile implementation with the architecture. The subcategories are:

- **Architecture recovery** involves extracting the implemented architecture from source code and other artefacts;
- **Architecture discovery** techniques are useful for eliciting the intended architecture from emergent system properties in the absence of architectural documentation;
- **Architecture reconciliation** methods help to reduce the gap between the implementation and the intended architecture of a software system.

An approach can fit in more than one category according to this classification. In the same way, different approaches could be combined as the authors show in their research. On next subsection, we are going to classify the proposed approach according to this classification.

2.4.1. Classifying the Approach According to Software Erosion

Considering the presented categories for software erosion, our approach could be classified in the category which helps to **minimize** the erosion by applying **architecture evolution management** techniques because of the similarity of some strategies usually used

for approaches in this category, and mainly because of the common goal of helping developers to manage erosion, and not prevent it or repair damaged systems. However, our work is also quite different because it is not only limited to static analysis, but it uses some architectural information added to the source code, such as scenarios, in order to perform dynamic analysis and automate the performance deviation analysis.

Due to these last features, the proposed approach is also similar to some research work from the **prevent** category, even that preventing erosion is not a goal, which is the case for the subcategory of **architecture to implementation linkage**. In this kind of work, the system's implementation is linked to its architecture to enable monitoring the structural compliance between them. Such works usually provide an infrastructure for evolving and executing the monitored systems. For example, ArchWare [47] provides a complete architecture-centric platform for developing evolvable software systems. The platform consists of an ADL for specifying executable architectures, and a runtime monitoring infrastructure to enables an executing system to be dynamically evolved at the architectural and implementation levels simultaneously. Our proposed approach can monitor the execution of the system in order to intercept method calls from scenarios. The architectural scenarios are linked to the source code with annotations. However, as said before, despite the similarities with some common strategies, our approach is not focused on prevent software erosion.

Returning to the first **minimize** classification (**architecture evolution management**), the approaches usually work versioning the software architecture and the system implementation with software configuration management (SCM) tools, also known as version control systems. A number of methods have been developed that exploit the capabilities of SCMs to specifically control, trace and record changes made to models of an architecture and its corresponding implementation. An example of this subcategory is the architecture evolution environment called Mae [48]. It combines SCM principles and architectural concepts in a single model so that changes made to an architectural model are semantically interpreted before the application in the SCM.

Another work from this subcategory is the ArchEvol tool [49]. This tool enables monitoring the evolution of the architecture specification and its corresponding implementation. Architecture specifications and Java code are linked through two developed plugins in Eclipse and ArchStudio [50]. ArchEvol allows architects and developers to work in parallel on their respective domains of the system and it takes the responsibility for maintaining

the consistency. This approach treats architecture as a structure consisting of only components and connectors, what might limit its usefulness. In this way, Molhado [51] is a tool similar to ArchEvol, but with additional support for interface elements.

Still in the same subcategory, ArchAngel [52] monitors the architectural conformance of the source code against a dependency model defined by the architect. It basically performs static code analysis to identify package relationships and compares these relationships to the dependency model of the intended architecture. Any relationship that does not conform to the model triggers a fault. The SCM plays the role of informing the ArchAngel engine when source files are edited. This tool can detect dependency violations, but it does not have the capability of reducing erosion from other types of violations.

Despite the similarities with the works summarized below, the current approach does not aim directly software erosion or data versioning. Indeed, control version systems are used for the approach as sources of data to be mined for rich information that help to explain the degradation or optimization of the performance quality attribute. However, the SCMs considered contain information only related to the implementation, and do not contain any information related to architectural models.

Classifying the current approach is a very complex task. Many of its characteristics could be found in more than one category. In this case, the category that has the most similar goals is **architecture evolution management**, even with some similarities with **architecture to implementation linkage**. Indeed, this decision was taken because prevent erosion is not part of the goal, but providing the continuous evaluation of scenarios, mainly for performance deviation analysis, and minimize the erosion of quality attributes in scenarios architecturally relevant to avoid degradation. The approach is not capable of preventing software erosion by itself. The effort to accomplish this approach in one of the defined categories is useful to help us to understand the context in which the work is inserted and to give us an initial idea of what techniques are more common in this context.

3. APPROACH FOR EVALUATING EVOLVING SOFTWARE SYSTEMS

This chapter gives an overview of the proposed approach and details each one of its phases. It focuses on describing the approach at a conceptual level by showing its different phases, and inputs/outputs artifacts. Next chapter shows how each phase was implemented in an object-oriented framework that uses static analysis, dynamic analysis and repository mining techniques to provide an automated way to revel potential sources of performance deviation of scenarios between releases of a software system.

This chapter is organized as follows: **Section 3.1** presents the two ways to use the approach; **Section 3.2** describes the phases of the approach for performance deviation analysis support, which is our primary purpose; and **Section 3.3** describes the phases to use the quality attribute traceability support for quality attributes, which is our secondary usage, still in preliminary phase.

3.1. Approach Usage: Performance Analysis and Quality Attribute Traceability

The proposed approach can be used for two purposes. The primary use is for performance deviation analysis. It combines dynamic analysis with repository mining techniques to define a scenario-based approach for the evaluation of the quality attribute of performance, measured in terms of execution time (response time). The secondary use consists on statically analyzing the source code to trace scenarios and quality attributes in order to highlight points in the source code that may represent risks related such quality attributes when evolving the system's implementation. Our studies have focused on the primary purpose for performance deviation analysis. As explained, it is the main usage of the approach. The first phase of the approach is shared between both purposes.

3.2. Performance Deviation Analysis Support

The main usage of the proposed approach is for performance deviation analysis and defines four phases, which are: **(i)** release preparation – choosing the scenarios and preparing the target releases to be evaluate by indicating properly scenario entry points; **(ii)** dynamic analysis – determining the performance of scenarios and methods by instrumenting the release execution and calculating the execution time (response time) of scenarios and methods of interest; **(iii)** deviation analysis – processing and comparing the results of the dynamic analysis

for two different releases of a system in order to determine which scenarios and methods had performance deviation in terms of execution time; and **(iv)** repository mining – identifying development issues from the issue tracking systems and commits from the version control systems related to those elements with performance deviation.

The release source code and the list of scenarios are used as input of the first phase, which produces a version of the source code with integrated metadata. The list of scenarios must be defined externally to the approach containing the scenarios or the main scenarios of the system. The output of the phase one is used as input of the phase two, which performs the dynamic analysis process. It instruments the release execution – by using AspectJ, for example – and, progressively, persists the dynamic analysis model to a database for posterior analysis which contains the logs collected during the instrumentation process. Phases one and two are summarized in Figure 3.

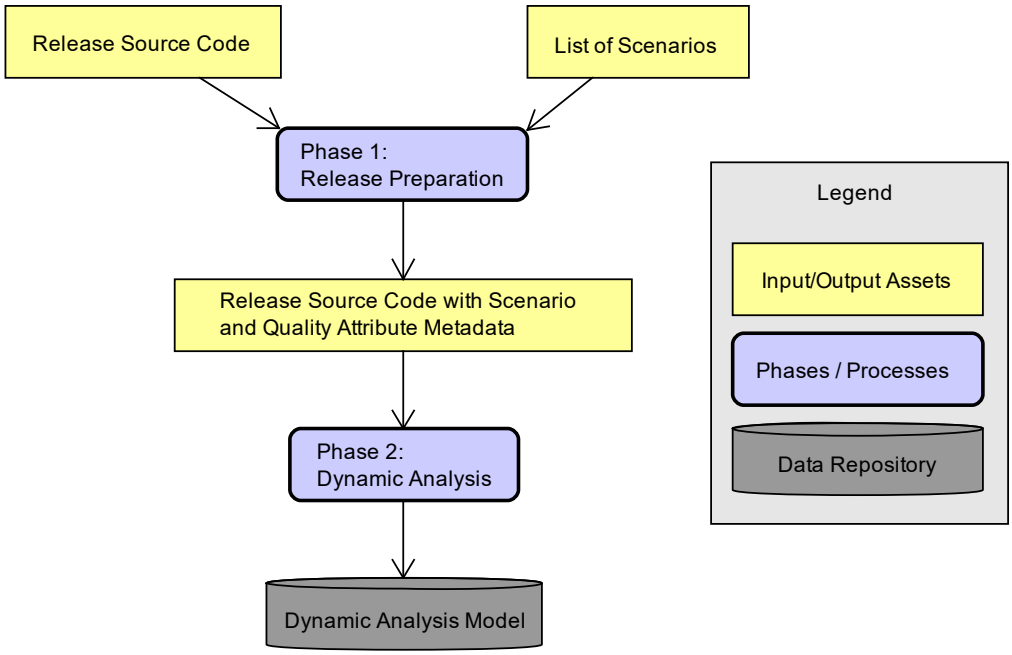


Figure 3. Phases one and two of the approach for performance deviation analysis.

Next phases perform the evolution analysis. The third phase receives as input the dynamic analysis model of two target releases of the system. Thus, phases one and two have to run twice, once to generate the dynamic analysis model of each release. The dynamic models are compared to discover the elements with performance deviation. Last phase mines such elements in repositories to retrieve development issues and commits related to them. Phases three and four are summarized in Figure 4.

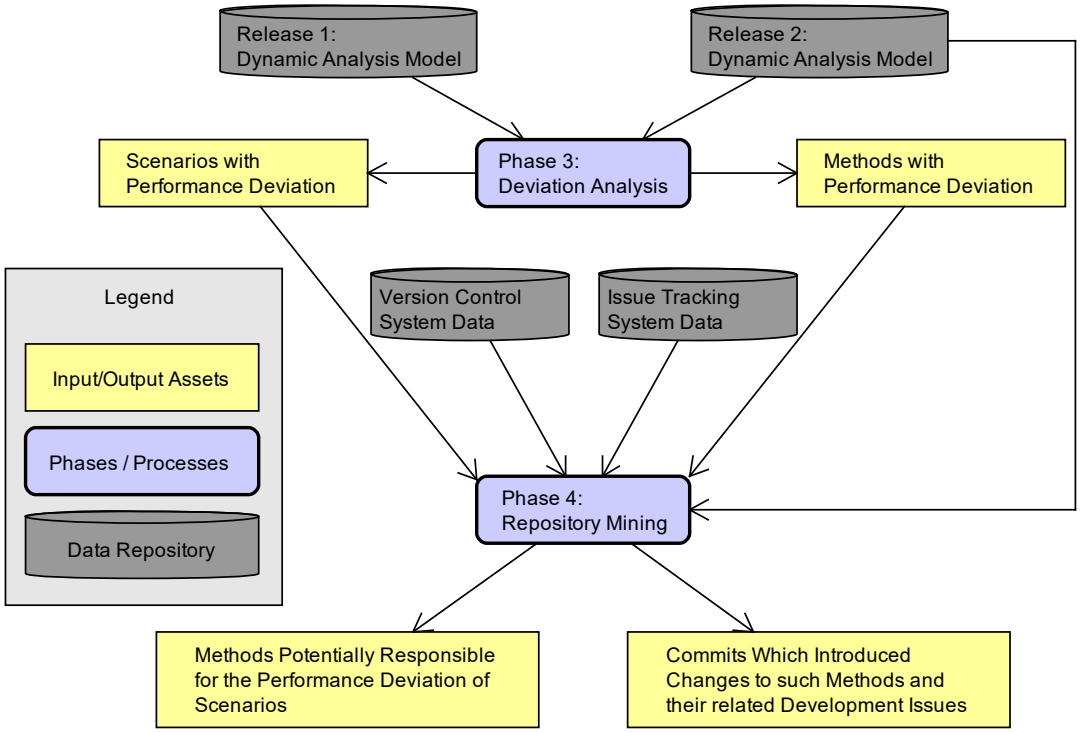


Figure 4. Phases three and four of the approach for performance deviation analysis.

Next subsections present a more detailed explanation about each phase of the approach for performance deviation analysis support.

3.2.1. Phase 1: Release Preparation

This phase consists mainly of selecting relevant scenarios, and identifying in the source code the methods that start the execution for each selected scenario. It receives the target release source code and the list of scenarios of the system as input, which must be defined externally to the approach containing the scenarios or the main scenarios of the system. The output is the source code of the release, integrated with such metadata information of scenarios and quality attributes, ready for the next phase (dynamic analysis). The release preparation phase is performed in three steps, which are illustrated in Figure 5.

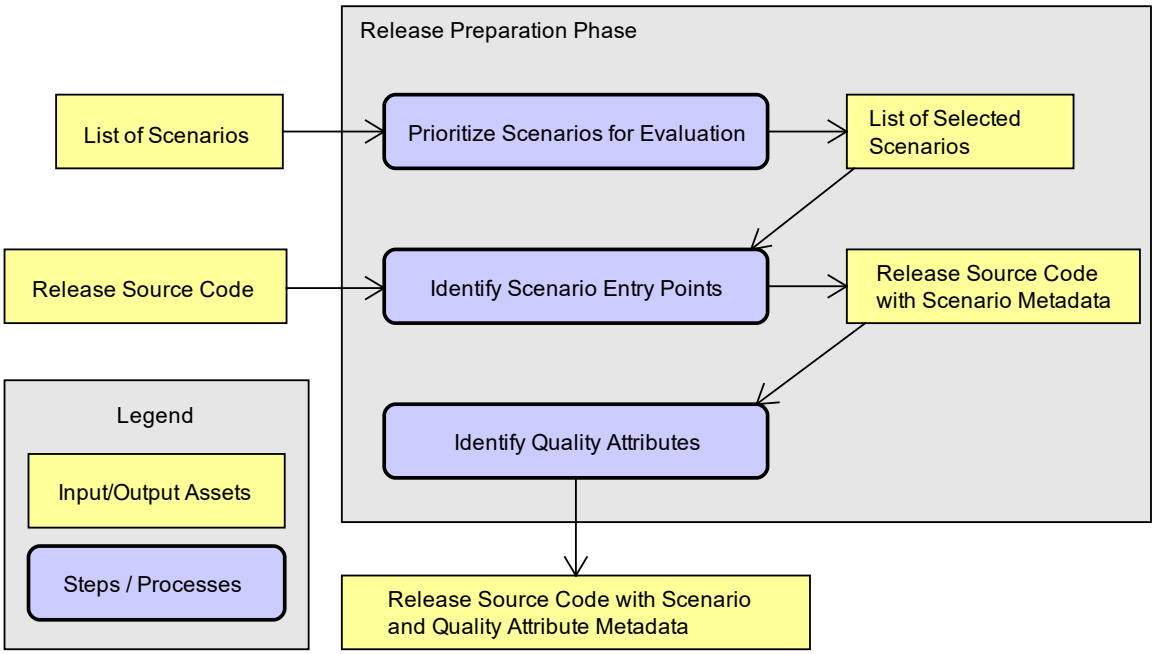


Figure 5. Release preparation phase.

Here, the identification of the quality attributes (third step in Figure 5) in the source code is not mandatory and may be skipped since the goal is to conduct the performance deviation analysis for every annotated scenario. At this moment, other quality attributes are not addressed, but can still be annotated with documentation purposes, or for posterior use. It is important to reinforce that the most important task in phase one is to identify the methods which represent scenario entry points with annotations in the source code.

Prioritize Scenarios for Evaluation is first step of this phase. Its goal is to choose relevant scenarios that deserve to be evaluated. That information can come, for example, from traditional methods of software architecture evaluation executed in previous moments of the development process. The idea of prioritizing the scenarios is inspired in the same named step of ATAM. A large-scale system may have several scenarios which could be evaluated, and despite the importance of every scenario, sometimes architects and developers are interested in more critical or particular functionalities. Thus, this step gives them the opportunity to choose particular scenarios from a complete list. Scenarios vary according to the system, for example, in a library component of an academic management system, a scenario could be *Check User Status* to verify if the user is able to borrow items from the library, such as books, journals and magazines. On the other hand, *Save Diagram* would be a scenario of interest in a modeling tool,

and *User Authentication* could be an important scenario in a large amount of enterprise information systems.

The second step is **Identify Scenario Entry Points**. Here, the goal is to identify starting methods which represent execution entry points for scenarios of interest. An entry point is the method that starts the scenario execution. Thus, this step consists of identifying in the source code the methods that start the execution of each selected scenario. It can be done by adding metadata to the source code in order to mark these methods, what is currently done by source code annotations. Considering the previous example of *Check User Status* scenario, it should be annotated with `@Scenario(name = "Check User Status")` in the `selectedUser()` method if it represents the entry point of such scenario, it means the method that starts its execution.

Next step, **Identify Quality Attributes**, involves the insertion of metadata (annotations) in the source code to represent the quality attributes of the system. Thus, it is possible to mark specific methods to inform that they are critically associated with particular quality attributes, or they implement design decision that affect particular quality attributes. Such information is used to perform traceability among quality attributes and source code artifacts, and to indicate potential tradeoffs points in scenarios that execute different quality attributes. As said before, annotations were used as source of metadata. In this context, quality attribute annotations are defined and currently used only to identify and to document methods of interest in the source code related to quality attributes. Some examples of these annotations are: `@Security`, `@Performance`, `@Reliability`, and `@Portability`. Thus, the output of the preparation phase is the source code of the target release integrated with metadata to identify scenarios and quality attributes of interest. This step is optional when addressing performance deviation analysis, but it is required if quality attribute traceability support with static analysis is desired, as will be explained in Subsection 3.3.1.

3.2.2. Phase 2: Dynamic Analysis

The second phase is named dynamic analysis (see Figure 3), and it requires the execution of the scenarios through a manual or automated way. This phase generates the dynamic analysis model, which is persisted in a database and contains information about the execution traces of the system modeled by a dynamic call graph that represents every execution of the selected scenarios for the target release. The dynamic call graph is assembled by intercepting the entry point methods and instrumenting their executions, while calculating the

execution time of each node as well as the information to say if it has failed or not. This call graph can be interpreted as a tree structure where each node is an execution of a regular method or a constructor, and the entry point methods represent the root nodes.

It is important to emphasize that our work only considers the performance quality attribute through the computation of the execution time. In addition, the dynamic analysis must ideally be executed in a controlled environment and using an automated way. Thus, it is fundamental to have a suite of automated tests for the system in order to enable the system execution. The quality and accuracy of the dynamic analysis model is directly related to the coverage of the tests used to monitor the execution of the selected scenarios. If the system does not have automated tests, an alternative strategy to execute the scenarios must be used. For example, to use JMeter performance testing tool [53] to submit requests that exercise scenarios of interest in a web information system which does not have automated tests.

This phase and the previous one need to run twice, once for each target release, if a performance deviation analysis for an evolution is desired. In this case, two databases with the dynamic analysis model of each release under comparison, are the outputs of this phase (see Figure 4).

3.2.3. Phase 3: Deviation Analysis

The degradation analysis is the third phase of the approach. It compares the release execution data (dynamic analysis model) extracted during the dynamic analysis phase for two different releases. The comparison reveals scenarios and methods of the system that were degraded or optimized over the target evolution. Two strategies to compare the execution time of scenarios and methods could be used, including arithmetic mean and statistic test. The outputs of this phase are reports with degraded and optimized scenarios/methods in terms of execution time (see Figure 4), considering one of the comparison strategies (arithmetic mean or statistic test).

The arithmetic mean strategy compares the average execution time for each method in both releases. If the value in the newer release increased or decreased more than a configured threshold, it considers that a performance deviation happened in the method. The second strategy should use a statistic test, for example, Mann-Whitney U-Test [82], to observe if two independent samples, which do not necessarily follow a normal distribution, have the same

tendency. In this case, the samples are formed by the set of the execution time values for each common method in each release.

It is important to emphasize that during the dynamic analysis phase, the execution time of some methods can be affected by the execution of other methods that are directly or indirectly called by them. This way, the increase in execution time of one method will affect all its parent nodes in the dynamic call graph. In this context, the deviation analysis phase will consider all these methods with performance deviation, but, actually, just one method should be blamed for the degradation. The next phase minimizes this situation by selecting only the methods that were also changed.

3.2.4. Phase 4: Repository Mining

The last phase mines data from the version control and issue tracking systems to find which specific commits and development issues have changed the methods identified in the previous phase. This phase retrieves commits from the version control system for each class that contains methods detected with performance deviation (degradation or optimization). If the commit changed lines inside the method of interest, the commit log is searched for issue numbers, which are used to access the issue tracking system and to find extra information to complement the commits, including the issue type (bug, improvement, new feature, etc.), for example.

Since the approach is guided by scenarios, the framework only considers degraded or optimized methods that impact at least one degraded or optimized scenario. The methods identified with performance deviation in the previous phases, but not changed during the evolution, are also selected and saved, but not included to the final report because they probably do not represent real sources of performance deviation. They are mainly impacted by other methods because of the call method hierarchy.

The final output (see Figure 4) contains: **(i)** degraded/optimized scenarios and degraded/optimized and changed methods potentially responsible for affecting them, and **(ii)** the associated code changes (commits and development issues) that introduced changes to any of these methods during the evolution. This information allows developers to analyze commits and development issues in order to understand the modifications and the reasons why they introduced performance deviation.

The full procedure to apply the approach for performance deviation analysis can be seen in Figure 6. Note that phases one and two need two applications in this case. Their outputs are used as input of the subsequent phases, resulting in the final reports, as explained before. In addition, the selected target scenarios, which will be evaluate, should be the same in both releases in order to make possible the comparison process during the deviation analysis phase. Obviously, only the common scenarios and methods between the releases are compared to each other.

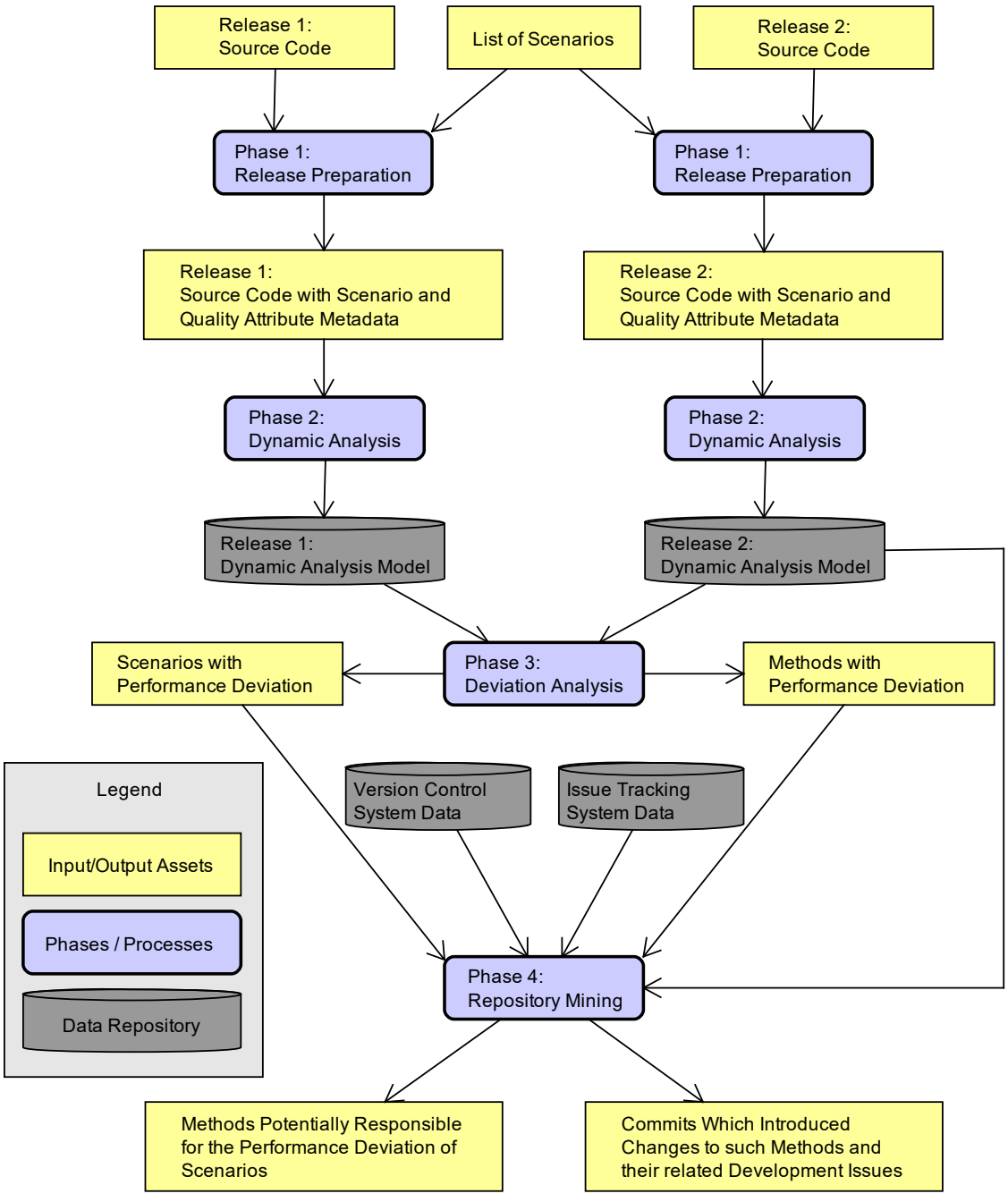


Figure 6. Full approach application for performance deviation analysis.

3.3. Quality Attribute Traceability Support

The secondary way of using the evaluation approach is to perform quality attribute traceability operations. It uses static analysis to parse the source code and build the static analysis model, represented by a static call graph of scenarios and methods. The static analysis model can be used to trace information related to scenarios and their corresponding quality attributes. The comparison of two static analysis model of different releases of a software system can identify added, removed or changed scenarios, in terms of source code implementation of methods and quality attributes.

This strategy to use the approach consists of two phases: **(i)** release preparation – preparing the source code of a particular release; **(ii)** static analysis – executing a support tool to perform the static analysis of the prepared source code that comes from the previous phase. In this context, the static analysis phase can be used to generate and analyze individual static models from a particular release, or to generate and compare static models from two different releases.

The support for quality attribute traceability is not the main focus of this work. Because of that, this thesis does not present any studies related to that. Preliminary studies with traceability and tradeoff points among quality attributes were previously published [85] [86] [87] [88].

3.3.1. Phase 1: Release Preparation

Before executing the static analysis phase, the source code of a target release of the system has to be prepared. This phase uses the source code of the release, and the complete list of scenarios as input of the process. The output is the source code of the release, integrated with such metadata information of scenarios and quality attributes, ready for the static analysis. It follows the exact same three steps of the release preparation phase presented in Subsection 3.2.1 and illustrated in Figure 5. At this moment, the identification of the quality attributes by using annotations (third step in Figure 5) is particularly important to enable the static traceability support.

3.3.2. Phase 2: Static Analysis

The static analysis phase can be performed to analyze an individual static model extracted from a particular release, or to compare two static models extracted from two different releases of a system.

In the first case, the output of the phase one is parsed in order to assemble the static analysis model, which is analyzed in the process. The output is the scenario and quality attribute traceability report that contains: **(i)** the relationship between quality attributes and source code artifacts as methods, classes and packages, and **(ii)** the potential tradeoff points in scenarios that execute different quality attributes. A simple strategy is used to determine scenarios with potential tradeoff points. Basically, scenarios associated with more than one quality attribute because they may execute methods related to such quality attributes are potentially highlighted. The static analysis model, assembled in the **Parse the Source Code** step, is basically one or more static call graphs of the release using the methods that represent scenario entry points as root nodes. The report is generated as output of the **Static Model Analysis** step. This process is summarized in Figure 7.

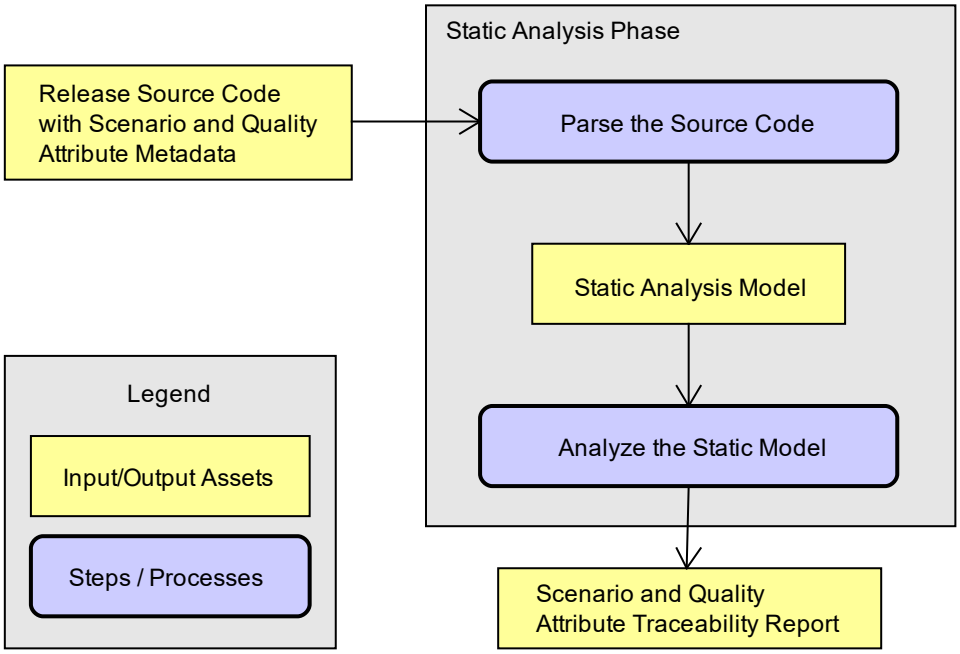


Figure 7. Static analysis for individual release evaluation.

In the second case, the static analysis is used to parse the prepared source code of two different releases from phase one (**Parse the Source Code** step), and to compare the assembled static analysis models (**Static Model Comparison** step) in order to produce the scenario and quality attribute evolution report as output of the process. Note that in this case, the phase one runs twice to produce the source code with scenario and quality attribute metadata for each target release. Here, the goal is to statically evaluate an evolution of the system considering its scenarios and quality attributes. The report generated by the comparison between the static

models contains information of the: **(i)** added, removed and changed scenarios between the releases; **(ii)** added and removed identified quality attributes in the source code; **(iii)** added and removed potential tradeoff points inside common scenarios between the releases. In this context, a scenario is considered changed when it has different static call graph in each release, and a renamed scenario is detected as removed from the first release and added to the second one. This second process is summarized in Figure 8.

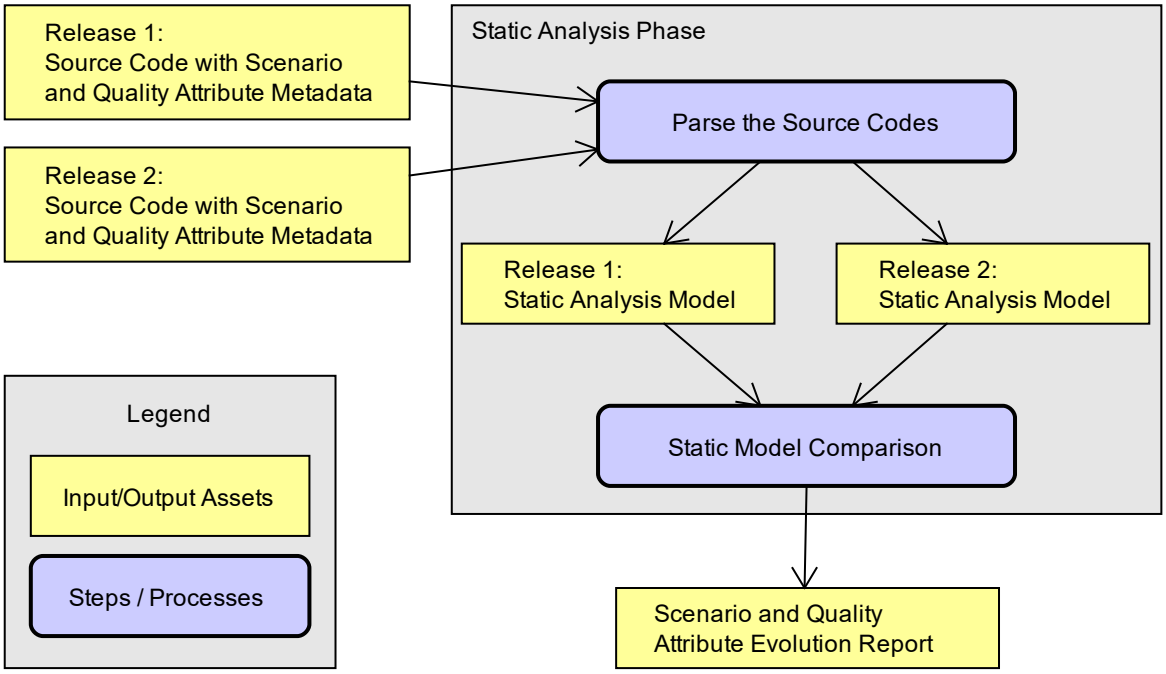


Figure 8. Static analysis for evolution evaluation.

The support for static analysis was implemented as an exploratory experience. Because of that, we only conducted a very preliminary studies by using traceability support with static analysis which are not going to be detailed here, but are available in previous publications [85] [86] [87] [88]. Indeed, the main efforts of this research work and its studies were concentrated on the performance deviation analysis process, which combines techniques for dynamic and static analysis, and repository mining. It is the main aim of the proposal, while the traceability support with static analysis plays a secondary role.

4. AUTOMATING THE EVALUATION APPROACH

A support framework was implemented to automate the evaluation approach. It integrates tools for static analysis, dynamic analysis and repository mining. It has been implemented in the Java programming language. The instrumentation process during the dynamic analysis phase was developed using AspectJ. The framework addresses the evaluation of software systems developed in the Java language.

This chapter follows the same structure of the previous one in order to show how the framework deals with each phase of the proposed approach. First, **Section 4.1** shows the framework support for dynamic analysis and repository mining in order to automate the performance deviation analysis of system releases, which is the primary use of our approach. After that, **Section 4.2** presents how the quality attribute traceability support was implemented, mainly the static analysis support.

It is important to emphasize the current implementation for performance deviation analysis and quality attribute traceability are independent although they share common classes. It means they are not related and have been accomplished as two independent components: **(i)** the traceability support is implemented as an Eclipse plugin; and **(ii)** the performance deviation analysis is made available as a set of libs (JAR files).

4.1. Performance Deviation Analysis Support

As explained before, in Section 3.2, the main purpose of the approach is to provide performance deviation analysis support in terms of execution time for selected scenarios. Here, Java annotations were used as source of metadata to identify the methods that represent scenario entry points as well as the methods related to particular quality attributes or which implement design decisions that affect particular quality attributes in the source code. The dynamic analysis performs the instrumentation of the execution through the definition of aspects using the AspectJ language.

Next subsections present how each phase for performance deviation analysis was addressed by the framework. Thus, Subsection 4.1.1 shows the preparation phase and how to support new annotations. Subsection 4.1.2 presents the current support for dynamic analysis by using `@Scenario` annotation or `@Test` annotation from JUnit 4, and how to extend the

abstracted defined aspect to intercept new kind of annotations. Subsection 4.1.3 discusses the implemented support for the deviation analysis phase, while Subsection 4.1.4 presents the framework support for repository mining.

4.1.1. Release Preparation Phase: Java Annotations as Source of Metadata

As explained on the previous chapter, the approach requires the system's source code and additional metadata information in order to proceed with the evaluation process. The code annotations are used, for example, to indicate entry points of relevant scenarios that will be monitored by the framework. The annotations currently implemented by the framework are shown in Figure 9.

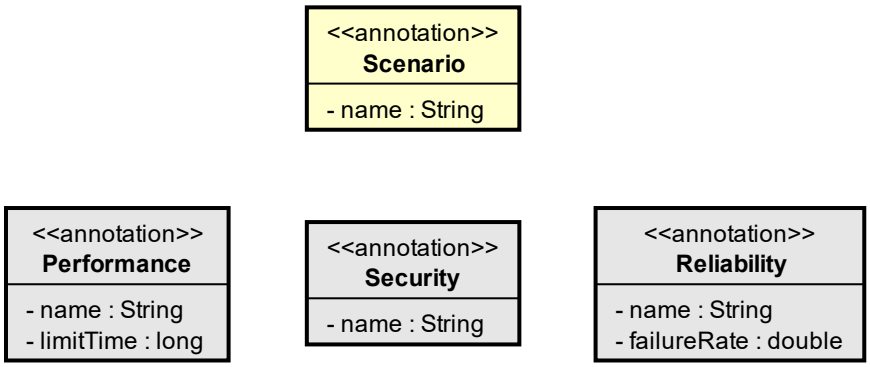


Figure 9. Currently supported annotations by the framework.

Thus, the framework identifies method entry points of scenarios in the source code using the default annotation `@Scenario`. When building the analysis models, the framework will consider these methods as root nodes of the call graphs. The `@Scenario` annotation defines only one attribute which is a string to represent the name of the scenario. Such value should be unique in order to difference every annotated entry point. During the dynamic analysis, the methods identified with such annotation will be monitored by the framework which provides an aspect to intercept and instrument the system execution starting from the entry point method.

The performance deviation analysis is performed for every method annotated with `@Scenario`. In addition to the scenario annotation, the framework also provides annotations for the quality attributes of security, performance and reliability with the only purpose of documenting methods that are important to achieve desired quality attributes and to enable traceability of quality attributes to code assets and scenarios, which will be detailed in Section 4.2.

In this context, the `@Security` annotation defines one string attribute (`name`) which uniquely identifies each occurrence of the annotation. The `@Performance` annotation has two attributes: `name` and `limitTime`. Name is a string that uniquely identifies it, and limit time is a long integer that specifies an expected maximum time in milliseconds. The related method must complete its execution in a shorter time compared to the limit time value. It enables the possibility of monitoring if particular methods related to this metadata are executing according to the expected time.

The `@Reliability` annotation has a string name attribute that is unique and a double attribute that specifies the failure rate. This represents the maximum failure rate expected for a particular method from zero to one. Zero means that it never fails and one that fails in all the cases, in the same way, a value of 0.9 means it fails in 90% of the cases. As well as performance, such attribute enables the possibility of monitoring if the method fails more times than the value specified in the attribute.

As said before, the quality attribute annotations (`@Security`, `@Performance` and `@Reliability`) and their attributes, for example, limit time and failure rate, are currently used just for documentation. They are optional for the performance deviation analysis usage. New quality attribute annotations – with documentation purposes – can easily be added by extending the framework. Such procedure consists of providing a new annotation, a new annotation reader, and extending the dynamic analysis model, which is used to keep information from the system execution, to ensure consistence.

The classes related to this procedure are `QAAnnotationReader`, which provides a generic reader for annotations, and `RuntimeQAAnnotation`, which keeps the annotation data saved. The new annotation has to be annotated with `@QualityAttribute` annotation in order to indicate its reader. A partial example of this procedure is shown in Figure 10 and Figure 11 for the performance annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@QualityAttribute(PerformanceReader.class)
public @interface Performance {
    public String name();
    public long limitTime() default 0;
}
```

Figure 10. An example showing the Performance annotation definition.

```
public class PerformanceReader extends QAAnnotationReader {
    public RuntimeQAAnnotation
    readAnnotation(Annotation a, Method m) {...}
}
```

Figure 11. An example showing how to define new annotation readers.

In Figure 10, it is possible to see that the performance annotation is annotated with `@QualityAttribute` which indicates `PerformanceReader` as the reader of such annotation. `PerformanceReader` class extends the `QAAnnotationReader` and parses the performance annotation in the source code to feed the model with the collected information. Thus, when the generic annotation reader (`QAAnnotationReader` in Figure 11) finds an annotation, it checks if it has the `@QualityAttribute` and, in positive case, the specific reader is instantiate at runtime. The same procedure has been done to define `@Reliability` and `@Security` annotations, and can be applied to any new quality attribute annotation for general purpose.

The framework provides the `@Scenario` annotation as a default way to identify entry points of scenarios. Such annotation can be replaced according to the needs of developers and architects. For example, in our case studies, the `@Test` annotation from JUnit 4 was used instead. The procedure to use a different entry point annotation is specified in the next subsection.

4.1.2. Dynamic Analysis Phase: AspectJ Instrumentation Support

The second phase requires the execution of the scenarios through a manual or automated test suite. This phase generates the dynamic analysis model, which is persisted in a database, and contains information about the execution traces of the system modeled by a call graph that represents every execution of the selected scenarios for the target releases. The support framework uses AspectJ to define an aspect to instrument the executions of the scenarios, intercepting the entry point methods to build the call graph and calculating the execution time of scenarios and methods of interest during the process. The methods which represent entry points for scenarios were annotated with an entry point annotation, such as `@Scenario`. The methods selected as entry points for scenarios represent root nodes in the dynamic call graph and each node represents an execution of a regular method or constructor.

By using AspectJ, we can define an abstract aspect, which incorporates the logic of the instrumentation process, but is not dependent on a specific entry point. It lets us to define different concrete strategies to identify and intercept method entry points of particular

scenarios, for example, by using different annotations (not only `@Scenario`) or choosing methods from particular classes. In addition, the use of AspectJ to implement our own instrumentation mechanism, which depends on models defined for our specific needs, seemed to be a more appropriate solution to deal with the large amount of data generated by dynamic analysis than using the output of a profiling tools, such as YourKit, JProfiling, or JMH. Because of the large amount of data generated in the profiling process, we need to persist the execution call graph of each scenario of interest. This feature was not found in existing profiling tools. Finally, the use of AspectJ is a common strategy adopted by other approaches [37] [38] to instrument system executions.

Thus, in order to allow different annotations for entry points, the framework has defined an abstract aspect named `AbstractAspectEntryPoint` which incorporates all logic of the instrumentation process. It is responsible for defining common and abstract pointcuts. A partial implementation of the abstract aspect is shown in Figure 12.

```
@Aspect
public abstract class AbstractAspectEntryPoint {

    public Class<? extends Annotation> getAnnotationClass() {
        return null;
    }

    @Pointcut
    public abstract void entryPoint();

    @Pointcut("within(scenario.analyzer..*)")
    public void ignore() {
    }

    @Pointcut(
        "cflow(entryPoint()) && (execution(* *(..)) || execution(*.new(..)))")
    Private final void entryPointFlow() {
    }

    @Around("entryPointFlow() && !ignore()")
    public final Object cgbuilding(ProceedingJoinPoint thisJoinPoint) {
        ...
    }

    @AfterThrowing(
        pointcut = "entryPointFlow() && !ignore()", throwing = "t")
    public final void throwingException(
        JoinPoint thisJoinPoint, Throwable t) {
        ...
    }

    @Before("handler(Throwable+) && args(t) &&
        entryPointFlow() && !ignore()")
    public final void handlingException(
        JoinPoint.EnclosingStaticPart thisEnclosingJoinPointStaticPart,
        Throwable t) {
        ...
    }

}
```

Figure 12. Abstract aspect implementation for entry points.

Thus, the pointcut `entryPointFlow()` defines what should be instrumented. Basically, it will intercept the execution of methods and constructors inside the execution flow of entry points, defined by `entryPoint()`, which is an abstract pointcut that have to be implemented in the concrete aspect. The `throwingException()` and `handlingException()` methods have the function of intercepting exceptions after they are thrown and before they are handled, respectively. It allows the framework to keep information about exceptions inside the scenarios of interest, for example, the methods where they are thrown and where they are handled, or which exceptions were not handled at all.

The `cgbuilding()` method is the one responsible for coordinating the assembling of the dynamic call graph and for feeding the dynamic analysis model with information from the

system execution. It basically intercepts each element that matches with `entryPointFlow()` and does not match with `ignored()`. By default, the framework ignores only its own implementation, but concrete aspect implementations can override this behavior if necessary.

The `getAnnotationClass()` method is used by the framework to get a class reference of the entry point annotation. This method has a default implementation because it sometimes does not need to be implemented in the concrete aspect. Thus, developers can define entry points for scenarios without annotations, if they need. For example, in order to reuse JUnit 3 tests as entry points, the concrete aspect could be specified to intercept methods that start with the *test* word and extends the `junit.framework.TestCase` class. In this particular case, an annotation is not required as shown in Figure 13.

```
@Aspect
public class AspectJUnit3EntryPoint extends AbstractAspectEntryPoint {

    @Pointcut(
        "within(junit.framework.TestCase+) && execution(* test*(..))")
    public void entryPoint() {
    }

}
```

Figure 13. Aspect implementation for JUnit 3 entry points.

In similar way, concrete strategies to intercept any annotation could be implemented. Figure 14 and Figure 15 shows the aspects to intercept the `@Scenario` annotation defined by the framework and the `@Test` annotation from the JUnit 4 framework, respectively.

```
@Aspect
public class AspectScenarioEntryPoint extends AbstractAspectEntryPoint {

    public Class<? extends Annotation> getAnnotationClass() {
        return scenario.analyzer.annotations.Scenario.class;
    }

    @Pointcut(
        "execution(@scenario.analyzer.annotations.Scenario * *(..))")
    public void entryPoint() {
    }

}
```

Figure 14. Aspect implementation for @Scenario annotation.

```
@Aspect
public class AspectJUnit4EntryPoint extends AbstractAspectEntryPoint {

    public Class<? extends Annotation> getAnnotationClass() {
        return org.junit.Test.class;
    }

    @Pointcut("execution(@org.junit.Test * *(..))")
    public void entryPoint() {
    }

}
```

Figure 15. Aspect implementation for JUnit 4 @Test annotation.

After `entryPoint()` is defined, the `cgbuilding()` is able to intercept the execution of methods of interest. The assembled dynamic call graph represents the scenario execution in the exact same way it has happened, excluding the aspect advices. For example, Figure 16 shows a `User` entity class that represents users of a library management system. Its attributes indicate if the user is active, has any loan overdue or has a fine. The user needs to be active and has no pendency with the library (without overdue or fine) to be able to borrow items from library. Figure 17 shows that the `selectedUser()` annotated method is the entry point of the *Check User Status* scenario that verifies the user status. The dynamic call graph in the analysis model must represent exactly the execution starting from that method. Figure 18A shows the dynamic call graph built by the abstract aspect, if the values for a specific user are *overdue=true*, *fine=false* and *active=true*.

```
public class User {
	...
	private boolean overdue, fine, active;

	public User(boolean o, boolean f, boolean a) {
		overdue = o; fine = f; active = a;
	}

	public boolean isOverdue() {
		return overdue;
	}

	public boolean hasFine() {
		return fine;
	}

	public boolean isActive() {
		return active;
	}
	...
}
```

Figure 16. An imaginary user entity class for a library management system.

```
...
private User u;

@Scenario(name = "Check User Status")
public boolean selectedUser() {
	return !hasIrregularity() && u.isActive();
}

public boolean hasIrregularity() {
	return u.isOverdue() && u.hasFine();
}
...
```

Figure 17. Working example for dynamic call graph.

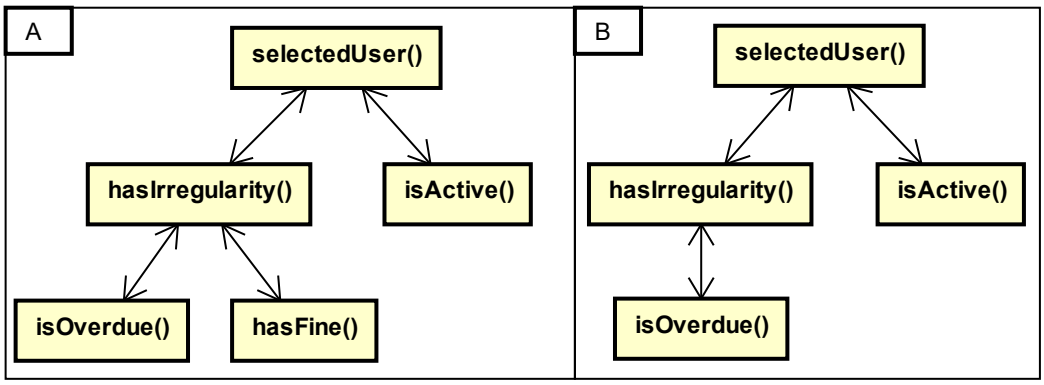


Figure 18. Dynamic call graph example.

It is important to realize that the dynamic call graph contains only the methods executed for a specific scenario. It means if the method exists in the source code, but it is not executed, the call graph will not contain it. Figure 18B illustrates what happens in the dynamic call graph if we consider the values *overdue=false*, *fine=true* and *active=true*, for example. The logical test of `hasIrregularity()` executes the `isOverdue()` method, and gets a *false* value. As result, the `hasFine()` method will not be executed because the logical expression *false* in this case is enough since *false* and anything else will always be *false*.

The dynamic call graph is the main element of the dynamic analysis model, but there are other important elements to represent the system execution, quality attributes and scenarios. Figure 19 shows a partial class diagram of the dynamic analysis model. The `SystemExecution` class represents a specific system execution. It has attributes that indicate the system name, the system version, and the date when the execution started.

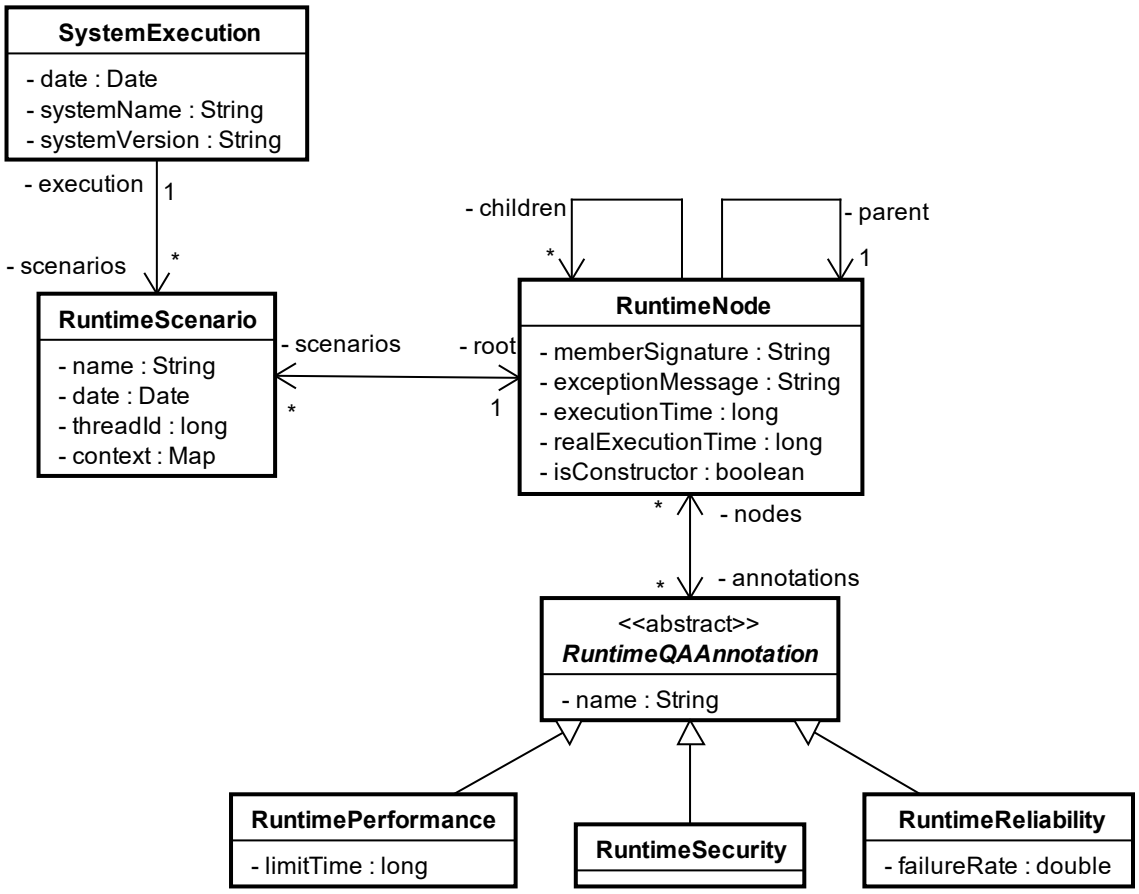


Figure 19. Partial UML class diagram for dynamic analysis model.

The `RuntimeScenario` class models every executed scenario. Scenarios have a name (*Check User Status*, for example), the date when it was executed, the id of the thread that has executed it, the context to represent web requests (in case of web systems), and its root node, Thus, the `RuntimeNode` class represents every member (method or constructor) executed inside a particular scenario. It maintains the method or constructor signature, the message if any exception has been thrown and the execution time of the method or constructor. In case it has thrown any exception and not handled it, aborting its execution, the execution time is set to -1.

A node might have associated quality attributes identified by annotations. The `RuntimeQAAnnotation` is an abstract class to indicate quality attribute annotations that were added to methods. The `RuntimePerformance` class defines the `limitTime` attribute that indicates the maximum time that the member should complete its execution. The `RuntimeReliability` class maintains the `failureRate` attribute, which represents the maximum failure rate expected for a particular method from zero to one.

During the system execution, the abstract aspect (see Figure 12) intercepts every method or constructor execution inside a particular scenario, building the dynamic analysis model progressively. At the end of each scenario execution, it persists the collected data into a database. Next section describes how the framework proceeds with the analysis of the evolution of the quality attributes using the produced analysis models.

4.1.3. Deviation Analysis Phase: Dynamic Analysis Model Comparison Support

The comparison of the models from two different releases in the third phase reveals which methods and scenarios of the system have been degraded or optimized over an evolution. The framework gets the execution time values for each method and compares the values for two releases of the system. As explained in Subsection 3.2.3, it can compare the values by using the Mann-Whitney U-Test or checking if the arithmetic mean has increased or decreased more than a configured threshold from the first release to the second one.

The first strategy compares the average execution time for each method in both releases. If the value in the newer release has increased or decreased by a default threshold of 5% or more (it can be configured in the framework), it considers that a performance deviation happened in the method. The second strategy uses a two-sided Mann-Whitney U-Test to observe if two independent samples, which do not necessarily follow a normal distribution, have the same tendency. The framework uses the U-Test to determine if the execution time of

the methods of the target scenarios in the first release has the same tendency in the second release. For each method M, the first sample consists of the set of execution time values for the method in the first release, and the second sample consists of the set of values in the second release. Our null hypothesis is “the values of the execution time for method M have the same tendency in both releases”, while the alternative hypothesis is “the values of the execution time for method M do not have the same tendency”, in other words, they are different.

For the statistic test (U-Test), the framework uses a default value to the significance level (*alpha*) of 0.05. If the *p-value* calculated using the output of the test is equal to or less than the significance level, the null hypothesis can be rejected and the alternative hypothesis is kept, what means there is a performance deviation between the releases for the method M. In that case, since it is a two-sided test and since we already know that the samples are different, the average execution time is used to determine if it increased or decreased. Developers are usually interested in degradations, but flagging optimization cases is also interesting because developers could check if some expected modifications had indeed decreased the execution time. Despite the possibility of using the arithmetic mean strategy, the statistic test is the recommended strategy, since execution time is a very sensitive property and a pure mean strategy might not represent truly reliable values.

4.1.4. Repository Mining Phase: Current Repository Support

The last phase mines commits from the version control system for each method detected with performance deviation. If the commit changed lines inside the methods of interest, the framework searches the commit log for issue numbers, which are then used to find more accurate information about development issues in the issue tracking system.

The mining process consists on the analysis of the results of several queries to the version control and issue tracking repositories. During this process, the framework uses regular expressions to match and discover issue numbers inside commit logs. Usually, developers use a particular notation to highlight the issue numbers inside the logs. The framework also implements a parser with Eclipse JDT to statically analyze Java classes and determine their method boundaries. The parser discovers the numbers of the first and last lines of every method declared inside classes that contain at least one method with performance deviation. The number of lines are used to check if the lines that the commits affected were inside or outside methods of interest in order to determine if they have changed during the evolution or not.

Figure 20 shows a partial class diagram with the main classes of the repository mining component.

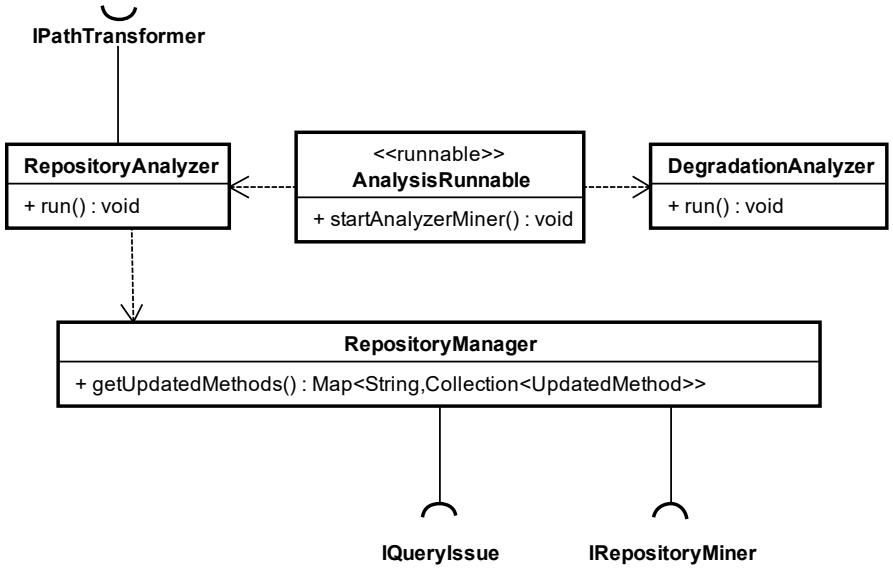


Figure 20. Partial UML class diagram for the repository mining component.

The `AnalysisRunnable` class starts and manages the repository mining process. First, it uses `DegradationAnalyzer` to execute the dynamic model comparison. It means the `DegradationAnalyzer` class is responsible for the deviation analysis on the previous phase. Second, `AnalysisRunnable` uses `RepositoryAnalyzer` class to mine data from the control version system related to classes which declare methods detected with performance deviation. Every line of the files is checked in order to discover which commit has introduced the last change between the target releases. After the commit is discovered, it uses regular expressions to match and discover development issue numbers inside the commit logs. Finally, the issue numbers are used in queries to retrieve related information, such as the issue types from the issue tracking system. The described process is executed by the `getUpdatedMethods()` method from `RepositoryManager` class. It accesses the control version system to mine a list of classes that declare methods with performance deviation. Such list comes from the previous phase (deviation analysis). `RepositoryManager` class also locates the working copies for each mined class of each release in order to automatically identify their revisions when it is necessary.

Thus, the main tasks performed in this phase are: (i) accessing the control version system to mine classes which declare methods detected with performance deviation; (ii) finding out where class are located based on the methods signature; and (iii) retrieving development issues related to commits in the issue tracking system. All of these tasks are system dependents,

and because of that, the framework defines extension points to each one of them through the interfaces `IRepositoryMiner`, `IPathTransformer`, and `IQueryIssue`, respectively.

In the first task, the interface `IRepositoryMiner` should be implemented to provide access to particular control version systems. Currently, the framework has support for Git and Subversion. In the second task, the framework needs to find out the full path location for each class that declares methods with performance deviation. It is necessary because in some systems this information cannot be extract through a simple analysis of the method signatures since packages with the same name may be organized in different folders, or even in different projects. The `IPathTransformer` implementation should map methods signatures to file (class) paths. In this case, the developer has to implement a strategy to check the full name of a class and indicate the correct path, including the package, where it belongs to. The case studies performed by this work has generated implementation of this interface for Netty, ArgoUML, Wicket, and Jetty.

Finally, the third task have to parse commit logs, which were discovered by the specific implementation of `IRepositoryMining`, to extract development issue numbers. Developers from different systems may use different notations to highlight the issue numbers which can usually be extracted by matching regular expressions. Thus, the implementation of `IQueryIssue` interface should receive a commit log and search it looking for development issue numbers. After find the numbers, it should execute a query in the issue tracking system and return the issue metadata. The framework currently provides implementations of `IQueryIssue` to work with the follow issue tracking systems: Issuezilla, IProject, Github, Bugzilla, and Jira.

To keep all collected data from the mining process organized, the framework has defined the change analysis model, which models the relationships among methods and modifications, as shown in Figure 21. In that case, an updated method (`UpdatedMethod`) – a method changed during the evolution – has its limits (`MethodLimit`) represented by its first and last lines in the source code. Each changed line (`UpdatedLine`) of an updated method have the changed source code line and the line number, and can have many commits (`Commit`). For each commit, zero or more development issues (`Issue`) might be found and a set of statistics (`CommitStat`) are calculated. In this context, removed and added methods are also considered as updated methods.

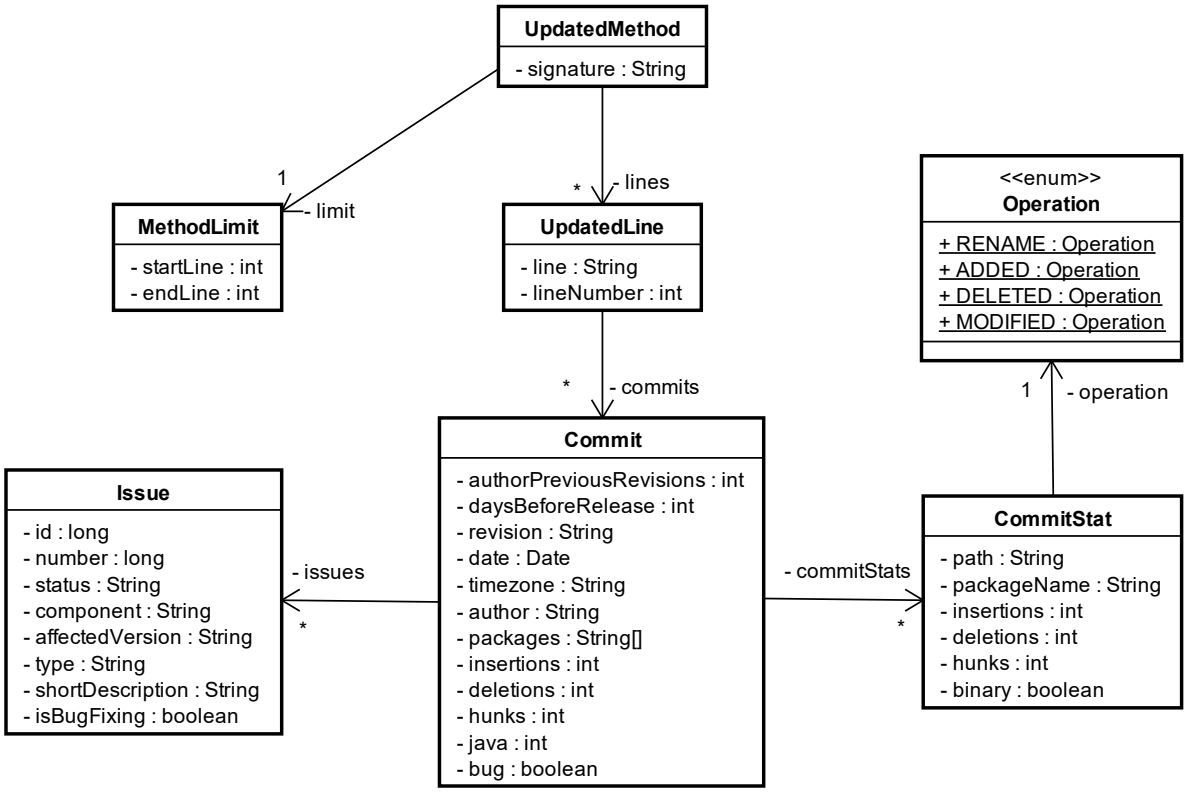


Figure 21. Partial UML class diagram for change analysis model.

Commits keep several information, including how many revisions the author has done until that point (`authorPreviouRevision`) and how many days have passed from the commit date to the release date (`daysBeforeRelease`). A set of structural information, calculated from the `CommitStat` class, is also cached to the `Commit` class, including: the commit code, the date, the timezone, the author, the list of affected packages, the number of insertions (lines), the number of deletion (lines), the number of hunks, the number of affected Java files, and if it was a bug-fixing commit.

The `Issue` class models the more common features of a development issue, including: the id or number for identification since some issue tracking systems use both, the status to indicate the current situation, the affected component and the affected version, the type of the issue, a short description, and if it is a bug fix. Some issue tracking systems may have more attributes, or may have less. The modeled attributes are enough to represent an issue in the context of the present work, making possible achieve the defined goals for the framework.

4.2. Quality Attribute Traceability Support

As explained before in Section 3.3, the secondary purpose of the approach is to provide quality attribute traceability support among scenarios, quality attributes and source code assets. Java annotations were used as source of the metadata to identify the methods that represent scenario entry points as well as the methods related to particular quality attributes or that implement design decisions that affect particular quality attributes in the source code. The static analysis relies on WALA [22] source code analysis support. Subsection 4.2.1 presents the current provided annotations, while Subsection 4.2.2 explains how the static call graph was modeled and the static analysis model class structure. As said before, due to the preliminary stage of this way of using the approach, studies about the traceability support will not be detailed here, but are available in previous publications [85] [86] [87] [88].

4.2.1. Release Preparation Phase: Java Annotations as Source of Metadata

The approach requires the system's source code and additional metadata information in order to proceed with the evaluation process. The code annotations used here are the exact same ones used in the release preparation phase presented in Subsection 4.1.1 of the performance deviation analysis support implemented by the framework and illustrated in Figure 9.

The main aim of the quality attribute annotations is to document source code methods where such quality attributes are critical. This is useful because enables traceability providing the possibility of statically determining which scenarios are affected by quality attributes or which ones have potential to contain tradeoff points. For example, if a path of a particular scenario in the static call graph is associated to more than one quality attribute, it should be carefully observed and monitored because it has potential chances to contain tradeoff points.

4.2.2. Static Analysis Phase: Static Analysis Model Structure

In this phase, the framework builds the static analysis model. It is implemented as a class structure in which the main element is the static call graph. The static analysis tool is an Eclipse plugin that allows executing the architecture evaluation over Eclipse projects. It currently parses source code from Java projects. Figure 22 shows a partial class diagram of the plugin.

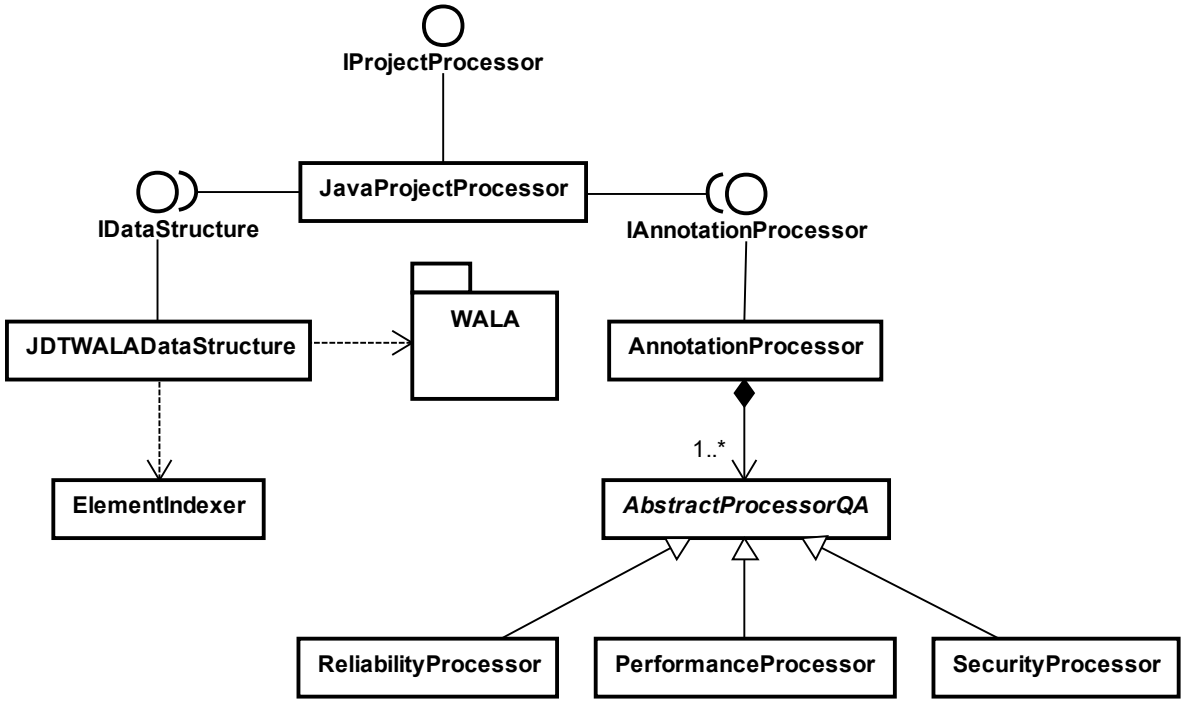


Figure 22. Partial UML class diagram for quality attribute traceability support.

The `JavaProjectProcessor` class coordinates the process of static analysis to build the necessary structures for the system under evaluation. `AnnotationProcessor` class aggregates a set of concrete strategy classes to process the different quality attribute annotations. Each one of them (`ReliabilityProcessor`, `PerformanceProcessor` and `SecurityProcessor`) extends `AbstractProcessorQA`, and are responsible for processing a particular kind of annotation. During the annotation reading, the `AnnotationProcessor` class also builds an index of the scenarios annotated to complement the data structures built previously. `JavaProjectProcessor` class also uses the `JDTWALADataStructure` to access and manipulate the system call graph. The `JDTWALADataStructure` class uses the CAST (Common Abstract Syntax Tree) front-end from WALA static analysis framework [22] to build the static call graphs of the scenarios of interest. It also uses `ElementIndexer` class to build indexes of methods, classes and annotations in order to make the access to the call graph more efficient.

The main steps of the process to build the static analysis model are shown in Figure 23. First, the `JavaProjectProcessor` uses `JDTWALADataStructure` to build the call graph and the indexes. It is important to emphasize that the call graph is assembled by WALA, which is not represented in the Figure 23. `ElementIndexer` is used to start the index creation process of methods and annotations. Then, an instance of the `AnnotationProcessor` class processes the

scenario annotations (`@Scenario`). Finally, each quality attribute annotation (`@Performance`, `@Security`, `@Reliability`) is processed by calling every concrete `AbstractProcessorQA` subclass.

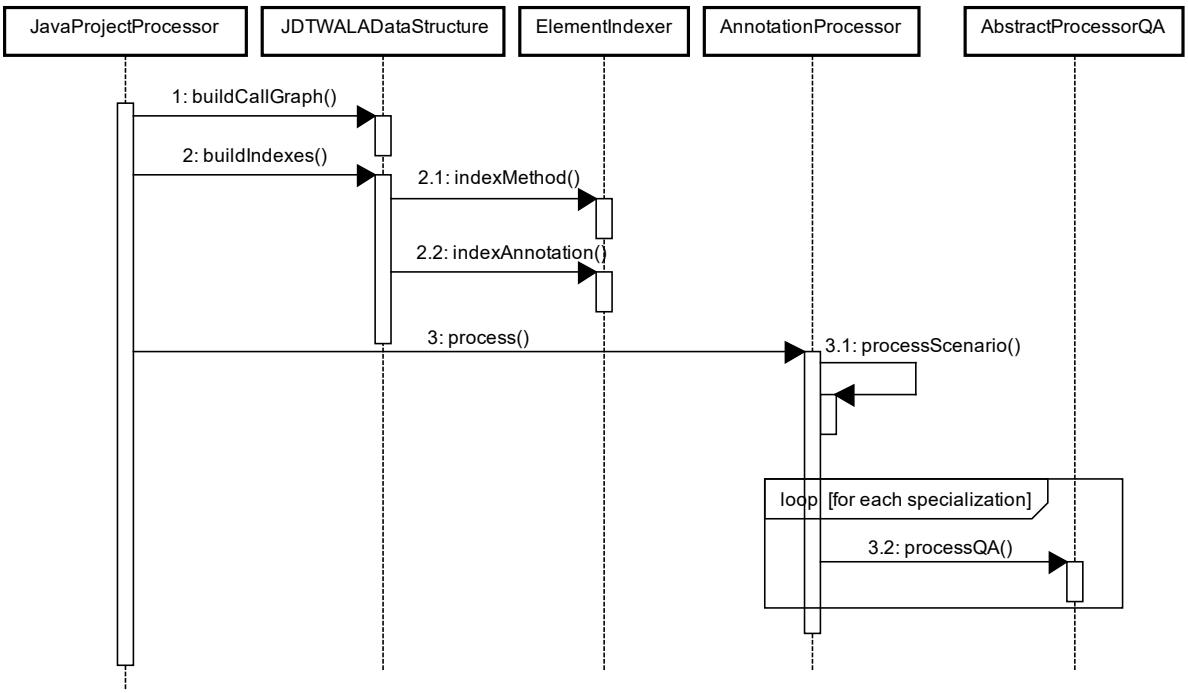


Figure 23. Partial UML sequence diagram for quality attribute traceability support.

The static analysis tool of the framework uses the static analysis model to represent the relationships among the system assets, such as classes, methods, scenarios and quality attributes. Figure 24 shows a partial class diagram of this model. The `ScenarioData` class has a starting root method, and `MethodData` has a declaring class. Each quality attribute is a specialization of the `AbstractQAData` class that keeps a reference to its related method. Finally, every `MethodData` instance has also an attribute signature that references the equivalent method node in the WALA call graph.

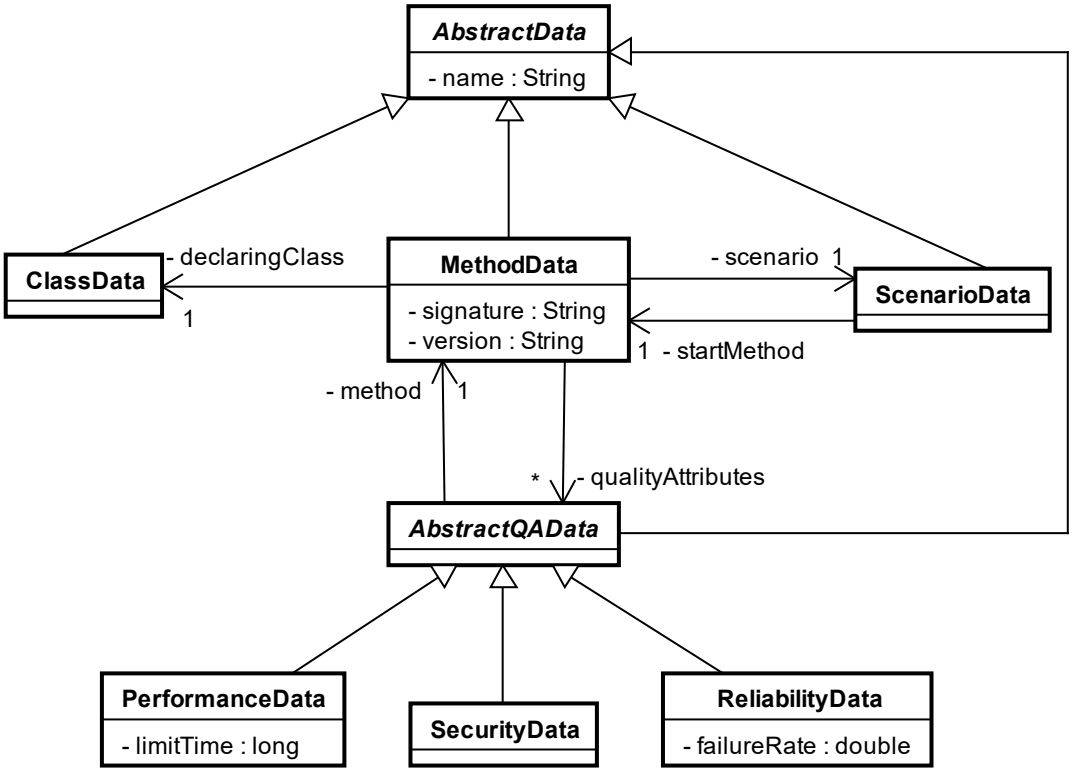


Figure 24. Partial UML class diagram for static analysis model.

The static analysis model gathers metadata information about the system and the static call graph assembled via WALA. The `MethodData` class works as a front-end to access the nodes of the static call graph and to keep WALA call graph complexities hidden from who is using the framework and analyzing the model. In this context, if the framework needs to change the static call graph generation tool by replacing WALA, the applications that use the defined static analysis model will not be affected.

5. PERFORMANCE DEVIATION ANALYSIS

This chapter describes an exploratory study for the evaluation of the performance quality attribute in terms of execution time for releases of the same system. The main aim of such study is to reveal performance degradations of scenarios architecturally relevant and their possible causes and to check the framework support for software from different domains. In addition, this chapter comes from previous published research work [89].

The rest of this chapter is organized as follows: **Section 5.1** presents how the study was designed, including the main contributions (*Subsection 5.1.1*), the goals and research questions (*Subsection 5.1.2*), the target systems (*Subsection 5.1.3*), and the methodology (*Subsection 5.1.4*); **Section 5.2** shows the results of the study; finally, **Section 5.3** concludes the chapter, presenting threats to validity and some discussion about the results.

5.1. Study Design

Three software systems from different domains were analyzed, including a large-scale web-based information system (SIGAA [27]), a client-server framework for development of network applications (Netty [28]), and an UML modeling tool (ArgoUML [29]). This study was accomplished using the proposed automated scenario-based evaluation approach for performance considering the property of execution time. Overall, the implemented framework was used to perform each phase of the approach: **(i)** first, the target scenarios were selected; **(ii)** second, the system execution was monitored using dynamic analysis; **(iii)** after that, the collected data from the dynamic analysis of two release for each analyzed software system were compared looking for degraded methods (and constructors); **(iv)** finally, the degraded elements from the previous phase were mined in their version control systems and issue tracking systems.

5.1.1. Main Contributions

The main contributions of this study are: (i) the evaluation of the proposed approach to analyze the performance degradation of system scenarios in the context of a single evolution; and (ii) the results generated by the application of the support framework to reveal performance degradations of evolution releases of existing software systems from different domains and their potential causes.

Overall, the study has found that: **(i)** 100%, 24% and 19% of the scenarios selected to be analyzed for SIGAA, Netty and ArgoUML, respectively, had performance degradation during the target evolution of each system; and **(ii)** 2, 31 and 29 development issues were pointed out to be the potential main sources of performance degradation for SIGAA, Netty and ArgoUML, respectively.

5.1.2. Goals and Research Questions

The main goal of the study was to investigate the performance degradation of architecturally relevant scenarios, considering the property of execution time by applying the proposed approach in software from different domains. It aims identifying degraded scenarios over the system evolution releases, and determine the possible causes of such degradation by indicating the responsible code assets, changes and issues.

The research questions that guided this study were elaborated to aim its defined goal, based on the exploration of the feasibility of the framework functionalities when applied to systems from different domains, and the characterization of the obtained results. They were:

RQ1. *Can the proposed approach find degraded scenarios in terms of the performance quality attribute when applied to systems from different domains?* We use the approach to detect and investigate existing system scenarios that exhibited degradation of the performance quality attribute through the comparison of the execution time from different releases of systems in three different domains: web information system, development framework and desktop tool.

RQ2. *Can the mining process improve the discovering of degraded methods resulted from the dynamic analysis?* The deviation analysis stage of our approach detects many methods responsible for the degradation of the system scenarios. We are going to analyze how the mining of the changed classes and methods from software repositories can help us to reduce the amount of degraded methods indicated by the dynamic analysis and deviation analysis phases.

RQ3. *Are there specific modules responsible for the majority of the performance degradations?* We also characterize some results in order to better understand them. Thus, we calculated how many methods identified as responsible for performance degradation of the target scenarios were found in each system module. By using such information, developers could be aware of which modules may represent potential risks related to performance degradation.

RQ4. *Which were the most common kinds of issues associated to performance degradation in this study?* Last, but not least, the study also investigates the kinds of issues that are responsible for the degradations in the studied systems.

5.1.3. Target Systems

The evaluated software systems were an enterprise large-scale web-based system (SIGAA) and two open source systems (ArgoUML and Netty). Different systems were chosen in order to analyze and evaluate the approach application in different domains and scenarios. Table 1 gives an overview of the target systems, including the system domain, the number of lines of code per release, the initial and final version number of each analyzed release, and the time frame between them. An interesting observation is that a longer time frame did not imply in more or less degradation necessarily. It will depend on the intensity of changes to the source code over the analyzed period.

Table 1. Target systems for performance deviation analysis.

Property	Target Systems		
	SIGAA	**ArgoUML**	**Netty**
System Domain	Web information system	Tool for UML modeling	Framework for network applications
Lines of Code	673,610 701,257	160,066 156,788	69,281 82,466
Release Version	3.11.24 3.12.18	0.29.4 0.34	4.0.0.Final 4.0.19.Final
Repository Date	August 2013 December 2013	January 2010 December 2011	July 2013 April 2014
Code Repository	Subversion	Subversion	Git
Issue Track	IProject (Proprietary)	Issuezilla	GitHub

The web system for academic management (SIGAA) was developed by using Java mainstream technologies. Different customizations of this are currently used by more than 20 universities and institutes in Brazil. A layered architecture is used to structure the Java implementation of the system, including the layers of *Graphical User Interface*, *Service*, *Business*, and *Data*.

The two investigated open-source systems were: ArgoUML and Netty. The first is an open source UML modeling tool. It runs on any Java platform and is available in ten languages. It has been downloaded more than 80,000 times. The latter is a NIO (Non-Blocking I/O) client server framework and tooling for the rapid development of network applications such as protocol servers and clients. It also simplifies and streamlines network programming.

5.1.4. Mining and Assessment Procedures

Two computers were used to run system and execute the dynamic analysis phase of the approach. The PC1 was reserved for SIGAA evaluation, while the PC2 was reserved for ArgoUML and Netty evaluation. They had the configuration described below:

PC1. An Intel Core i7 processor with 16GB of RAM memory, running Microsoft Windows 7 operating system, Oracle Java version 7, and JBoss server application 4;

PC2. An AMD Phenom II processor with 8GB of RAM memory, running Microsoft Windows 7 operating system, and Oracle Java version 7.

The first step in the study was to check out every release of the target systems from their repositories and to configure the projects to correctly compile and run, because the dynamic analysis phase requires the execution of the system. Afterwards, we selected common scenarios considering each pair of releases to analyze. Different strategies were used to exercise the scenarios on each system. For SIGAA, six scenarios from the library management module were selected and annotated with `@Scenario`. The SIGAA's library module has been selected because we had support from one of its main developers. Thus, the target scenarios and their entry points in the source code for SIGAA were indicated by the developer during a meeting. ArgoUML and Netty use JUnit3 and JUnit4 tests, respectively, so their scenarios were exercised by running some selected tests from their test suite. The selection included tests without random inputs and preferably those which cover functionalities and not only a single method. In this case, the abstract aspect (Figure 12) was extend for two concrete implementations in order to consider entry points as JUnit 3 tests (Figure 13) and entry points as JUnit 4 tests (methods annotated with `@Test` – Figure 15).

The reduced number of selected scenarios for SIGAA – six scenarios – was caused due to the absence of automated tests in this system what increased the complexity and difficulty to execute its scenarios. In this case, we had to manually create *scripts* by using the JMeter tool [53] to execute them. Thus, the selected scenarios for SIGAA were:

- *Simple Library Search* performs a simple search in the library catalogue matching simple key words;
- *Generating Discharge Declaration* generates a document which declares the user does not have pendency in order to disable the user's library account;
- *Performing Loan* allows users borrow items from the library;

- *Renewing Loan* allows the users renewing a loan previously performed;
- *Returning Loan* allows users returning library items;
- *Check User Status* verifies if the user is free of pendency.

After that, the dynamic analysis was executed on the same computer for each release in the exact same conditions and with all non-essential services disabled (e.g., updates, antivirus, indexing services, and virtual memory). Thus, the automated test suite of each release was repeated ten times in the same conditions for each system in order to have a more accurate average of the performance quality attribute. SIGAA was an exception because the system does not provide automated tests. To work around, a series of manual testing requests to the web system were created and executed to exercise the chosen scenarios. These requests were then recorded and automatic repeated ten times using JMeter tool [53].

During the study, the test suite was executed to monitor, collect and store application execution data using dynamic analysis. After that, the execution data of each old release was compared against the data of the new release, generating the sets of degraded methods for the performance evaluation. For this particular study, the deviation analysis phase has used the arithmetic mean strategy (Subsections 3.2.3 and 4.1.3) to compare the execution time of the releases. Thus, the default threshold of 5% was applied, it means that scenarios and their methods are considered degraded when the execution time increases more than 5% from the old release to the new one. Finally, the version control system and the issue tracking system for each system were mined to find commits and development issues responsible for introducing changes in the scenarios detected with performance degradation.

5.2. Study Results

In this section, the results of the application of the approach to the target systems are presented and discussed. The discussion is organized in terms of the proposed research questions (Subsection 5.1.2).

5.2.1. Degraded Scenarios

(RQ1) Can the proposed approach find degraded scenarios in terms of the performance quality attribute when applied to systems from different domains? Table 2 presents the total of analyzed and degraded scenarios for the three investigated systems. As we can see, all six investigated scenarios of SIGAA exhibited performance degradation. For Netty, the analysis investigated 21 scenarios and five of them, representing 24% of the total, were

considered as degraded. Finally, 63 scenarios were investigated for the ArgoUML, and the analysis detected 12 of them (19%) with performance degradation.

Table 2. Total of analyzed and degraded scenarios.

Number of Scenarios	Target Systems		
	SIGAA	Netty	ArgoUML
Analyzed:	6	21	63
Degraded:	6 (100%)	5 (24%)	12 (19%)

The individual scenarios and how they have degraded during the evolution for SIGAA, ArgoUML and Netty are shown in the graphics illustrated by Figure 25, Figure 26 and Figure 27, respectively. The study found performance degradation in scenarios for all the three systems. Some of the scenarios presented a degradation of more than 100% when comparing the new and old releases of the same system.

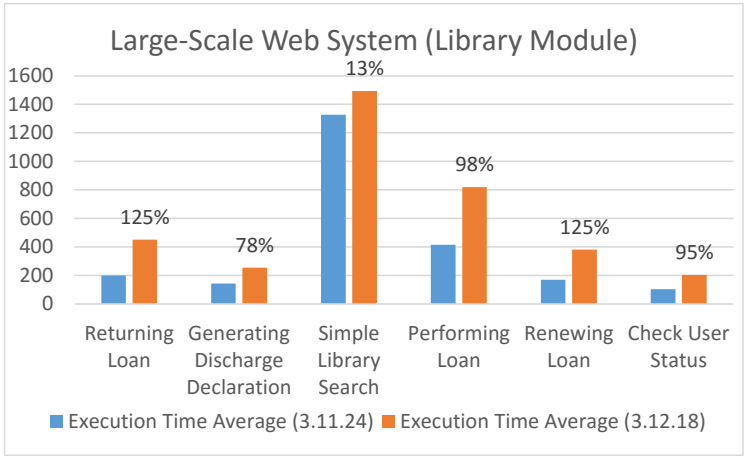


Figure 25. Degraded scenarios of the SIGAA web system.

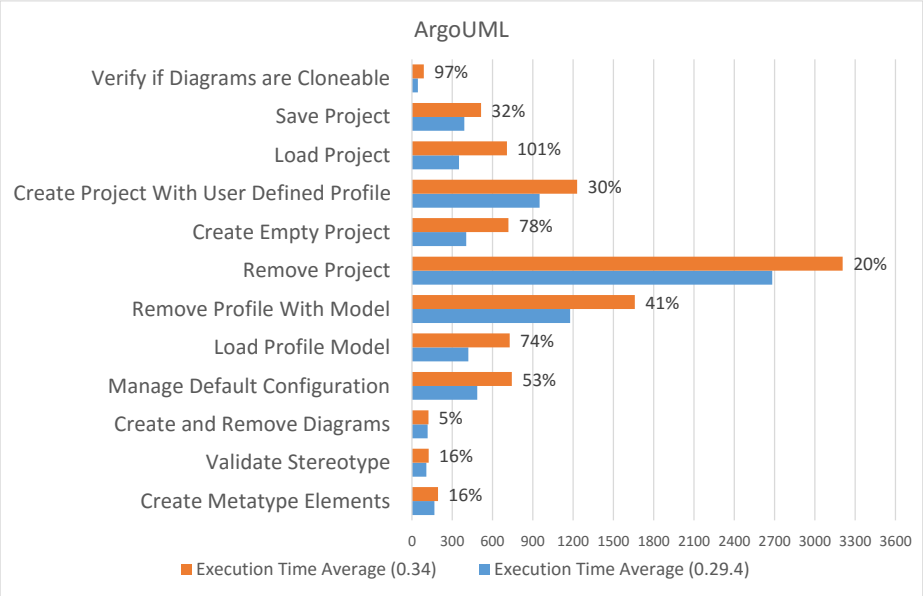


Figure 26. Degraded scenarios of the ArgoUML tool.

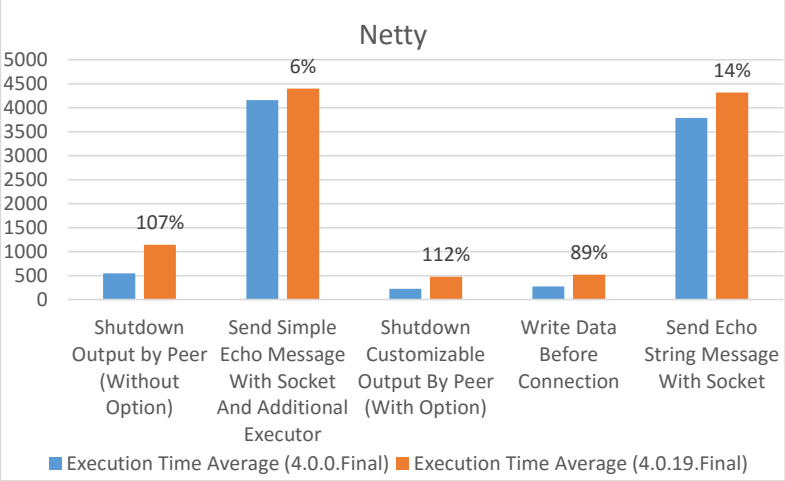


Figure 27. Degraded scenarios of the Netty framework.

Figure 25 shows, for example, that the *Returning Loan* and *Renewing Loan* scenarios had an impact of 125% in their execution times while *Performing Loan* degraded 98%. In Figure 26, we can see that the ArgoUML project has four scenarios with performance degradation superior to 74%. For Figure 27, we can see that *Shutdown Output by Peer (Without Option)* and *Shutdown Customizable Output By Peer (With Option)* scenarios of the Netty framework degraded 107% and 112%, respectively.

The code assets and modifications introduced to the performance-degraded scenarios are going to be indicated in the answer of the next research question. The results of Table 2, Figure 25, Figure 26, and Figure 27 show that the framework was able to manage the domain specificities in the cases of SIGAA, ArgoUML and Netty.

5.2.2. Filtering Degraded and Changed Assets

(RQ2) Can the mining process improve the discovering of degraded methods resulted from the dynamic analysis? Table 3 shows the total of degraded scenarios and the total of degraded methods. The numbers 229, 349 and 1181 indicates affected methods for SIGAA, Netty and ArgoUML, respectively.

The repository mining phase has contributed to reduce considerably the amount of degraded methods to be analyzed by indicating only the methods that have been modified between the two releases of each system. Thus, after the mining phase, we found 9, 100 and 230 degraded methods, respectively. Finally, we selected only the performance-degraded methods related to the scenarios of interest which were also degraded. It has reduced even more the final amount of methods to 9, 26 and 153, respectively. Table 3 shows these results in the last line.

Table 3. Performance degradation of scenarios and methods.

Property	Target System		
	SIGAA	**Netty**	**ArgoUML**
Total of Degraded Scenarios:	6	5	12
Degraded Methods Found Before Mining:	229 (100%)	349 (100%)	1181 (100%)
Degraded Methods Found After Mining:	9 (4%)	100 (29%)	230 (19%)
Degraded Methods of Degraded Scenarios:	9 (4%)	26 (7%)	153 (12%)

The mining phase helps to decrease the number of performance-degraded methods indicated by the deviation analysis phase. For example, one of the 229 methods found in the web system before mining was `QueryLoanDAO.findPersonInfoNoLink()`. This method was found by the deviation analysis because it has increased more than 5% its execution time. However, it was filtered from the final result because the mining phase detected it has not changed. Indeed, it has increased its execution time because it depends on the method `SessionLogger.registerCaller()`, which was introduced during the system evolution and represents the real cause of the degradation, as we will explain on the next research questions. Thus, the framework has decreased the amount of potential methods responsible for the degradation by mining the degraded set of methods from the deviation analysis phase, and selecting only those ones which were added or changed during the evolution and belong to degraded scenarios.

As result, the set of methods had a reduction of 96% (from 229 to 9) in the number of the degraded methods for SIGAA. Netty had a first reduction of 71% (from 349 to 100) after mining and a second reduction of 93% (from 349 to 26) after matching methods related to degraded scenarios. For ArgoUML, the reductions were of 81% (from 1181 to 230) and of 88% (from 1181 to 153). The conclusion is that the integration of dynamic analysis and repository mining really contributed to reduce the amount of degraded methods for the investigated systems, by providing a more optimized solution than using only one of these techniques. Thus, developers will have a small set of data to analyze and understand.

5.2.3. Modules of the Target Systems

(RQ3) Are there specific modules responsible for the majority of the performance degradations? The study found 9, 26 and 153 methods (Table 2) responsible for performance degradation distributed in 7, 18 and 84 classes, respectively, for SIGAA, Netty and ArgoUML.

The analysis shows that the distribution is uniform and there are no predominant classes with a significant number of blamed methods. For example, the class that concentrates more degraded method (10 methods) is `UserDefinedProfile` from `org.argouml.profile` package in ArgoUML. Table 4 shows every package for each system that contains degraded and changed methods. The complete list of classes is available in Appendix A and online [54].

Table 4. Methods potentially responsible for degradation over the system's packages.

System	Package Prefix	Number of Methods	Sum of the Execution Time (ms)
SIGAA	`websystem.core.dao`	2	12.87
	`websystem.core.security`	1	5.31
	`websystem.core.util`	2	1.03
	`websystem.sig.library`	3	398.29
	`websystem.sig.domain`	1	0.09
Netty	`io.netty.buffer`	3	1.58
	`io.netty.channel`	23	4.28
ArgoUML	`org.argouml.application`	2	0.27
	`org.argouml.i18n`	1	0.11
	`org.argouml.kernel`	5	26.44
	`org.argouml.model`	43	1680.24
	`org.argouml.notation`	27	19.38
	`org.argouml.persistence`	15	1753.69
	`org.argouml.profile`	23	1781.14
	`org.argouml.uml`	36	19,62
	`org.argouml.util`	1	0.01

The affected packages in Table 4 indicates the total number of methods inside each package and the sum of all degraded execution times. For example, ArgoUML had 43 changed and degraded methods inside classes of the `org.argouml.model` package, and the sum of the increase of the average execution time for all 43 methods was 1680.24 milliseconds.

Most of the blamed methods of Netty belong to the `br.netty.channel` package (Table 4) because, in the fourth version, many classes under this package have gone through a major overhaul. The channel package contains the core API to handle channels and to abstract the communication. A small performance variation in the methods of the classes from this package might cause a considerable impact because this API is used in many situation and several times during a communication. The sum of the average execution time of the degraded and changed methods of the `oi.netty.channel` package was 4.28 milliseconds. It means that this package has caused more impact than the `io.netty.buffer` package with 1.58 milliseconds. The analysis also shows that the `NioEventLoop.openSelector()` method from the io.netty.channel package was the main responsible for the degradation.

For ArgoUML, the kernel module of the system contains only five degraded methods, while the other packages, such as, notation, model, persistence, uml and profile have concentrated most of the changes because they are related to model manipulation functionalities. Table 4 shows that the model, persistence and profile packages have caused considerable impact to the performance. The methods that exhibited more critical execution times were `XmiReferenceResolverImpl.toURL()`, `UserDefinedProfile.getAllCritiquesInModel()` and `UmlFilePersister.doLoad()`, respectively, from the model, profile and persistence packages. ArgoUML had many development issues associated to bugs in scenarios related to drawing elements in diagrams, creating panels, loading profiles and many others related to UML2 issues.

The analysis of degraded methods for SIGAA shows five methods from the core package, three from the library module, and one from the domain package. Analyzing the code evolution of those methods, we identified that the main change responsible for the degradation was a new log strategy that implements an auditing service for the system. It generates logs of the execution stack traces for all the modules that use the common core. The strategy has affected the library module and all analyzed scenarios. The results were discussed with one of the main developers from the library module in order to validate them.

5.2.4. Types of Issues

(RQ4) Which were the most common kinds of issues associated to performance degradation in this study? Mining development issues related to the changes applied to the degraded methods is useful because it can help the development teams to understand which high-level changes related to source code affect existing scenarios. In order to obtain an initial understanding of the causes of the performance degradations, a manual inspection of the issues discovered by the framework was conducted.

For SIGAA, we found two main issues related to the code changes during the evolution. The first one was a bug-fixing issue. The `LibraryServiceUtil` class from the business layer has been updated to optimize the calculation of the extra days to return book loans by reducing the database accesses. Those changes affected the *Performing Loan* and *Renewing Loan* scenarios, but they were not the main cause of the degradation. Despite this attempt of optimization, the performance decreased because of code changes related to the second issue which was an improvement. The `GenericDAOImpl.getSession()` method introduced a call to the new method `SessionLogger.registerCaller()` which caused several

performance degradations for all the investigated scenarios. The `SessionLogger.registerCaller()` method is responsible for implementing an auditing service in the system. It is called many times during the execution of the selected scenarios. The implementation of this method calls the `UtilService.stackTraceInvoker()` method to store the stack traces of the invoker.

The analysis of the Netty framework found 13 bug-fixing, 9 improvements, 1 new feature, and 8 unlabeled issues. The last ones are issues that were not associated to a specific type. Again, some manual inspection of this issues was accomplished to try to understand important modifications. A unlabeled issue has changed the `PoolThreadLocalCache` class from the `io.netty.buffer` package to check periodically if threads are still alive. The motivation was to release memory when threads are not alive anymore even for short living threads. It added an extra background verification that contributed to degrade some scenarios, for example, those ones to shutdown output by peer and sending messages.

Another interesting change in Netty, added by a bug-fixing issue, was the modifications to the `DefaultChannelHandlerContext` class from package `io.netty.channel`. It has added several new method calls to perform validations in order to avoid errors when a communication channel is canceled. They have contributed to degrade some scenarios, including sending messages with sockets and writing data before connection.

Finally, new feature issue that causes performance degradation was also found during the Netty analysis. This issue has changed the way to create new sockets in the `NioSocketChannel`, `NioServerSocketChannel` and `NioDatagramChannel` classes. After that, when creating a socket, developer can indicate its own concrete provider for selectable channel by using a `SelectorProvider` class that implements the `openSelector()` method. This modification has impacted both scenarios of sending messages with sockets and shutdown channels.

For ArgoUML, our mining found 4 improvements and 25 bug-fixing issues. Two improvements issues affected mainly the notation and model packages. For example, `NotationProvider` class manages the string rendering on diagrams and was changed to modify the way the provider communicates with classes responsible for drawing diagrams. Before that, these classes had to listen their providers. Now, they receive a new string from their providers when a rendering update is needed. It has simplified and clarified the implementation, but the

modifications have replaced an asynchronous strategy for a blocking call. Thus, these issues have contributed for the performance degradation of scenarios related to diagram and project operations, for example, removing projects and verifying if a diagram is cloneable.

An interesting bug-fixing issue was created to correct a bug in the profile module when users create a profile that depends on another profile. According to this issue, "ArgoUML fails to start after creating dependency between two user profiles". This issue was initially corrected, but afterwards, another bug-fixing issue was added and related to the first one. According to a developer comment, there was a design flaw in the initialization process, since it was accessing the `ProfileManager` class which has not been initialized. The solution has performed several modifications, including the new `DependencyResolver` class which implements a new state-full algorithm to resolve dependencies of user defined profiles. It has affected scenarios related to profile management, including the scenarios of loading and removing projects and profiles.

Table 5 summarizes the kind of issues found. As we can see, bug-fixing was the predominant kind of issue that caused performance degradation of scenarios in this particular study. However, the results cannot be generalized for other systems and releases than the ones investigated by the study. The low number of issues for SIGAA is justified by the reduced number of changes in the analyzed releases.

Table 5. Issue types discovered during the performance deviation analysis.

Issue Type	Target System			Total Number
	SIGAA	Netty	ArgoUML	
Improvement	1	9	4	14
Bug-fixing	1	13	25	39
New Feature	0	1	0	1
Unlabeled	0	8	0	8
TOTAL SUM:	**2**	**31**	**29**	**62**

5.3. Conclusion

This chapter presented a study which has analyzed the performance deviation, considering the property of execution time, for two releases of three different software systems from different domains. The study allowed us to determine the possible causes of the scenario degradations for performance.

About the degraded scenarios (RQ1 and Table 2), all six selected scenarios from SIGAA exhibited performance degradation. For Netty, from a total of 21 scenarios, five of them (25%), were considered as degraded. Finally, from 63 investigated scenarios for ArgoUML, the

analysis detected 12 of them (19%) with performance degradation. In addition, we could observe that the mining process has reduced significantly the amount of methods which developers should consider (RQ2 and Table 3). Such methods are distributed in 7, 18 and 84 classes, respectively, from SIGAA, Netty and ArgoUML, and in a few packages (RQ3 and Table 4). The type of issue with more occurrences (RQ4 and Table 5) was bug fixing (39 issues), followed by enhancement (14 issues). Finally, on the next subsections of this chapter, some threats to validity we have identified during this study are discussed.

5.3.1. Impact of Instrumentation

The instrumentation process causes a threat related to the measurement strategy. It needs to intercept the methods during their execution, which might affect the execution by contributing to increase the original execution time. We have not measured the impact that instrumentation causes to execution time in this context, but we believe it does not form a problem since our analysis compares pairs of values of execution time from two releases, and the instrumentation should cause the same or very similar increases in both values.

5.3.2. Rename Problem

The framework currently considers renamed code elements as new elements that will potentially affect the performance of scenarios. However, manual inspection of the results conducted for each framework application showed that, for these cases, names were consistent in general. As an alternative solution, we could also highlight these methods to present them to the developers, and they could indicate which of such methods might not be new. This problem could be especially problematic for refactoring because the approach will indicate many changes. Thus, in such cases, developers should already expect that behavior when analyzing refactoring.

5.3.3. Generalization and Limitation of Results

Some results of our study cannot be generalized to other systems, or even to other releases of the target systems. For example, we tried to characterize the types of development issues that are most likely to contribute to performance deviation in the evolutionary analysis (Table 5). The results showed it was bug-fixing, but it is still a small amount of issues to consider. Defining the type of the issues is also optional in most of the issue tracking systems.

6. EVOLUTIONARY ANALYSIS

This chapter presents a more detailed analysis compared to the last one. We conducted an empirical study to analyze multiple evolutions of three different network/web application frameworks with the main aim of evaluating the capacity of the approach and support framework to automate the performance deviation analysis over multiple evolutions of existing software systems in order to identify performance issues, and to check if the developers of these systems were aware of such issues. In addition, this chapter comes from previous published research work [90].

The rest of this chapter is organized as follows: **Section 6.1** presents how the study was designed, including the main contributions (*Subsection 6.1.1*), the goals and research questions (*Subsection 6.1.2*), the target systems and the study procedures (*Subsection 6.1.3*); **Section 6.2** shows the results of the study; finally, **Section 6.3** concludes the chapter, presenting threats to validity and some discussion about the results.

6.1. Study Design

The study analyzed a total 21 releases from Netty [28], Wicket [30], and Jetty [31], being seven releases for each system, considering 57 scenarios in total. In addition, we collected feedback about the study results from eight developers of these systems by an online survey. Overall, the framework was applied to execute the approach phases: **(i)** first, the target scenarios were selected by reusing JUnit 4 tests; **(ii)** second, each release was executed and monitored using dynamic analysis, resulting in multiple databases; **(iii)** after that, the collected data from the dynamic analysis was compared by grouping subsequent releases for each analyzed software system; **(iv)** finally, the elements with performance deviation from the previous phase were mined in their version control systems and issue tracking systems. Then, the online survey was applied. We sent the questionnaire to every developer with an available e-mail address registered in the contributor page of each system. From these contributors, eight of them have answered our survey with some feedback.

6.1.1. Main Contributions

The main contributions are: (i) the identification of the potential causes of performance deviation for Netty, Wicket and Jetty extracted from an evolutionary study through multiple

releases; and (ii) a preliminary evaluation of the results with eight contributors of Netty, Wicket and Jetty. Overall, the approach application automatically identified 14 scenarios with significant performance deviation for Netty, 13 for Wicket, and 9 for Jetty, almost all of which could be attributed to a source code change. When developers were asked whether they had already been aware of the performance deviations (degradation or optimization) identified, most of the eight developers, which answered our online survey, indicated that this was not the case. This preliminary evidence suggests that the proposed approach is able to identify performance deviations that developers are not aware of.

6.1.2. Goals and Research Questions

The goal of this study was to assess the capacity of the approach to identify performance issues over multiple evolutions of existing systems and to check if the developers of these systems were aware of the issues. By using the approach, developers can be aware of performance issues and reduce them before distribution or deployment of new system releases. Methods and commits related to performance deviation that are discovered by it represent guidelines and recommendations that can be used to improve the system performance. The study was guided by two research questions.

RQ1. *Can the proposed approach find corresponding source code changes in the scenarios with performance deviation?* We expect that the approach will be able to identify performance issues over multiple releases of the target system and discover their sources. It is important to understand which modifications lead to performance problems and how they could be fixed or optimized. In order to characterize these results, we also identified modules of the system (packages and classes) that concentrated most of the sources and the types of the development issues (bug/defect, new feature, improvement and others) that are most likely to cause performance deviation in the target systems. This information is useful for development teams since it will tell them what aspects of a system they should pay particular attention to when evolving the implementation.

RQ2. *Are the developers aware of the performance issues that the approach was able to find?* We would like to verify if developers were aware of the performance issues that the approach for performance deviation analysis has found. In order to answer this research question, the feedback from eight developers was collected through online surveys for each target system.

6.1.3. Target Systems and Procedures

Netty is an open-source asynchronous event-driven network application framework for rapid development. Seven releases of the fourth version of Netty were chosen, because it was the latest stable version when this study began. Thus, the first and last releases at that time were selected, respectively, 4.0.0.Final and 4.0.21.Final. Each intermediate release was chosen after manually analyzing the release notes, attempting to identify stable releases that concentrated more significant changes. The seven selected releases for Netty framework were 4.0.0.Final, 4.0.6.Final, 4.0.10.Final, 4.0.15.Final, 4.0.17.Final, 4.0.18.Final, and 4.0.21.Final.

Wicket is an open-source web application framework developed by the Apache Foundation. For this system seven releases, which represented the last ones at the time the study began, were also selected. The selected releases were: 6.15.0, 6.16.0, 6.17.0, 6.18.0, 7.0.0-M1, 7.0.0-M2 and 7.0.0-M4. We did not include the release 7.0.0-M3 because we were unable to execute it due to compilation issues. Jetty is an open-source framework that provides a web server and a Java servlet container. It is part of the Eclipse Project. The seven selected releases of Jetty were: 9.2.6, 9.2.7, 9.2.8, 9.2.9, 9.2.10, 9.3.0.M0 and 9.3.0.M1.

In order to run the framework on the target systems we have instantiated it to consider the specific version control systems and issue tracking systems that the target software systems work with. All analyzed systems work with the Git version control system. On the other hand, they use different issue tracking systems: GitHub (Netty), Jira (Wicket), and Bugzilla (Jetty).

For the preparation phase, the existing automated tests of each system were selected as scenarios. These selected tests cover important functionalities of the system and were used to reproduce their execution over different releases. In this case, the test cases were considered as entry points of scenarios and the results were grouped by test classes, since they exercise similar functionalities. Said that, it is important to emphasize we are not interested in performance deviations caused by changes in test packages because we want to reveal performance issues caused by the evolution of the application source code and not just because the tests have changed. In order to do that, the framework uses a keyword for excluding certain classes or packages which were applied to filter eventual modifications in the test cases.

The target projects were configured to support AspectJ features and to include the framework libraries, but without any source code modification. In order to accomplish the first phase of the approach, the `@Test` annotation from JUnit 4 was reused as scenario entry point

annotation, and the corresponding concrete aspect (see Figure 15) was applied posteriorly during the dynamic analysis. Thus, their scenarios were exercised by running some selected tests from their test suite. The selection included tests without random inputs and preferably those which cover functionalities and not only a single method.

The dynamic analysis phase was executed on the same computer for all releases in the exact same conditions and with all non-essential services disabled (e.g., updates, antivirus, indexing services, and virtual memory). The computer was an AMD Phenom II with 8GB of RAM memory, running the Microsoft Windows 7 operating system and Java version 7. The test suite of each release was executed ten times for Netty and Wicket, i.e., each target scenario was executed ten times. On the other hand, for Jetty, the test suite was executed 30 times. We decided to execute Jetty's test more times because most of the tests are shorter compared to the other systems. Thus, it helps to get a more accurate performance measure in terms of execution time by increasing the number of repetitions.

After that, the seven releases were grouped in six pairs of evolutions for each system to execute the third and fourth phases of the approach. Thus, the releases were organized as below:

- Netty releases: 4.0.0.Final, 4.0.6.Final, 4.0.10.Final, 4.0.15.Final, 4.0.17.Final, 4.0.18.Final and 4.0.21.Final
    - Netty Evolution 1 (NE1): From 4.0.0.Final to 4.0.6.Final
    - Netty Evolution 2 (NE2): From 4.0.6.Final to 4.0.10.Final
    - Netty Evolution 3 (NE3): From 4.0.10.Final to 4.0.15.Final
    - Netty Evolution 4 (NE4): From 4.0.15.Final to 4.0.17.Final
    - Netty Evolution 5 (NE5): From 4.0.17.Final to 4.0.18.Final
    - Netty Evolution 6 (NE6): From 4.0.18.Final to 4.0.21.Final
- Wicket releases: 6.15.0, 6.16.0, 6.17.0, 6.18.0, 7.0.0-M1, 7.0.0-M2 and 7.0.0-M4
    - Wicket Evolution 1 (WE1): From 6.15.0 to 6.16.0
    - Wicket Evolution 2 (WE2): From 6.16.0 to 6.17.0
    - Wicket Evolution 3 (WE3): From 6.17.0 to 6.18.0
    - Wicket Evolution 4 (WE4): From 6.18.0 to 6.19.0
    - Wicket Evolution 5 (WE5): From 7.0.0-M1 to 7.0.0-M2
    - Wicket Evolution 6 (WE6): From 7.0.0-M2 to 7.0.0-M4
- Jetty releases: 9.2.6, 9.2.7, 9.2.8, 9.2.9, 9.2.10, 9.3.0.M0 and 9.3.0.M1

- Jetty Evolution 1 (JE1): From 9.2.6 to 9.2.7
- Jetty Evolution 2 (JE2): From 9.2.7 to 9.2.8
- Jetty Evolution 3 (JE3): From 9.2.8 to 9.2.9
- Jetty Evolution 4 (JE4): From 9.2.9 to 9.2.10
- Jetty Evolution 5 (JE5): From 9.2.10 to 9.3.0.M0
- Jetty Evolution 6 (JE6): From 9.3.0.M0 to 9.3.0.M1

In this study, the statistic test strategy was applied instead of the arithmetic mean for comparison (Subsections 3.2.3 and 4.1.3). As *p-value* for the U-Test, a default significance level (*alpha*) of 0.05 was applied. Finally, after conclude the framework application, we conducted an inspection of the results to get a better understanding of them and to answer the defined research questions.

6.2. Study Results

This section presents and discusses the study results and how the defined research questions were addressed. First, an overview of the selected scenarios detected with performance deviation is presented (Subsection 6.2.1) as well as the changes which introduced modifications to them (Subsection 6.2.2). After that, a characterization of the main results was accomplished (Subsection 6.2.3), and, finally, the feedback from the online applied survey is discussed (Subsection 6.2.4).

6.2.1. Performance Deviation of Scenarios

RQ1. Can the proposed approach find corresponding source code changes in the scenarios with performance deviation? The approach application has identified 32 scenarios, out of 57 (56%), with performance deviation and corresponding source code changes. Table 6, Table 7, and Table 8 summarize the results. Evolutions were numbered from one to six (one evolution between each subsequent release) for each system.

Table 6. Degraded and optimized scenarios of Netty.

Scenarios (Test Classes)	NE1	NE2	NE3	NE4	NE5	NE6
Entry Point for DatagramUnicastTest			↑		↓	↑
Entry Point for SocketBufReleaseTest		↓L	↑	↑L	↓	
Entry Point for SocketCancelWriteTest					↓L	↑L
Entry point for SocketConnectionAttemptTest				↑L	↓L	
Entry Point for SocketEchoTest		↑L	↑	↑L	↓L	↑L
Entry Point for SocketFileRegionTest		↑L	↑	↑L	↓L	
Entry Point for SocketFixedLengthEchoTest			↑	↑L	↓L	
Entry Point for SocketObjectEchoTest		↑L	↑	↑L	↓L	

Entry Point for SocketShutdownOutputByPeerTest	↓L	↑L	↑	↑L	↓L	
Entry Point for SocketShutdownOutputBySelfTest		↑L	↑		↓L	
Entry Point for SocketSpdyEchoTest		↑L	↑	↓T	↓L	↓
Entry Point for SocketSslEchoTest	↑	↓	↑	↑L	↓	↓
Entry Point for SocketStartTlsTest		↑	↑	↑L	↓L	↓
Entry Point for SocketStringEchoTest	↑L		↑	↑L	↓L	↓
Entry point for UDTClientServerConnectionTest		↑				
Entry Point for WriteBeforeRegisteredTest		↑L	↑	↑L	↓L	

Table 7. Degraded and optimized scenarios of Wicket.

Scenarios (Test Classes)	WE1	WE2	WE3	WE4	WE5	WE6
Entry point for AjaxTest				↑L		↓
Entry point for Compref Test				↑	↑L	↓
Entry point for EncodingTest		↑L		↑		↓
Entry point for FormInputTest				↑		
Entry point for GuestbookTest						↓
Entry point for HangManTest	↑L			↑		↓
Entry point for HelloWorldTest		↓L				↓
Entry point for ImagesTest						↓
Entry point for LibraryTest						↓
Entry point for LinkomaticTest				↑		↓
Entry point for NiceUrlTest		↑L	↓L	↑		↓
Entry point for Signin2Test						↓
Entry point for TemplateTest						↓
Entry point for WordGeneratorTest				↓T		

Table 8. Degraded and optimized scenarios of Jetty.

Scenarios (Test Classes)	JE1	JE2	JE3	JE4	JE5	JE6
Entry point for AsyncContextListenersTest					↑	
Entry point for AsyncContextTest		↓L	↑L	↓L	↑L	↓L
Entry point for AsyncIOServletTest		↓L		↓L	↑	↑L
Entry point for AsyncServletLongPollTest					↑	
Entry point for AsyncServletTest				↑L	↑L	↑L
Entry point for DefaultServletRangesTest		↑L			↑	
Entry point for DefaultServletTest					↑L	↑T
Entry point for DispatcherForwardTest				↓L	↑	
Entry point for DispatcherTest		↑L	↓L	↓L	↑L	↑L
Entry point for ErrorPageTest			↑L		↑	↓L
Entry point for InvokerTest					↑	
Entry point for RequestHeadersTest					↑L	
Entry point for ResponseHeadersTest					↑L	
Entry point for ServletContextHandlerTest					↑	↓L
Entry point for ServletHandlerTest		↑L		↑L	↓L	
Entry point for SSLAsyncIOServletTest					↑	

For a specific scenario and evolution, an upward pointing arrow indicates an increase in execution time, while a downward pointing arrow indicates a decrease, but not necessary a significant degradation or optimization, respectively. In order to highlight the greater deviations, we have defined a threshold, which was used with the only purpose of coloring the tables. Thus, yellow cells denote deviations we have considered not relevant because the variation was smaller than this threshold (yellow cells with letter L), or because the changes were in parts of the source code we are not interested in such as the test packages (yellow cells with letter T). On the other hand, blue, red and green cells denote deviations greater than the predefined threshold. Red and green cells represent, respectively, significant degradation and optimization, it means scenarios with deviation greater than the threshold. In this case, the framework could identify source code changes inside the methods flagged by the framework as responsible for such deviation. The blue cells also represent deviations greater than the threshold, but they could not be associated to source code changes. It might be the result of external factors, for example, different libraries or settings, an isolated measurement effect as will be discussed in Subsection 6.3.1, or even changes outside the methods, such as in class attributes, but which would cause indirect impact.

The threshold can be configured in the framework. Thus, in order to color these tables, we applied thresholds of 15ms for scenario from the Jetty framework and 100ms for scenarios from Netty and Wicket frameworks. The Jetty threshold was chosen to be smaller only because Jetty has shorter tests compared to Netty and Wicket. The threshold values were applied with the only purpose of coloring the tables and separating very small variations, since they are probably irrelevant for developers. Thus, it is important to emphasize it does not represent a framework limitation related to short or long scenarios. As we can see, in the scenarios with significant deviation, the framework was able to find corresponding source code changes in most cases (green and red cells), and only six cases did not have corresponding code changes (blue cells). Thus, it is possible to conclude that practically all deviations above the given threshold can be correlated to source code changes. It means there is at least one commit that introduced modifications to degraded or optimized methods of such scenarios, what is a strong indication that these deviations actually reflect changes in the system and are not random fluctuations in the measurements.

It is interesting to note that most scenarios exhibited significant degradations (or optimizations) only for a specific release. There are only a few scenarios that had performance degradation for more than one release. For example, Netty had 13 performance-degraded only

in NE3, and Wicket had 6 degradations, and 12 optimizations only in WE4 and WE6, respectively. Jetty had 9 degradations in JE5.

6.2.2. Understanding the Performance Deviation Sources

Next, some of the code changes corresponding to the performance deviations of the indicated scenarios are going to be detailed. Table 9, Table 10, and Table 11 show an example of the framework output with the methods that were changed and had performance deviation in Netty, Wicket, and Jetty, respectively. They are the methods potentially responsible for contributing to the performance deviation of scenarios. The tables include the method name (following the pattern *class.method*), the number of scenarios it has affected, the number of commits which introduced changes in method over the evolution, and the performance impact. The results are ordered by the performance impact of each method, which is calculated as the arithmetic mean of the impact of the method in each scenario that it affects. The impact of a method is the total time it takes running in a particular scenario. This is just one strategy to show the results, and it does not necessarily imply that methods in the top of the table have caused more performance issues.

Table 9. Sources of optimization (N2, N5, N6) and degradation (NE1, NE3) for Netty.

Evolution	Method	Number of Scenarios	Number of Commits	Issues	Performance Impact
NE1	DefaultChannelHandlerContext.validatePromise	1	3	1606	18ms
	ChannelOutboundBuffer.newInstance	2	1	-	1ms
NE2	DefaultChannelHandlerContext.write	1	3	1707, 1832	161ms
	ChannelOutboundBuffer.incrementPendingOutboundBytes	1	3	1697, 1832	2ms
NE3	DefaultChannelHandlerContext.safeExecute	5	2	1947	80ms
	NioEventLoop.openSelector	12	1	1908	16ms
	ThreadPerChannelEventLoopGroup.register	10	1	2060	3ms
	SingleThreadEventLoop.register	10	1	2060	3ms
	ChannelOption.valueOf	12	1	-	2ms
	PooledByteBuf.deallocate	1	1	-	2ms
	AbstractByteBufAllocator.toString	1	1	-	2ms
NE5	PoolArena.free	1	1	2264, 808	337ms
	PoolThreadCache$MemoryRegionCache	1	1	2264, 808	334ms
	PoolThreadCache.createSubPageCaches	1	1	2264, 808	298ms
	PoolThreadCache.createNormalCaches	1	1	2264, 808	36ms
	NioSocketChannel.newSocket	2	1	2311	2ms
	NioServerSocketChannel.newSocket	2	1	2311	2ms
	PoolThreadCache.add	1	1	2264, 808	2ms
	NioDatagramChannel.newSocket	1	1	2311	1ms
NE6	DefaultChannelPipeline.addLast0	5	1	-	7ms
	DefaultChannelPipeline$HeadContext	1	1	-	5ms

Table 10. Sources of degradation (WE4) and optimization (WE6) for Wicket.

Evolution	Methods	Number of Scenarios	Number of Commits	Issues	Performance Impact
WE4	MarkupContainer.addedComponent	7	5	5410, 3335	605ms
	MarkupContainer.add	7	1	3335	603ms

	MarkupContainer.dequeue	7	10	3355	548ms
	Page.onBeforeRender	7	1	5426	406ms
	WebPageRenderer.isPageStateless	7	1	5426	400ms
	WebPageRenderer.shouldRenderPageAndWriteResponse	7	5	5426, 5484, 5522	380ms
	WebPageRenderer.respond	7	6	3347, 5309, 5426	370ms
	ListView.onPopulate	7	1	-	357ms
	MarkupContainer.newDequeueContext	7	1	-	331ms
	AbstractRepeater.dequeue	7	5	3335	215ms
	DefaultPageFactory.newPage	6	1	5215	159ms
	Page.renderPage	7	1	5426	144ms
	WebPageRenderer.renderPage	7	1	-	130ms
	MarkupContainer.onInitialize	7	1	-	118ms
	Component.internalRenderHead	7	1	4964	108ms
	Initializer.register	1	1	-	106ms
	MarkupContainer.dequeueAutoComponents	7	3	3335	105ms
	Initializer.createProxy	1	1	-	102ms
	Initializer.init	1	2	-	95ms
	MarkupCache.loadMarkupAndWatchForChanges	7	2	5294	54ms
	MarkupCache.loadMarkup	7	2	5294	53ms
	MarkupContainer.canDequeueTag	1	6	3335	35ms
	Component.setMetaData	1	1	5459	27ms
	JavaSerializer.deserialize	3	1	-	22ms
	WebPageRenderer.shouldRedirectToTargetUrl	1	4	5426	20ms
	XmlPullParser.parse	6	1	5398	16ms
WE6	WebPageRenderer.respond	12	2	5689	614ms
	MarkupContainer.dequeueAutoComponents	12	1	5730	356ms
	Application.initializeComponents	5	1	5713	342ms
	MarkupContainer.newDequeueContext	5	1	5730	47ms

Table 11. Sources of performance deviation for Jetty. Only degradation (JE5).

Evolution	Methods	Number of Scenarios	Number of Commits	Issues	Performance Impact
JE5	MimeTypes$Type	6	2	439375	5ms
	PathResource.checkAliasPath	1	4	-	4ms
	ServerConnector.newSelectorManager	1	1	-	1ms

The results have indicated that when an optimization happens after a past degradation it is not necessarily caused by changes in the same source code assets that were responsible for the past degradation. That happens on some Netty evolutions (N3 and N6). Degraded scenarios in N3 were optimized in N6, as we can see in Table 6. However, observing the Table 9, we can noticed that the modified methods indicated by our approach as responsible for the deviations are different. It shows that optimizations are not necessarily caused by an attempt of the developers to fix past degradations.

The scenario names and commit codes are not presented in order to not decrease the readability of the tables. The complete list of commit codes and issues is available in Appendix B and online [55]. On the next paraphrases, an overview of some of the commits and development issues, which were found by the approach application, is presented. In order to do that, we have conducted an inspection of the repositories of the target systems to get a better

understanding of the changes. It includes the version control system, issue tracking system, and release notes.

However, we will not detail or explain every change the framework has found because it is not our goal, and it is out of the scope of the work. The full description of commits and development issues can be found in the official websites of the target systems (Netty [28], Wicket [30] and Jetty [31]). In each website is available the links to access the version control system, the issue tracking system, and the release notes for each target release.

Netty: From release 4.0.0.Final to 4.0.6.Final (NE1). The framework found four commits and an improvement issue (#1606). Most of the changes affected the `validatePromise()` method by adding some extra code validation. A new way to instantiate the `ChannelOutboundBuffer` class in order to make its objects recycled was introduced in the `newInstance()` method. The commit added 7 and deleted 45 lines of code, respectively, and there is a performance improvement when objects from `ChannelOutboundBuffer` are recycled, and Netty avoids creating them again, but for new objects the performance decreased.

Netty: From release 4.0.6.Final to 4.0.10.Final (NE2). All versions in between these releases were bug fixing with some improvements. The framework found five commits and three development issues, including one improvement (#1707) and two unlabeled issues (#1697 and #1832). An important unlabeled issue (#1697), also highlighted in the releases notes of the release 4.0.7.Final, fixed a bug related to buffer management. The solution introduced a new way to estimate the size of messages that should be written in buffers. Another interesting change introduced by another commit was intended to fix a callback problem when writing to a channel in the release 4.0.8.Final. This commit was not linked to a development issue.

Netty: From release 4.0.10.Final to 4.0.15.Final (NE3). For the third evolution, the framework detected that most of the scenarios were degraded (13 out of 20 scenarios). Seven commits and three development issues were found related to seven methods. The issues were two bug fixing (#1908 and #2060) and one unlabeled (#1947). The unlabeled issue (#1947) changed the `DefaultChannelHandlerContext` class in order to deal with a problem related to reject execution exceptions. It also added a new method named `safeExecute()` to the class, which affected part of the degraded scenarios during this evolution. One of the bug fixing issues (#1908) introduced changes to the method `NioEventLoop.openSelector()` to validate if internal objects are assignable. Another interesting change was a commit intended to improve

the buffer leak report, which introduced wrappers. Now, a leak-aware buffer can detect and report memory leaks. It results in 1800 added lines and 17 changed files. Table 6 (NE3) shows the list of performance-degraded scenarios in this evolution.

Netty: From release 4.0.17.Final to 4.0.18.Final (NE5). As shown in Table 6 (NE5), this evolution has optimized some scenarios. Two commits related to three development issues, including two improvements (#808 and #2264) and one new feature (#2311), were found. One of the responsible modifications for the optimization has introduced changes to the `PoolThreadCache` and `PoolArena` classes. According to this commit description, the changes "remove the synchronization bottleneck in `PoolArena` and so speed up things". The problem was solved by improving the synchronization and implementing a thread-local cache for pooled buffers.

Netty: From release 4.0.18.Final to 4.0.21.Final (NE6). The framework detected one commit responsible for the optimized scenarios (see Table 6) which changed the `DefaultChannelPipeline` class to improve memory usage and initialization time. It refactored some source code and modified the strategy to generate the names of a communication channel.

Wicket: From release 6.18.0 to 7.0.0-M1 (WE4). This was the only evolution with significant performance degradation for Wicket (see Table 7). The framework found 38 commits and 12 issues (one new feature, four improvements, and seven bugs). One of the new features (#3335) has implemented a queueing strategy for adding and extracting hierarchy information from markup. It added a substantial amount of new code to 14 files, including `MarkupContainer`, `Page`, and `AbstractRepeater` classes. Methods such as `add()`, `addedComponent()` and `queue()` from these classes have introduced new validations. Obviously, new features and new code might cause execution time increases, but it is a team decision to say if these increases are suitable or not. The proposed approach automatically detects the deviations and corresponding changes, so that developers can be aware of the specific consequences of their work. Another change caused by a bug fixing issue (#5426) corrected problems related to component states when they are rendering. One of the main classes changed was `WebPageRenderer`. An improvement issue (#3347) tried to simplify the way in which the `WebPageRenderer.respond()` method decides whether it will redirect or directly render the current page depending on several complex conditions. This issue resulted in 830 added lines and 95 deleted lines.

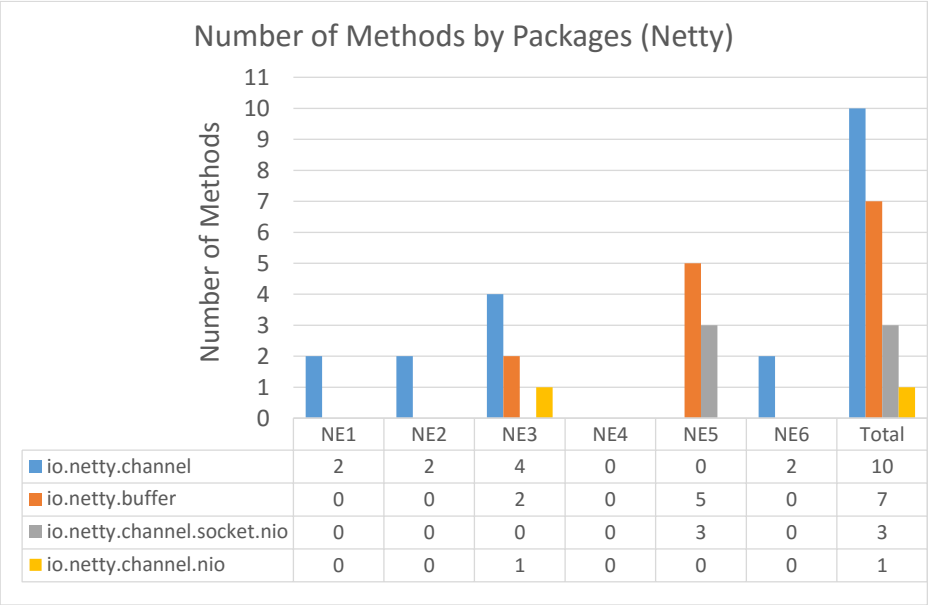
Wicket: From release 7.0.0-M2 to 7.0.0-M4 (WE6). For this evolution, the framework detected only performance optimization in terms of execution time. We believe that the changes from version 6.x to 7.x (WE4) introduced many problems and unsolved situations due to unstable code that was responsible for the degradation in the previous evolution (WE4), which were then addressed in this evolution. A bug fixing issue (#5689) changed a lot of source code in order to solve conflict problems in the `WebPageRenderer` class. An extra commit also changed this class in order to refactor and improve the `respond()` method, replacing a big part of the code introduced by the issue #3347 in WE4. Another bug fixing issue (#5730) simplified and corrected the de-queueing component process.

Jetty: From release 4.2.10 to 4.3.0.M0 (JE5). This was the only evolution in which the framework found significant performance deviation for Jetty. Changes were introduced in three methods and related to seven commits and one bug issue (#439375). The main changes were introduced by a commit that aimed to pre-encode HTTP fields. It modified 25 files with 449 line additions and 147 line deletions which was enough to affect all six blue scenarios from Table 8.

As said before, providing a detailed description of each commit and development issue found is out of the scope of this work. However, we conducted a manual inspection of the results in order to better understand the changes and to ensure that they make sense.

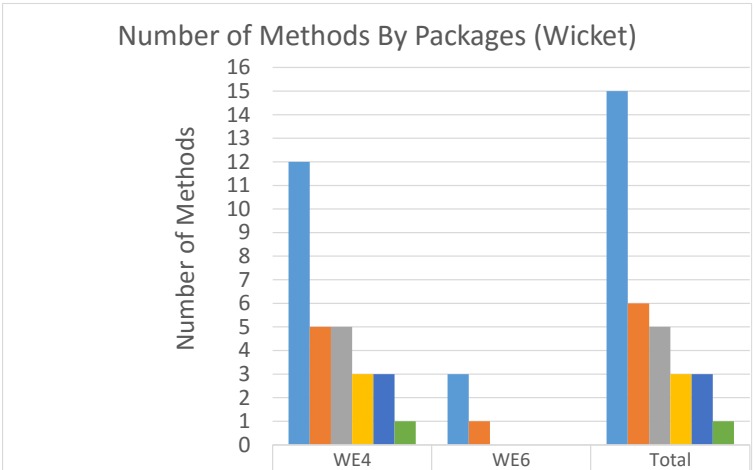
6.2.3. Results Characterization

In order to characterize these results and to provide developers with a better understanding of them, the framework has also determined how the total number of degraded and optimized methods from the scenarios detected with performance issues are spread over the system implementation for each analyzed evolution, including packages and classes. Thus, Figure 28 and Figure 29 shows how these methods are spread over the packages for Netty and Wicket, respectively, while the complete data including the information related to the classes is available in Appendix C and online [55]. Jetty had only three methods.



	NE1	NE2	NE3	NE4	NE5	NE6	Total
io.netty.channel	2	2	4	0	0	2	10
io.netty.buffer	0	0	2	0	5	0	7
io.netty.channel.socket.nio	0	0	0	0	3	0	3
io.netty.channel.nio	0	0	1	0	0	0	1

Figure 28. Number of Methods by Packages for Netty.



	WE4	WE6	Total
org.apache.wicket	12	3	15
org.apache.wicket.request	5	1	6
org.apache.wicket.markup	5	0	5
org.apache.wicket.jmx	3	0	3
org.apache.wicket.session	3	0	3
org.apache.wicket.serialize	1	0	1

Figure 29. Number of Methods by Packages for Wicket.

The framework has identified only four packages that contain all 21 methods responsible for performance deviations in Netty. According to Figure 28, the channel package concentrates the majority of the methods potentially responsible for the variations with a total of ten methods distributed over ten classes. Thus, we can conclude that the channel package may represent one of the sensitive points for performance in Netty. Regarding the distribution of the methods inside classes, we detected that the numbers are balanced with one, two, or three methods.

For Wicket, the `MarkupContainer` and `WebPageRenderer` classes contain the majority of the methods responsible for performance deviations (ten and six, respectively). The classes inside the `org.apache.wicket` package also concentrated most of these methods, 15 in total (see Figure 29). These numbers are indicators that developers should pay particular attention when evolving these code assets because changing them might lead to performance deviation that could increase or decrease the execution time of methods and scenarios.

For Jetty, the framework found only three methods that belong to the `MimeTypes$Type`, `ServerConnector` and `PathResource` classes from the `org.eclipse.jetty.http`, `org.eclipse.jetty.server` and `org.eclipse.jetty.util.resource` packages, respectively.

Based on the results, we conclude that the channel package from Netty, and the `MarkupContainer` and `WebPageRenderer` classes as well as the `org.apache.wicket` package from Wicket are the parts of the systems that concentrate most of the sources of performance variation and represent elements that developers should pay attention to when evolving the systems. On the other hand, there was no package or class that concentrated most of the deviation-related changes for Jetty.

To complete the characterization of the study results, the following kinds of issues, which were linked to commits that introduced changes to methods with performance deviation, were also identified: **(i)** improvement (9 issues), **(ii)** bug/defect (12 issues), **(iii)** new feature (2 issue), and **(iv)** unlabeled (3 issues). Defining the issue type is not mandatory in most of the issue tracking systems, and not all commits are associated with an issue. We found that 32 commits out of 66 – 10 for Netty, 16 for Wicket and 6 for Jetty – are not linked to any issue. This represents almost 50% of the commits we found, which makes it difficult to draw conclusions about the types of issues that are most likely to cause performance deviation. Table 12 summarizes the types of issue for each system.

Table 12. Types of issues for each analyzed system.

Type	Netty	Wicket	Jetty	Total
Unlabeled	3	0	0	3
Improvement	4	5	0	9
Bug/defect	2	9	1	12
New Feature	1	1	0	2
Total:	10	15	1	26

6.2.4. Online Survey Feedback

RQ2. Are the developers aware of the performance issues that the approach was able to find? Performance deviation is difficult to notice, it may occur in a progressive way and some problems might be realized only after multiple evolutions. We collected feedback from eight developers using a web-based survey to investigate if they were aware of the deviations we found and to give us a preliminary perception of what they think about the proposed approach, support framework, and results.

The surveys were prepared for developers of Netty, Wicket and Jetty. They asked developers about tools and strategies they usually use to manage performance and if they were aware of some of the performance deviations that the approach found. The total number of developers, extracted from the contributor pages of each system on GitHub, is 100, 30 and 20, for Netty, Wicket and Jetty, respectively. We contacted the developers that had published their email address, and received responses from four, three and one developer for those systems, respectively.

For the scenarios identified with performance degradation, we gave three examples with increase of time that we judged representative samples and important functionalities to the participants of each system and asked questions in order to investigate whether they already were aware of the variations. The examples were (extracted from Table 6, Table 7, and Table 8):

- Netty:
  - `WriteBeforeRegisteredTest`
  - `SocketEchoTest`
  - `SocketObjectEchoTest`
- Wicket:
  - `HangManTest`
  - `NiceUrlTest`
  - `LinkomaticTest`
- Jetty:
  - `AsyncServletLongPollTest`
  - `AsyncIOServletTest`
  - `DispatcherForwardTest`

On the last page of the survey, we presented an overview of the approach and we summarized the main study results with a table, similar to Table 9, Table 10, and Table 11 according to the specific system of each developer. Presenting such information aimed to introduce the approach and ask what participants think about it, including different strategies they would use, and to ask for additional comments. On the next paragraphs, we are going to discuss some feedback from these developers. The questions of such surveys and the answers of these eight developers are available in Appendix D and online [55].

In response to the question *"In your opinion, how important is execution time (response time) for the system?"*, seven participants confirmed that performance in terms of execution time is important in the context of these systems, while only one was neutral about it. Some participants (3 out of 8) were aware that some releases had execution time issues when we asked them *"Are you aware of any execution time variation (increase or decrease) in any of these releases: <list of target releases we selected for each system>?"*.

However, when we asked *"We noticed that the execution time increased for several test cases between releases 6.18.0 and 7.0.0M1. After that, the execution time decreased again between releases 7.0.0M2 and 7.0.0M4. This was the case for scenarios tested by classes such as HangManTest, NiceUrlTest, and LinkomaticTest. Are you aware of these execution time variation?"* and *"Considering the examples of the previous question (HangManTest, NiceUrlTest, and LinkomaticTest), what methods, commits, or development issues caused the increase in execution time?"*, none of our participants were aware of the specific execution time variations that the approach had identified, and were unable of indicating sources. We recognize these questions might be very difficult to answer, but it confirms that without a suitable performance analysis tool it is very difficult to indicate causes, even if developers are aware of the performance deviations. These questions were adapted for each system with the appropriate target releases and the three selected examples indicated previously.

In addition, the majority of the participants (5 out of 8) indicated the usefulness of the approach, while two were neutral, and only one thought it was not very useful. Three participants also mentioned the profiling tool YourKit when we asked about tools they usually use for performance testing. However, this tool does not automatically compare execution time between releases, which is why one of the participants said about our work: *"Nice tool if it is really automatic since currently comparison is a manual process"*. In addition, one of the participants was going to check specific methods based on the data we provided in the survey:

“It is interesting enough that I will be looking at MimeType$Type and checkAliasPath changes to double check we've not done anything stupid”.

Some participants were concerned about micro-benchmarking, mainly because of the use of test cases for performance testing, since test cases can be very small and may not have practical impact in performance. We recognize that this may be a threat in the study especially for Jetty that has the smallest test cases. However, the use of test cases as scenarios was simply a choice we made for the evaluation study. Because of the usage of annotations, the approach is flexible and can instrument any part of the system source code. Thus, it is a limitation of the study rather than of the tool or approach. Nevertheless, it is interesting to notice that even when using test cases, the framework had found several performance degradations in scenarios.

In this context, it is also important to analyze the significance of the deviations. For very small deviations, developers may not notice the variation and will not fix it in a future release because it actually does not matter. A workaround is to configure the framework to only report results with a specific minimum impact. This can be currently done by defining different values for predefined thresholds.

We conclude that, in general, the approach can find performance deviations that developers are not aware of. They could realize such deviation by running a profiling tool, but these tools do not automate the comparison of different releases and they do not provide details about methods, commits, and issues responsible for deviations.

6.3. Conclusion

The outcomes returned by the approach application shows the feasibility of the proposed approach to evaluate performance of scenarios in terms of execution time. The goal of this study was to assess the capacity of the approach to identify performance issues over multiple evolutions of existing systems and to check if the developers of these systems were aware of the issues. Thus, the approach identified degraded and optimized scenarios over the evolutions of Netty, Wicket and Jetty, and determined the potential causes of such variation by indicating code assets, development issues, and commits (RQ1). The feedback obtained through surveys indicated that developers are unaware of the performance variations we found (RQ2). In addition, we were able to characterize the modified methods responsible for the performance deviation by highlighting packages and classes where such methods were declared and associated issues. We also noticed that optimizations are not necessarily achieved by fixing past

degradations, but also improving source code assets which had not been degraded before, as we observed on evolution N3 and N6 from Netty.

Through the study results, we expect to help developers detect performance issues before releasing software systems, and easing the process of fixing these issues by identifying their causes. In this context, the framework can be used as a preventive tool. This study shares some common limitations and threats to validity with the study presented on the previous chapter. Thus, in addition to the threats to validity presented in Chapter 5, next subsections of this chapter discuss some extra threats.

6.3.1. Measurement Threat

Our approach relies on multiple executions of scenarios to increase the confidence in the measurements. Even though we executed the test suite only ten times for each release (and 30 for Jetty) for performance deviation analysis, a particular method might be executed much more often. For example, considering the repetitions, the Wicket method `respond()` (Table 10) was executed 30 times inside the *Entry point for NiceUrlTest* because it is called three times as part of the corresponding test case. On the other hand, the method `addedComponent()` was executed 1020 times inside the same entry point. In these cases, we are able to obtain samples that are more representative. This and other precautions, such as disabling every non-essential service of the environment, running each test in its own JVM, using a random order for each repetition and conducting a manual inspection of the results help us to decrease the risks of measurement bias [79] due to the high sensitivity of measuring execution time.

6.3.2. False Positive and False Negative

The execution time is calculated while the system is running. It is not an estimation, but the real time the scenarios take to execute. Considering that the measurement risks were minimized and the values for the execution time of scenarios and methods are right, the framework will always be able to correctly identify the scenarios with performance deviation by using the arithmetic mean or statistic test strategies (Subsections 3.2.3 and 4.1.3). In this case, the framework will mine the methods with performance deviation for such scenarios and retrieve the associated commits. As every deviation was identified, the framework will be able to get the commits. It means the approach does not suffer with false negatives since every commit related to deviations of interest will be taken. Obviously, it is true when the source of the deviation is a code element. As the approach addresses source code changes, deviations caused by different settings or infrastructure changes, for example, will be detected in the

deviation analysis phase, but the framework will not be able to associate a commit to the problem in the repository mining phase.

However, we recognize that the approach, in the way it was applied in the studies, may suffer with false positives because different commits can affect the same source code assets over an evolution, or because a method impacted by other method has also changed over the same evolution. It could be solved through a per-commit application of the approach instead of a per-release application as we did in the studies. While such strategy may take a long time, it will achieve a more accurate result. Despite these issues, the main idea of the proposed approach is to provide a set of sources of performance deviation in order to help developers to improve and optimize software systems, and we are not claiming that every commit found by the framework in a per-release application is indeed blamed for performance deviation.

6.3.3. JUnit Tests and Micro-benchmarking

Some developers, which answered the online survey applied in this study with multiple releases of Netty, Wicket and Jetty, were concerned about micro-benchmarking, since test cases may not have practical impact for execution time because they were not written for performance-testing purposes. We recognize this threat, but the use of test cases as scenarios was simply a choice we made for the evaluation study. Developers could use the provided scenario annotations instead of the test annotation since the approach can instrument any part of the code. Thus, any strategy to exercise the scenarios will work. Despite the usage of such automated system tests in the studies, they still allowed us to find scenarios with performance deviation, particularly performance degradation, for the investigated systems.

6.3.4. Impact of Instrumentation and Rename Problem

As explained on the previous study, the threats related to impact of instrumentation and rename problem also apply for this one. The ways to handle them were the same. For the first one, we believe it does not form a problem since our analysis compares pairs of values of execution time from two releases, and the instrumentation should cause the same or very similar increases in both values. For the second one, our manual inspection to better understand the found results showed that, for this study, names were consistent in general.

6.3.5. Generalization and Limitation of Results

Some results of our study cannot be generalized. For example, the type of issue with more occurrences found during the evolutionary analysis was bug/defect (Table 12). However,

we cannot say that issues with such type are more likely to contribute to performance deviation than others. We also recognize the low number of developers that participated in our surveys (Subsection 6.2.4). However, while such results cannot be generalized, we believe they offer strong evidence that our approach and support framework are able to find performance deviations, commits and development issues that developers are not aware of. In addition, we could not check if every commit flagged was correctly selected by the framework because to the best of our knowledge none of these systems have any kind of structured repository to keep or flag performance regression changes.

7. PERFORMANCE AND CODE CHANGE PROPERTIES

The degradation of quality attributes happens when a new release of a software system exhibits inferior measurements of the quality attributes compared to previous releases. The ability to understand and analyze the impact of newly introduced changes are an essential prerequisite to avoid issues related to software erosion, what means the overall deterioration of the engineering quality of a software system during its evolution.

In this context, this chapter focuses on the relationship between the degradation of the performance quality attribute and code change properties. It considers the performance in terms of execution time in the same way of the proposed approach. It refers to the responsiveness of the system. Execution time can reflect issues caused by many other properties, such as high CPU, memory or disk usage, which will very often decrease the system responsiveness, and it can be noticed in terms of time.

The rest of this chapter is organized as follows: **Section 7.1** motivates this part of the work by suggesting the importance of going deeper in the results, and understand how code changes can affect the performance in terms of execution time; **Section 7.2** describes the study, including goals and research questions; **Section 7.3** presents an overview of the code change properties mined by the support framework during the evaluation and repository mining phase of the approach; **Section 7.4** presents the methodology to build a performance regression model; **Section 7.5** reports the results; **Section 7.6** discusses the threats to validity; and, finally, **Section 7.7** concludes this study.

7.1. Motivation

Basically, we define code change properties as metrics calculated from commits and source code assets. The goal is to improve the understanding of the performance degradation phenomenon by studying specific characteristics of commits that can affect the likelihood of introducing this problem. Be aware of performance issues and their causes are particularly important when degradations have been introduced. Several research works have used a similar approach to explore the likelihood of developers introducing general bugs. Most of these studies consider popular commit properties: **(i)** the commits' time and day [64] [65], **(ii)** the experience of the developer responsible for the commit [64] [66] [67], and **(iii)** some complexity metrics of the commits (e.g., added and removed elements in the source code) [68].

Despite being an important software quality problem, there is no significant research work on performance bugs. Research has focused especially on improving the overall software quality by doing qualitative and quantitative studies on software bugs in general [69]. For example, some software defect prediction models use metrics derived from software bugs to predict the number of defects and the locations where to fix them [70] [71] [72]. Other work focuses on different types of bugs like security bugs [73] [74], or usability bugs [75] [76].

In addition, despite some research work related to performance regression repositories, there is an absence of approaches capable of automatically identifying commits potentially responsible for performance regression, and build logistic regression models from properties of these commits in order to understand which properties are tight related to performance degradation. For example, Nguyen et al. [51] propose the mining of regression-causes repositories to assist the performance team in identifying the cause of a newly identified regression. They use machine-learning techniques, but their focus is on classifying types of changes that cause performance regression and not in discovering the changes (commits) responsible for that by mining the software repositories. Huang et al. [78] propose a performance risk analysis to prioritize and recommend which commits should be first tested for performance regressions. Zaman et al. [69] have qualitatively studied performance bug reports, comments and attached patches, in order to understand the differences in the actual process of fixing performance and non-performance bugs, and potentially identify concrete ways to improve performance bug resolution.

In this context, our study proposes the definition of a regression model that aims to explain the phenomenon of performance degradation based on code change (commit) properties. It relies on the presented scenario-based approach for automating the performance analysis of multiple system releases considering the property of execution time. The results of the approach application for 23 releases of four systems, including ArgoUML, Netty, Wicket and Jetty were used to build the model.

7.2. Goals and Research Questions

In the previous studies (Chapters 5 and 6), the framework was collecting code change properties related to the commits involved with the selected scenarios when executing the last phase of the approach. The goal of this study is to reuse such data to build a regression model to explain the phenomenon of performance degradation over evolutions. It is an exploratory study since the majority of the approaches require the analysis of regression test repositories,

as said before. Thus, there is no equivalent approach that use data extracted from automatic performance analysis to build performance regression models.

The following research questions are explored in our study:

RQ1. *What are the most influential commit attributes to determine performance degradation?* For this research question, we define explanatory variables based on some of the properties from the previous subsection. They were analyzed in order to determine if there is an explanatory variable, or even a set of variables, that may increase the probability of a commit to cause performance degradation.

RQ2. *Is there any relationship between the commit size and performance degradation?* It is a point we would like to verify, mainly because the previous research question detected that commits integrated earlier are more likely to cause performance degradation. Thus, we also want to check if commits integrated earlier are more complex than commits integrated later (near the release date).

RQ3. *Can the regression model accurately predict when commits cause performance degradation?* Here, we are interested in evaluate how accurate the built regression model is to predict when a commit may cause performance deviation based on its explanatory variables. It could be used to create development support tools to assist the developers.

With this study, we expect to improve the understanding of the performance degradation phenomenon by studying specific characteristics of commits that can affect the likelihood of introducing this problem.

7.3. Code Change Properties

The framework can be enabled to calculate code change properties related to commits during the last phase of the proposed approach for performance deviation analysis. In this context, most of our commit properties were inspired from existing work, including the time and day [67] [68], the experience of the developer [67] [69] [70], and metrics related to complexity (e.g., added and removed elements in the source code) [68]. Thus, a set of properties was collected when the framework was applied to the target systems, including ArgoUML (from Chapter 5), Netty, Wicket, and Jetty (from Chapter 6). The collected properties are stored in the change analysis model (see Figure 21), which are:

- **Author activity.** The number of previous commits an author has already done counting from the current analyzed commit. It is a way to measure the author activity, since the expected is that authors with more revisions are also more active and may have more expertise in the project.
- **Number of days before release.** The number of days from the current commit date to the release date. The idea of including such properties was to analyze if there is a relationship between the number of days before a release and the likelihood of introducing performance degradation. Maybe, authors of commits near the release date had not enough time to test their solutions, or even in an opposite thinking, commits far from the release date might be affected or broken by more recent commits.
- **Is the commit a bug-fixing?** The information if the commit is related to a bug fix or not. It is basically a *Boolean* that will be true if the commit is linked to a bug or defect development issue.
- **Number of files.** The number of files impacted by the commit. The same idea of the previous metric, but considering every changed file, including Java files and other types.
- **Number of inserted lines.** It is a commit complexity metric to indicate the number of added lines.
- **Number of deleted lines.** It is also a commit complexity metric to indicate the number of removed lines.
- **Number of churns.** The sum between the number of inserted and deleted lines.
- **Effective inserted lines.** The difference between the number of inserted and deleted lines. It can be negative in case the commit has more deleted than inserted lines.
- **Number of hunks.** According to Alali et al. [77] "a hunk basically represents the ranges of lines that contain the changes in the two versions of the same file. The number of such ranges for a file approximately depends on the contiguous changed and unchanged lines". In other words, "a hunk is a continuous group of lines that are changed along with contextual unchanged lines" [77]. It can be an indicator of the size and complexity of the commits because it takes into account how changes made by developers are spread over the impacted files of the system. For example, imagine two commits C1 and C2. The first one changed a file between lines 50 and 70 (30 lines of code), while the second one changed a file between the lines 10 and 20 (10 lines of code), 33 and 35 (two lines of codes), and 40 and 43 (three lines of code), resulting in a total of 15 lines of code. Considering only the number of lines, we may say that C1 has introduced more

complex modifications. However, considering the number of hunks, we could conclude that C2 is more complex, since it has three hunks and C1 has only one.

- **Hour of day.** The hour of the day the commit was registered. The supposition is that commits registered at night or near to lunchtime may be more likely to cause performance problems.
- **Day of week.** The day of the week the commit was registered. Maybe, commits registered at the weekend are more likely to cause performance problems.
- **Type of performance deviation.** It indicates if the commit is associated to a method with performance deviation. The values can be 1 for performance degradation, 2 for neutral, and 3 for performance optimization.

7.4. Methodology and Target Systems

We use data collected from previous studies for performance deviation analysis of ArgoUML, Netty, Wicket, and Jetty. We use the framework to mine all properties described in previous section for each commit related to the executed code assets. We discovered a total of 997 commits, of which 103 were retrieved from degraded code assets, 19 from optimized code assets, while 875 had no impact on performance in terms of execution time.

In this study, we fit logistic regression models to study the relationship that our explanatory variables (properties shown in Section 7.3) share with the performance degradation (our response variable Y). We model our response variable as Y = 1 when a performance degradation occurred, and Y = 0 otherwise. In order to build our model, Harrell Jr. approach [80] was followed. Basically, we performed the following steps:

- We estimate the degrees of freedom that can be spent in our model. The degrees of freedom are spent every time that we use an additional explanatory variable to fit our model. According to Harrell Jr. [80], no more than *M/10* exploratory variables should be examined to fit a regression model, where *M* is the number of events in the less frequent category for binary outcomes. In our study, we have 10 degrees of freedom to spend in our explanatory model (103 degradations divided by 10).
- We remove any highly correlated explanatory variables, since highly correlated variables can distort the relationship between explanatory variables and the response variable. To do so, we use the Spearman rank correlation, which is resilient to data that is not normally distributed. In case that we find explanatory variables with a correlation

higher than 0.7 we remove one of them. For example, it was the case of number of *inserted lines* which has a high correlation with *churn*. Always *inserted lines* increase, *churn* also increases. Hence, we remove *inserted lines* from our models since *churn* has more information about code modification (both *inserted and deleted lines*). We perform this process iteratively until no highly correlated variables are present.

- We also remove redundant variables. Redundant variables are variables that can be explained by other variables that are being used. Hence, they do not add any explanatory power to the model. For example, if *churn* could be explained by *effective inserted lines*, there would be no additional gain in using both variables in our explanatory models. Instead, redundant variables may distort the relationship between the explanatory variables and the response variable. The redundant variables were *number of churns*, *number of hunks*, *number of files* and *number of deletions*. We use the *redum* function of the *rms* package [80].
- In order to check the stability of our models, we use the ROC area and the Brier score. The ROC area is used to evaluate the degree of discrimination achieved by the model. The ROC area is the area below the curve plotting the true positive rate against false positive rate. The value of ROC area ranges between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the explanatory model outperforms random guessing. The Brier score is used to evaluate the accuracy of probabilistic predictions. This score measures the mean squared difference between the probability degradation assigned by our model for a particular commit C and the actual outcome of C (i.e., if C is related to performance degradation or not). Hence, the lower the Brier score, the more accurate are the probabilities that are assigned by our explanatory models.
- In order to check if our explanatory models are not over-fitted, we subtract the *bootstrap-calculated optimism* from the initial ROC area and Brier score estimates [81]. This optimism is a robust measure to evaluate overestimation of the fit of explanatory models.
- To evaluate the impact of each explanatory variable used by our model, we use Wald X^2 maximum likelihood tests. The larger the X^2 value, the larger the impact that a particular factor has on our explanatory performance model. The X^2 values were calculated by using the *anova* function from the *rms* package to execute the tests. Finally, we study the relationship that the most impactful variables share with the response variable (performance degradation). To do so, we plot the change in the

estimated probability of degradation against the change in each impactful factor while holding the other factors constant at their median values using the *predict* function in the *rms* package [80].

7.5. Results

This subsection presents the main results of this exploratory study. First, it discusses about the most influential properties for performance degradation based on the framework results, and second about the accuracy of the performance regression model.

RQ1. What are the most influential commit attributes to determine performance degradation? After applying the Wald X^2 test on the predictive variables, two of them seem to be the most relevant: number of days before release and day of week. Table 13 shows the explanatory variables we chose for our performance regression model with the X^2 values, the degree of freedom, and the corresponding p-values. As we can see, the explanatory variables days before release and day of week have the highest X^2 values: 6.22 and 3.94, respectively. In addition, their p-values are both less than 0.05: 0.0126 and 0.0473, respectively. They represent the most relevant variables for the model. We also tried other combinations of explanatory variables in our explanatory models for the sake of completeness. However, those other combinations did not yield interesting results.

Table 13. Explanatory variables with X^2 values, degree of freedom and p-values.

Explanatory Variable	Chi-square (X^2 value)	Degree of Freedom	P-Value
Author Activity	0.17	1	0.6779
Days Before Release	*6.22*	*1*	*0.0126**
Bug-fixing	0.07	1	0.7917
Effective Inserted Lines	0.03	1	0.8619
Hour of day	0.93	1	0.3338
Day of week	*3.94*	*1*	*0.0473**
TOTAL:	11.36	6	2.7252

For the day of the week variable, Figure 30 shows that there is a tendency of increasing the probability of degradation (y-axis) when the day of the week also increases (x-axis). It means that the general tendency, represented by the blue line, of commits introducing performance degradation increase when the day of the week increases. It could be intuitively explained, since developers could be more careless or even being in a hurry to complete their tasks as soon as possible when the weekend is coming. The scale considers day one for Sunday, day two for Monday, and so on. In addition to this behavior, Sliwerski *et al.* [83] [84] analyzed

fix-inducing changes – changes that lead to bugs, indicated by fixes – and they have discovered for the analyzed systems that the likelihood that a change will induce a fix is highest on Friday.

The gray area in Figure 30 indicates how stable such tendency is. A thinner area indicates a more stable tendency for the analyzed period of days, while a larger area indicates this tendency is becoming more unstable. Thus, we can observe that during the week the general tendency is more stable from day two (Monday) to six (Friday) than near the weekend, mainly on day one (Sunday) and seven (Saturday). A possible explanation is the lower number of commits during the weekend, which are shown by Figure 31. Thus, we conclude that the tendency of causing performance degradation increases until Friday, but becomes unstable at the weekend due to the lower number of commits to analyze in this period.

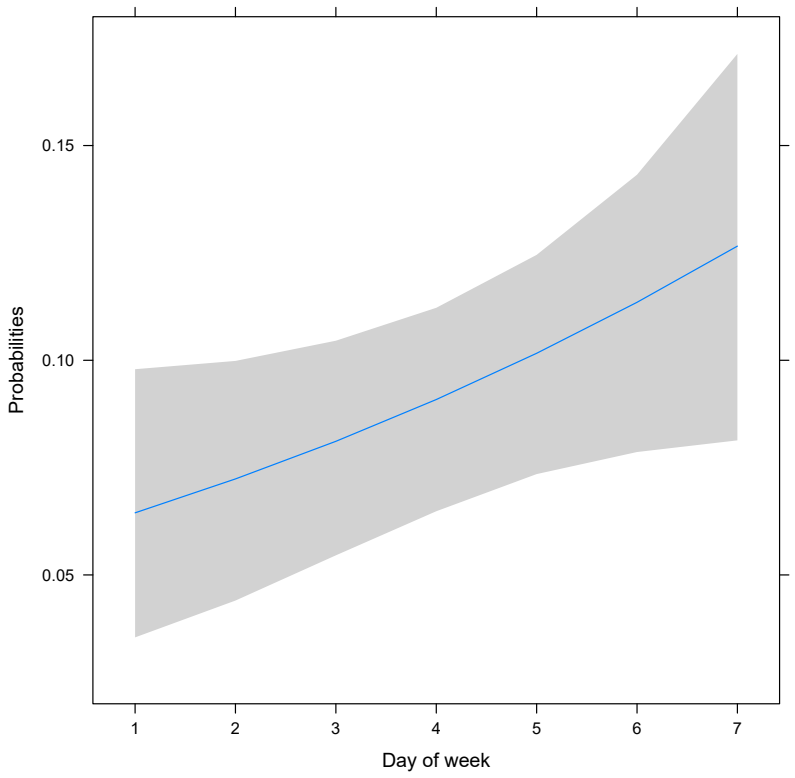


Figure 30. Tendency for day of week variable.

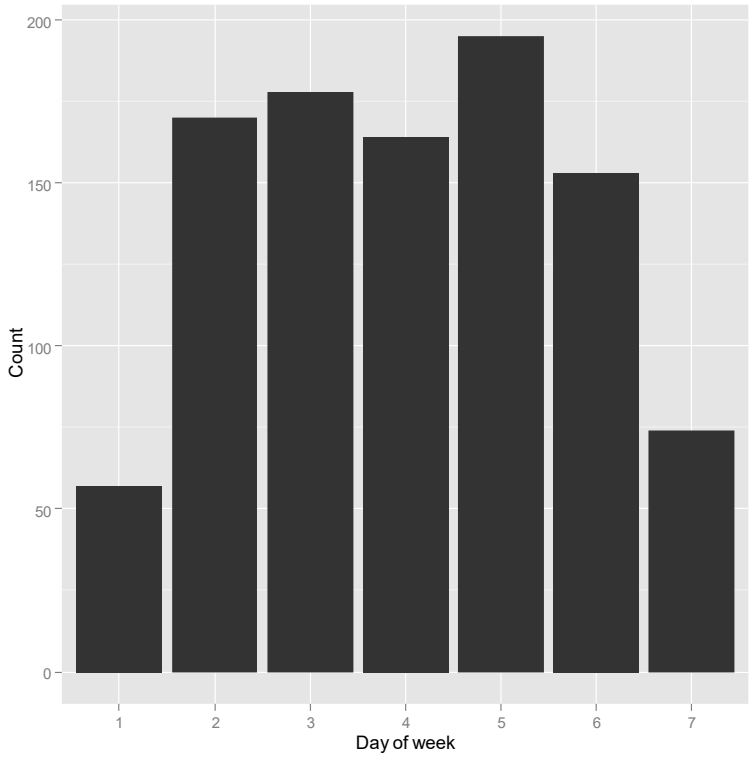


Figure 31. Frequency of commits over the week.

Regarding the number of days before release variable, the apparent tendency was the opposite of what we expected. The expected behavior was that authors of commits very near to the release date had not enough time to test their solutions, or even the deadline to implement the solution itself was too tight. However, the tendency was the opposite, the possibility of performance degradation increases when the number of days also increases. It means, commits that were performed earlier are more likely to cause performance degradation. According to Figure 32, the likelihood of performance degradation (y-axis) increases as the number of days (x-axis) increase. However, as the number of days increase, the general tendency becomes more unstable.

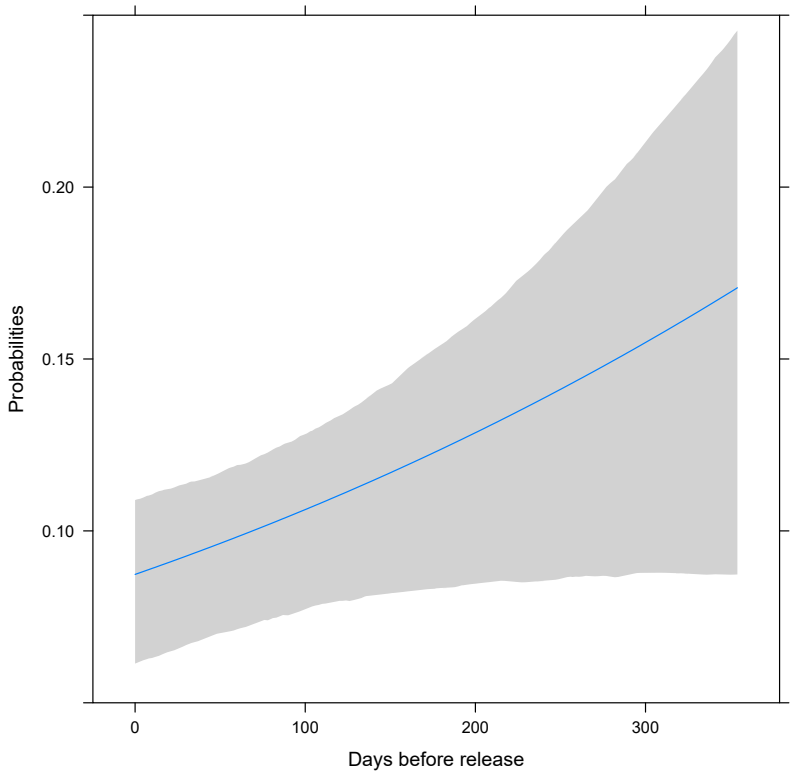


Figure 32. Tendency for number of days before release variable.

In spite of the initial unexpected tendency for the number of days before release variable, after thinking about that, we realized that it actually makes sense considering the way the dataset was built. Such behavior could be justified from the perspective that commits performed very soon have more chances to be impacted by posterior commits truly responsible for performance deviations, what may cause interference overlapping modifications.

For example, imagine the commits C1, C2, and C3, which have changed a method M. C1 is the newest commit and C3 is the oldest one. In addition, consider that only the changes of C3 have caused performance degradation for the method M and the scenarios it belongs to. In this case, the changes introduced by C3 are overlapped in the same code asset also modified by C1 and C2 during the analyzed evolution. Thus, when performing a per-release evaluation, the framework will indicate the commits C1, C2, and C3 because all of them have introduced changes to a degraded code asset, the method M during the analyzed period. Summarizing this perspective, the last commits always have a strong tendency to change code assets already modified, what may push early commits into the solution.

The opposite could also happen. Early commits truly responsible for performance deviations can push late commits into the solution when they have modified common code

assets. However, we believe it is not the most common situation. In this case, we are assuming that commits near of the release date are indeed more likely to cause performance deviation than the old commits which could be developed with more dedicated time. However, because of the way the dataset was built, by using a per-release analysis strategy, these new commits will always push old commits into the solution too. It increases the chances of commits integrated earlier being included, what may have interfered and masked the original behavior of the variable in the regression model. As we will explain in Subsection 7.6, the per-release analysis was a study limitation.

RQ2. Is there any relationship between the commit size and performance degradation? Apparently, there is no significant relationship between commit size and performance degradation. For example, only three of the top ten commits, considering the number of churns, are related to performance degradation. Four out of the top ten commits are merges and also the bigger ones. From these, only one commit is related to performance degradation.

The main motivation for such research question was the tendency of the variable number of days before release. We theorize that commits integrated earlier have more chance to cause degradation because they concentrate the majority of the modifications which may are implemented as soon as possible. Our inspection shows it was not the case, old commits are not more complex or bigger than the new ones. In order to check it out, cumulative graphs were plotted for some size and complexity properties, including number of churns and number of hunks, as shown in Figure 33 and Figure 34.

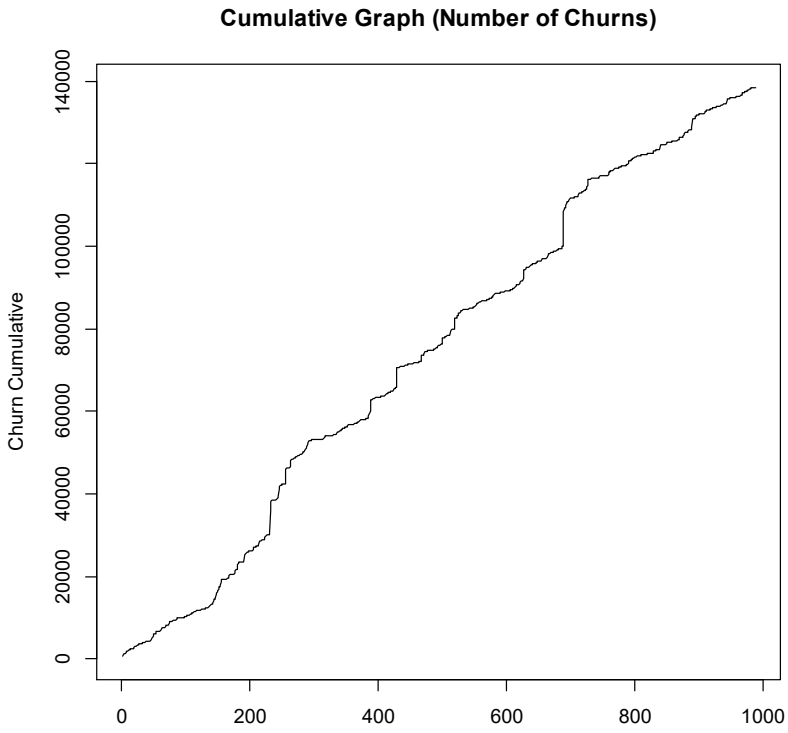


Figure 33. Cumulative graph for number of churns variable.

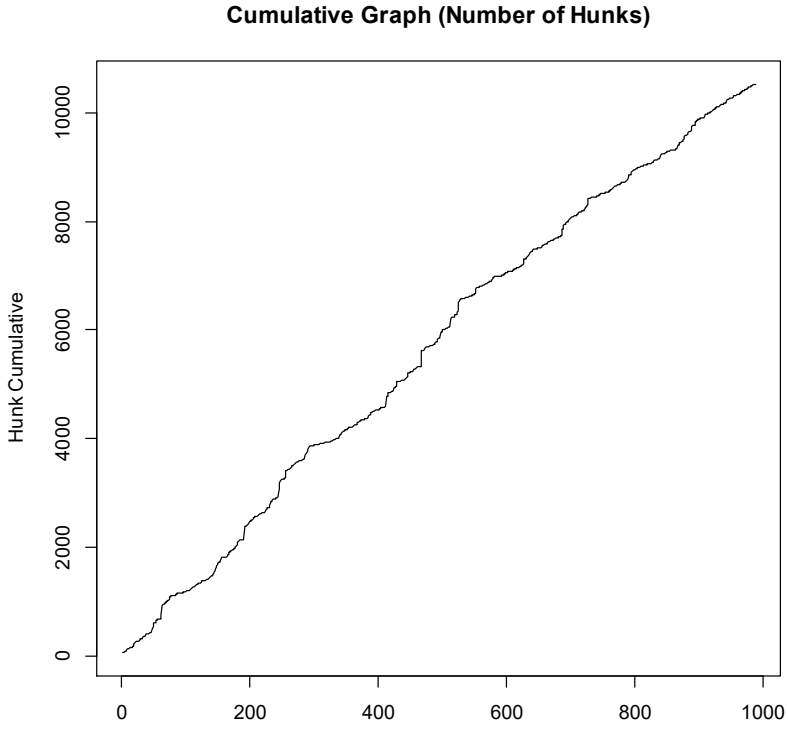


Figure 34. Cumulative graph for number of hunks variable.

The linear tendency of the cumulative graphs shows that the number of churns as well as the number of hunks grow gradually. It indicates that there is no apparent relationship, and old commits are not more complex or bigger than the new ones. This conclusion is supported

by the regression model, since the properties related to size and complexity of commits were not highlighted as the most influential variables.

RQ3. Can the regression model accurately predict when commits cause performance degradation? Our regression model has ROC area of 60%. It means that deciding if a commit will cause performance degradation or not by using the model is 10% better than a random guess, considering that a guess for true or false has ROC area of 50%. The Brier-score of our model was 0.09. A low Brier-score value indicates that the probabilities estimated by our model are fairly close to the real results. In addition, the *bootstrap-calculated optimism* from the initial ROC is 0.0839 and from the initial Brier-score is -0.0025, indicating that our model does not face over-fit issues.

We recognize that our explanatory model has room for improvement. Nevertheless, we consider that our current preliminary results are interesting, since, to the best of our knowledge, there is no approach doing similar analysis for data automatically extracted from commits related to performance degradation. Current approaches usually focus on mining repositories of regression tests, which do not exist for many software systems. Moreover, we aim to perform further studies using a larger dataset and a larger set of explanatory variables.

7.6. Threats to Validity

We highlight three main threats to validity to this study:

- **The dataset size.** Despite the amount of releases, we have analyzed, the data set is small for this kind of analysis. The framework has discovered a total of 997 commits, but only 103 commits came from degraded sources. The number is even smaller for optimization, only 19 commits. In this case, we cannot draw any conclusion about performance optimization, and for the same reason we could only draw preliminary conclusions related to performance degradation. Another implication is that we cannot generalize such conclusions for other systems or even other releases of the same systems.
- **The per-release strategy.** The framework was applied to analyze evolutions of software systems. When analyzing the period of time between two releases, many different commits may introduce changes to the same set of elements. Thus, if one of these commits has caused performance issues to a particular method, the other commits which also changes such method will also be included in the solution because it would not be possible to distinguish which one was truly responsible for the problem. It is not

a framework or approach limitation, but a study limitation that may have masked the real tendency of the number of days before release variable. The framework could be applied to analyze the changes after each individual commit in order to achieve a more accurate dataset. Obviously, it will take much more time to analyze the same period of time.

- **Dataset diversity.** The diversity of our dataset also makes more difficult the generalization of the obtained results. The dataset was built with commits from different systems which are maintained by different developers around the world. Some properties from these commits might suffer from local and cultural aspects. For example, the commit date, or even the release date, may be affected by local events and holidays from the places where their main developers live.

7.7. Conclusion

This chapter has just focused on the relationship between the degradations of the performance quality attribute and the related code change properties we have found. The main goal was to reuse the data from previous studies to build a regression model to explain the phenomenon of performance degradation over evolutions. In order to accomplish this task, the framework has been enabled to mine some metrics related to the commits associated with the executed source code elements. A subset of such metrics were used as explanatory variables in our regression model (see Table 13).

We mined a total of 997 commits, of which 103 were retrieved from degraded code assets, 19 from optimized, while 875 had no impact on execution time. Number of days before release and day of week were the most relevant variables of commits that cause performance degradation in our model (RQ1). Apparently, there is no relationship between the commit size and its probability to cause performance degradation (RQ2). The receiver operating characteristic (ROC) area of our regression model was 60% (RQ3), which means that deciding if a commit will cause performance degradation or not by using the model is 10% better than a random guess.

Finally, we recognize some threats to validity. First, despite the amount of releases we have analyzed, the dataset was still small for this kind of analysis. Second, a per-commit evaluation strategy may be more accurate than a per-release strategy, which we have adopted in our studies to analyze a period of time between two releases. Finally, the commits in the dataset may suffer from local and cultural issues of their developers.

8. RELATED WORK

Any research work related to evolution and performance could be relevant for this work. However, we noticed that there is no much research work focusing on the identification of the sources of performance deviation, considering software evolution, dynamic analysis and mining software repositories. Thus, the main novelty of this work is the possibility of automatically indicating the causes of detected performance deviations for scenarios in terms of methods and corresponding source code changes, what is achieved by the combination between dynamic analysis and repository mining techniques. Next paragraphs detail some related research work organized in groups according to the similarity of their approaches. The groups (sections) represent research topics where this work contributes directly or indirectly.

8.1. Architecture Evaluation

Several methods that explore the software architecture evaluation of quality attributes based on scenarios had been proposed [4] [8] [10] [11] [12]. These methods rely on manual reviews what implies in advanced skills from the evaluation team [4]. The proposed approach does not aim to compete with such methods because each one focuses on different stages of the development process. While the proposed approach performs late evaluation (when the system implementation is available), the methods for software architecture evaluation perform early evaluation (before the implementation began). However, our approach can complement these existing methods by providing automated analysis support for conducting the performance evaluation of multiples releases in all system's life-cycle, since it contributes to the continuous evaluation of the software system during its implementation and evolution stages.

8.2. Approaches for Architecture Documentation

Recent research work has proposed adding extra architectural information to the source code with the purpose of applying automated analysis or documenting the software architecture. Christensen & Hansen [56] use annotations to add information about components and design patterns, but their purpose is only to document the software architecture. Similarly, Mirakhorli et al. [57] present an approach for tracing architecturally significant concerns from the source code, specifically related to architectural tactics, which represent solutions for a wide range of quality concerns. Both research works do not explore the integrated usage of adding information related to scenarios or quality attributes.

Tibermacine & Zernadji [58] propose an approach to supervise the evolution of Web service orchestration, considering quality requirements. Their proposal is based on documentation analysis. They show how important design decisions that affect quality attributes can be formally documented and how it could be used to supervise architectural changes. Their strategy contains information about which quality attributes are in conflict, informing if a change that affect quality attributes in the software architecture can conflict with another. One of the main differences with our proposal is the analysis strategy, since we use dynamic analysis to collect more accurate information because we are able to instrument the source code during the system execution, and Tibermacine & Zernadji rely on documented rules.

8.3. Automated Approaches

Holmes & Notkin [37] propose to identify when developer's changes introduced to the program source code, tests, or environment affect the system behavior. In general, they consider that the system behavior has changed when program paths appear or disappear between executions of two subsequent versions of the system. Their approach does not aim to support an automated architecture evaluation based on scenarios or quality attributes, but parts of their techniques have been applied in a similar way to ours. Similar to our approach, their work builds the static call graph through source code analysis, and the dynamic graph with AspectJ during the tests execution. Then, they confront the graphs, and determine if changes in the system (static graph) affect the system behavior (dynamic graph), or vice-versa.

Similarly, Ciraci et al. [38] present a runtime approach for detecting inconsistencies between the dynamic behavior of the documented architecture and the actual runtime behavior of the system. They also use aspect-oriented development to instrument the source code in order to monitor the execution of scenarios that conflict with the documented architecture design. However, their verifications focus on reveal implementation inconsistences. There is no support to evaluate quality attributes such as the performance of scenarios.

Malik et al. [5] propose some different strategies to help performance analysts to more effectively compare results of load tests to find performance deviation in large-scale systems. Thus, they can compare the latest results with the previous one for similar tests and determine if there were any performance deviations. It adopts benchmarks to collect properties, such as CPU utilization, disk I/O, network traffic, related to performance during load tests. Their approach is specific for performance and works in a lower level of abstraction than our

approach. The identification of the problems introduced during the evolution is indicated in terms of measurements and their related elements. There is no mention of repository mining or attempts to identify the sources related to the performance problems.

8.4. Mining Software Repositories Approaches

Nistor et al. [19] investigate how performance bugs are discovered, reported to developers, and fixed by developers and compare these results with non-performance bugs. They defend that designing effective techniques to address performance bugs requires a deep understanding of how performance bugs are discovered, reported, and fixed. Thus, they explore if functional bugs are more likely to be introduced by performance bugs or by non-performance bugs, and if fixing a performance bug is more difficult than fixing a non-performance bug. In addition, they compare how performance bugs are discovered compared to non-performance bugs. In this case, the conclusions are that the majority of the non-performance bugs are discovered by users or developers due to the observation of unexpected behaviors (e.g. system crashes), while a large amount of performance bugs are discovered through reasoning about the code. According to the authors, the result suggests that techniques to help developers reason about performance, better test oracles, and better profiling techniques are needed for discovering performance bugs. It shows how important is to consider the source code when evaluating performance. Our approach integrates dynamic analysis and repository mining in order to instrument the system execution and discover changes related to source code assets, respectively. Nistor et al. do not address the causes of the performance bugs.

Nguyen et al. [20] propose mining a regression-causes repository to identify causes of new performance regressions. The repository contains the results of performance tests and causes of past regressions. They use machine-learning techniques to determine the causes of new regressions based on data from the repository. The causes are a pre-defined set of situations extracted from bug reports that represent actions that usually cause performance regression, such as adding frequently executed logic or adding blocking I/O access. Thus, the authors are able to categorize causes of new performance regression based on past data. Our approach does not categorize causes of regressions, although it indicates a set of commits, which is a more detailed, and fine-grained result, but developers have to interpret the results themselves. Nguyen's work does not mention any usage of dynamic analysis or repository mining for providing more fine-grained results or the sources responsible for the regressions.

Foo et al. [59] introduce an automatic approach to derive performance signatures by capturing the correlations among metrics in performance regression repositories and comparing new test results against these correlations. The reports signal potential problematic metrics that violate the extracted performance signatures. Performance analysts can leverage the report to ensure better coverage in their assessments of performance regression tests and to derive the causes. Foo's approach is able to reveal performance regressions related to different performance attributes, not only execution time, but disk, memory and CPU utilization. However, the performance analysts still need to derive the causes manually, what could be time-consuming. In our approach, the framework is able to indicate the deviations in more fine-grained way (methods) and the corresponding changes (commits and development issues).

Ghaith et al. [60] conducted an experiment to show that a transaction profile approach, which the authors consider a load independent representation of transaction response time, can detect performance anomalies when applied to two different releases of a web application. The first release was used as a baseline, while a known anomaly was added to the second one to cause extra processing. Despite the similarity of comparing performance of software releases, the possibility of discovering potential causes of performance deviation is not present in their work and there is no indication for future support.

Koziolek et al. [63] present a methodology to predict the quality attributes of performance and reliability using properties as response time and failure rate. The failure rate was calculated in the study by filtered the failure reports from the bug tracker database of the system, a large-scale control process system. The goal was to quantitatively predict the quality attributes for different architectural alternatives and then to choose the best alternative considering the trade-off among them. Their work differs from ours, which focuses on the analysis of existing system releases in order to detect existing performance deviation and their potential causes.

Huang et al. [78] suggest a new strategy for performance risk analysis to improve performance regression testing efficiency via testing target prioritization. They statically evaluate a given source code commit's risk in introducing performance regression. Thus, the test for performance regression can use the analysis result to test commits with high risk first. The designed performance risk analysis relies on two estimations: the cost and the frequency. First, a cost model, which defines a basic cost table for different instructions based on its type and operands, is used to estimate the expensiveness of a change instruction. After that, the

frequency of the change instruction is also estimated. For example, if it is inside a loop which has non-determined count, it is considered to be executed frequently. With the two estimations, the change instruction is indexed into a risk matrix to assess the risk level. Identifying the source code assets, commits and development issues responsible for performance deviation is not the focus of this related work, but its goal is similar to our study to determine the commit properties more likely to cause degradation. The difference is that the current related work does not consider properties of commits, but some metrics extracted from the source code with static analysis to prioritize the evaluation of commits by risk. It can only be done after the commit was made, while an improved version of our regression model could warn the developer before doing the commit.

8.5. Software Energy Consumption

Finally, there are some research work focused on the impact of changes on software energy consumption [61] [62], which are also related to this work, since poor performance may increase software energy consumption. In this context, Hindle [61] has proposed a green mining methodology of relating software changes to power consumption. The main goal is to give recommendations based on past evidences extracted by looking at each change in a version control system and dynamically measuring its effects on power consumption and alerting developers before they make a software change that negatively affects power consumption. The power tests dynamically measure the resources used by the system in a global way by monitoring CPU, disk and memory usage of the entire system. There is no mention if the approach can also help developers by indicating the sources of the energy consumption increase when the changes have already been made and an energy consumption regression was introduced.

8.6. Conclusion

First, if we talk about the software erosion, the approach is not completely focused on this problem like some related approaches presented in Section 2.4, but it can be considered a strategy to minimize this problem. Second, none of the related approaches that focus on performance are able to indicate causes of performance deviation in the level of the source code, or even to indicate the methods of scenarios responsible for such deviation and the corresponding changes. However, our approach only considers the performance in terms of execution time, while other approaches, include different metrics as memory usage, CPU usage, disk usage, power consumption, and others.

The conclusion is that there is still a gap of approaches and supporting tools to improve the performance analysis over multiple evolutions during all life-cycle of software systems. And, despite the differences, these research works present many similarities of techniques, for example, the use of static analysis and dynamic analysis to assemble call graphs, and the mining software repository support.

9. CONCLUSION

This research work presented a scenario-based approach for performance deviation analysis in terms of execution time and traceability of quality attributes. A framework to automate the proposed approach was also implemented with Java and AspectJ programming languages. It enables the analysis of software systems implemented in the Java language by combining static analysis, dynamic analysis, and software repository mining. The approach indicates scenarios with performance deviation, the responsible methods and the associated changes, including commits and development issues. Through the results of the conducted studies, we expect to help developers to detect performance issues before releasing software systems, making easier the process of fixing these issues by identifying their causes. Thus, the framework can be used as a preventive tool. Such results indicate that the approach is feasible and useful for helping developers to identify and understand the reasons of performance problems because it is able to substantially reduce the amount of information they have to analyze manually.

This chapter is organized as follows: **Section 9.1** presents a review for each conducted study; **Section 9.2** discusses the limitations of the approach, its support framework, and the conducted studies; **Section 9.3** revisits the proposed general research questions of this work; and **Section 9.4** presents some perspectives of future work and improvements for the current approach and framework.

9.1. Studies Review

In the study from **Chapter 5**, we presented an exploratory analysis that aimed to perform a scenario-based evaluation of the quality attribute of performance in terms of execution time for two releases of three software systems from different domains. The study allowed us to determine the possible causes of the scenario degradations for performance. The kind of issue with more occurrences was bug fixing (39 issues), followed by improvement (14 issues). Some explanation for the high number of bug fixing issues could be that they usually add new code to existing scenarios to execute new validations or to cover possibilities that were not addressed before. This study has evaluated the approach and support framework for systems from different domain, including desktop (ArgoUML), communication framework (Netty), and enterprise web information system (SIGAA).

The evolutionary study from **Chapter 6** has performed the analysis over multiples releases of Netty, Wicket, and Jetty. We found 13 changed-degraded scenarios out of 20 analyzed scenarios for Netty, 6 out of 16 for Wicket, and 6 out of 21 for Jetty. The potential causes of these deviations were found in the form of methods, commits and issues. This study shows that the approach was able to identify scenarios with performance deviation that developers were unaware of and it also identified the classes and packages that contain most of the sources, what is a strong indication that the approach is useful to help developers to identify and understand the reasons of performance deviations.

Finally, the study from **Chapter 7** has focused on the relationship between the quality attribute of performance and code changes properties introduced during the system evolution. The goal was to improve the understanding of the performance deviation phenomenon by studying specific characteristics of commits that can affect the likelihood of introducing this problem. The study proposed the definition of a regression model that aims to explain the phenomenon of performance deviation based on code changes (commit) properties. Based on the regression model, we found that the variables number of days before release and days of the week are the most influential properties to determine performance degradation. In addition, our regression model had ROC area of 60% and, apparently, there is no relationship between the size of the commit and performance deviations.

9.2. Discussion and Approach Limitations

9.2.1. Architecture Evaluation Effectiveness

The studies presented by this work for performance deviation analysis have successfully identified sources of performance degradation and optimization in evolutions of real systems by using a framework which implements a scenario-based approach. We consider this kind of analysis is architectural because it is guided by scenarios. Scenarios are high-level and relevant architectural concepts addressed by many methods of software architecture evaluation. As discussed in the studies, the framework application could indicate the sources of deviations in different granularity, for example, methods, classes and packages. In general, classes or packages can be mapped to components in order to identify the high-level assets of the system that contain performance deviations. The ability to find out the sources of the deviations and its support for such high-level concepts distinguishes the proposed approach from other ones which just use benchmarks to test isolated source code elements.

9.2.2. Quality Attribute Compliance

For the studies presented, we assume that the collected values of the performance quality attribute for the initial releases meet the requirements of the system. However, even in the cases when it is not true, it should not compromise the evolution analysis because the approach aims only to determine if the scenarios have performance deviation or not. For example, whether the initial release is already degraded compared to expected values, we will just discover that the next release is even more degraded, or that eventually have some optimizations. However, there is always the opportunity to analyze previous releases to understand and recover the code changes responsible for the found problems. It is also possible to check out if the values of the supported quality attributes belonging to an expected range informed in the annotations, but in this case, we can just perform an architectural compliance checking because there is no previous version to compare and mine the changes.

9.2.3. AspectJ Instrumentation

Our framework uses AspectJ to instrument the execution of scenarios, intercepting the entry point methods – annotated with `@Scenario` in the first phase of the approach (see Subsections 3.2.1 and 4.1.1) – to build the call graph and collect the execution time of scenarios and methods. The use of AspectJ was the most appropriate way we found to implement a solution capable to handle with the complexity of dynamic analysis and it is a common approach adopted by other studies [37] [38]. As discussed before in Subsection 4.1.2, abstract definitions of entry points can be implemented with AspectJ. It provides an interesting possibility for us because it enables the definition of different strategies to identify entry point methods.

9.2.4. Approach Execution Challenges

We recognize that our approach and current framework implementation have some execution challenges. The first one is the need for the manual annotation of the scenarios (when not reusing JUnit annotations or other existing ones, for example), which requires architectural knowledge of the target software. For systems with automated functional tests, these can be used as evaluation scenarios. Another requirement is the availability of all code artifacts in different versions as well as traceability data between development issues and commits. Thus, despite the capacity of the support framework to automate the approach application, the release preparation phase is still a manual procedure.

9.2.5. Execution Time Limitations

The performance was measured only in terms of execution time. Other possible metrics for performance are memory consumption, disk activity, and CPU usage, for example. These metrics are not being addressed by this work, but we recognize that for some systems, memory, for example, might be a more relevant performance requirement. Another problem is that new lines of code caused by the addition of new features or bug fixes might potentially increase the execution time, which will be detected by the framework. However, it is important to realize that some deviations cannot be avoided, and, in such cases, the developers need to decide if the increase is suitable or not, or even be aware of the impact of such changes for the system scenarios. We consider that the primary use case of the approach is when it finds deviations that developers did not expect. This will allow them to investigate the deviations further. In case of expected deviations, the approach can confirm exactly which scenarios were affected.

9.2.6. Discovering Issues

The framework searches for issues number inside commit logs. Usually, developers indicate the issue number using a pre-defined notation and it can be matched by a regular expression. If the developers forget to indicate the issue or they use a wrong pattern notation, the automate analysis cannot find it. To overcome this problem, we verified the commits in an analyzed period of interest to certificate that the implemented regular expressions can match the issue numbers indicated.

9.3. Research Questions Review

Based on the results obtained from the conducted studies, this section reviews the general research questions of this research work (see Section 1.2). They are:

RQ1. *How feasible is a scenario-based approach for automating the performance deviation analysis in terms of execution time considering source code assets and changes (commits and issues)?* Despite the limitations, presented in Section 9.2, and the threats to validity of each study, we consider that the results obtained from the approach application represent strong indications that the proposed approach and support framework are not only feasible, but also useful for developers according to the feedback from the online survey. Different from other current approaches, it was achieved by combining multiple techniques, including static analysis, dynamic analysis,

and mining software repository. For different target systems, being ArgoUML, Netty, Wicket, Jetty, and SIGAA, the framework was able to identify scenarios and their methods with performance deviation in terms of execution. In addition, it was also able to retrieve commits and development issues responsible for changing these methods over the selected evolutions. Finally, such results were reused to build a performance regression model in order to determine the most influential commit properties to cause performance degradation, which is another application of the proposed approach.

RQ2. *Did the scenarios that exhibit performance deviation also changed in terms of implementation?* As discussed before, an important concern related with the proposed approach is that it could find degraded or optimized scenarios that had no source code modification. If it happened, it would be an indicator that scenarios are being deviated because of external factors, for example, different libraries, settings from the runtime environment, or even any measurement problem, and not because their source code has truly changed. Fortunately, the framework shows that most of the scenarios detected with a significant performance deviation, considering a pre-configured threshold for Netty, Wicket, and Jetty, can be associated to code changes (see Table 6, Table 7, and Table 8), it means that the scenario has modifications introduced by at least one commit over a particular evolution.

RQ3. *Can automated functional tests be used to exercise scenarios of interest in order to measure performance in terms of execution time?* To make easier the release preparation phase and when performing the dynamic analysis phase, the `@Test` annotation from JUnit 4 was reused in most of the cases. Some developers, which have participated of the online survey, were concerned because such tests are not designed for performance analysis. We recognize that some tests cannot be used in this particular situation. For example, they should not use random inputs, but they have to use the same inputs in both release, and the tests should ideally cover functionalities and not only a single method since the approach is scenario-based. Despite the problems that may happen, we conclude that automated functional tests can be used with such purposes, since these particularities are addressed. For the studies of this work, for example, we only selected the tests after a manual review to ensure that these problems will not happen.

RQ4. *Can we accurately predict when commits may cause performance degradation based on their most influential properties?* In Chapter 7, we reused results from chapters 5 and 6 to build a performance regression model in order to indicate the properties of code changes (commits) that are more likely to cause performance degradation. Thus, a

subset of commit properties was used as explanatory variables in our regression model (see Table 13). In order to build such model, we mined a total of 997 commits, including 103 from degraded code assets, 19 from optimized, and 875 without impact on execution time. The explanatory variables *number of days before release* and *day of week* were the most relevant variables of commits that cause performance degradation in our model. It had a Receiver Operating Characteristic (ROC) area of 60%, what means that deciding if a commit will cause performance degradation or not by using the model is 10% better than a random guess. We recognize that our regression model needs improvements, what will require to increase the number of commits in our dataset and explore new ways of applying our approach, for example, adopting a per-commit evaluation strategy instead of a per-release strategy, as discussed in section 7.6.

9.4. Future Work

We are planning to work on some directions to improve the approach, its implementation, and the obtained results.

9.4.1. Approach Deployment in a Software Company

We are planning to use the approach in the development process of a software company. The approach was applied in some open source systems from different domains, such as, ArgoUML, Netty, Wicket and Jetty, and in SIGAA, an enterprise web information for academic management. However, every application was conducted by people who have also developed the approach and implemented the framework. We would like to analyze how other people will deal with the framework and how to introduce the current approach into an established development process of an existing company, for example, the Superintendence of Informatics (SINFO) of the Federal University of Rio Grande do Norte (UFRN).

Some work was already accomplished in this direction to evaluate the scenario responsible for student registration in courses, which happens once in a semester. The approach was used by SINFO developers to analyze and discover the most time-consumption methods since this scenario has very critical requirements for performance. However, due to particular interest of the company and deadline restrictions, only the first (release preparation) and second (dynamic analysis) phases of the approach were performed.

9.4.2. Instrumentation Impact

We are also planning to conduct new studies to measure the impact of the instrumentation process during system execution. We know that the instrumentation process increases the original execution time of the intercepted scenarios, but we have not conducted any study to measure such impact yet. On the other hand, we also believe it does not represent a critical problem since our analysis compares pairs of values of execution time from two releases, and the instrumentation should cause the same or very similar increases in both values.

In order to determine this impact, we are going to execute the scenarios with the instrumentation disabled to determine the real time they take to run. After that, the real time without instrumentation will be compared to the time the scenarios take to run in the exact same condition, but with the instrumentation enabled.

9.4.3. Additional Quality Attributes and Properties

The performance was measured only in terms of execution time. We recognize that it may not be enough for some systems. As a future work, we would like to include support for new performance properties – memory consumption, disk activity, and CPU usage – or even to add support for new quality attributes, such as, reliability.

The challenge about other properties related to performance is how to measure them in the granularity of methods since our mining phase searches for changes related to methods associated with degradation or optimization. Here, measuring the global use of memory consumption, for example, during the execution of a particular task is not suitable for the purposes of our approach. We are currently analyzing how our current aspect, implemented with AspectJ, could be extended to consider other properties. However, we still need to figure out how to handle it.

Reliability could be quantified by using the failure rate metric. In this context, an adaptation of our current approach and framework can analyze logs generated during the real usage of the system in order to determine the methods that have failed and how they have impacted their scenarios. Thus, we could provide feedback about the methods usually responsible for breaking their scenarios and indicate the changes introduced to them during the target evolution in order to help developers to understand and fix the problem.

9.4.4. Quality Attribute Traceability

Regarding traceability, as discussed in chapters 3 and 4, we have conducted only very preliminary studies. Thus, improving the static analysis support for quality attribute traceability and tradeoff detection, and applying it to analyze large-scale systems is an interesting path to explore in future. As said before, some research with initial results has already been published [85] [86] [87] [88].

In this context, we could use data extracted from static and dynamic analysis to more accurately determine scenarios with tradeoffs among quality attributes of interest, such as performance and security, which are a classic example of tradeoff. In this case, methods that implement particular design decisions that affect particular quality attributes could be identified – with annotations, for example – and static and dynamic analysis could be used to determine if such methods, which implement conflicting design decisions, will execute inside the same scenario of interest. The current support for such analysis is very simple because only static analysis is currently used and we still need to annotate the methods with the quality attributes they affect, and to define annotations for design decisions.

Another interesting possibility for quality attribute traceability is to identify code patterns in the source code that may lead the system to performance deviation. In this case, we could statically analyze the changes we found and try to identify similarities among the source code implemented for them.

9.4.5. Improvement of the Performance Regression Model

We are also planning to improve the performance regression model we have built. In this way, some points can be addressed. First, we can increase the amount of commits in our dataset by analyzing new evolutions or even new systems. Second, a per-commit strategy that evaluates each commit individually could be applied instead of a per-release application. It will demand much more time evaluating the system performance, but the results will be more accurate.

Finally, our performance regression model is discrete, it means that given a commit it will say “yes” or “no” for performance degradation. In this context, another interesting possibility to explore is to build a linear performance regression model by using the calculated property of execution time. In this case, we could estimate the degradation caused by the commit in terms of the execution time.

9.4.6. Framework Evaluation

Even with the results extracted from the evaluation of real software systems presented in this work as an evidence of the feasibility of the proposed approach and framework, we consider that more studies are still necessary. For example, in order to determine different situations of performance degradation which the framework is able or not to identify, we are planning to analyze controlled evolutions of a software system in which we will introduce some modifications that lead to performance degradation in terms of execution time. In this way, we can determine which are the specific situations when the framework fails and when it works properly.

Another interesting study is to compare the results obtained by our framework for the evaluation of these controlled evolutions and compare them to the results from other approaches. In case of the absence of similar approaches, we could compare intermediate results, for example, the impact to execute dynamic analysis and to build the dynamic call graph.

REFERENCES

[1] Lakshitha de Silva and Dharini Balasubramaniam. 2012. **Controlling software architecture erosion: A survey**. J. Syst. Softw. 85, 1 (January 2012), 132-151. DOI=10.1016/j.jss.2011.07.036 http://dx.doi.org/10.1016/j.jss.2011.07.036.

[2] Dewayne E. Perry and Alexander L. Wolf. 1992. **Foundations for the study of software architecture**. SIGSOFT Softw. Eng. Notes 17, 4 (October 1992), 40-52. DOI=10.1145/141874.141884 http://doi.acm.org/10.1145/141874.141884.

[3] Mary Shaw and David Garlan. 1996. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[4] Paul Clements, Rick Kazman and Mark Klein. **Evaluating Software Architectures: Methods and Case Studies**. 2001. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[5] Haroon Malik, Hadi Hemmati, and Ahmed E. Hassan. 2013. **Automatic detection of performance deviations in the load testing of large scale systems**. In Proceedings of the 2013 International Conference on Software Engineering (ICSE'13). IEEE Press, Piscataway, NJ, USA, 1012-1021.

[6] Abram Hindle. 2012. **Green mining: a methodology of relating software change to power consumption**. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'12). IEEE Press, Piscataway, NJ, USA, 78-87.

[7] Ricardo Pérez-Castillo and Mario Piattini. **Analyzing the Harmful Effect of God Class Refactoring on Power Consumption**. IEEE Software, vol.31, no. 3, pp. 48-54, May-June 2014.

[8] Muhammad Ali Babar and Ian Gorton. 2004. **Comparison of Scenario-Based Software Architecture Evaluation Methods**. In Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04). IEEE Computer Society, Washington, DC, USA, 600-607. DOI=10.1109/APSEC.2004.38 http://dx.doi.org/10.1109/APSEC.2004.38.

[9] Richard N. Taylor, Nenad Medvidović and Eric M. Dashofy. **Software Architecture: Foundations, Theory, and Practice**. Wiley Publishing (2009).

[10] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. 2004. **Architecture-level modifiability analysis (ALMA)**. J. Syst. Softw. 69, 1-2 (January 2004), 129-147. DOI=10.1016/S0164-1212(03)00080-3 http://dx.doi.org/10.1016/S0164-1212(03)00080-3.

[11] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. 1996. **Scenario-Based Analysis of Software Architecture**. IEEE Softw. 13, 6 (November 1996), 47-55. DOI=10.1109/52.542294 http://dx.doi.org/10.1109/52.542294.

[12] Banani Roy and T. C. Nicholas Graham. **Methods for Evaluating Software Architecture: A Survey**. Technical Report 2008-545, School of Computing, Queen's University at Kingston. Canada, April, 2008.

[13] Ackermann, C.; Lindvall, M.; Cleaveland, R. (2009). **Towards behavioral reflexion models**. In Proceedings of the 20th IEEE international conference on software reliability engineering (ISSRE'09), Andrew Podgurski and Pankaj Jalote (Eds.). IEEE Press, Piscataway, NJ, USA, 175-184.

[14] Ganesan, D.; Lindvall, M.; Ruley, L; Wiegand, R.; Ly, V.; Tsui, T. (2010). **Architectural Analysis of Systems Based on the Publisher-Subscriber Style**. In Proceedings of the 2010 17th Working Conference on Reverse Engineering (WCRE '10). IEEE Computer Society, Washington, DC, USA, 173-182.

[15] Ganesan, D.; Lindvall, M.; Cleaveland, R.; Jetley, R.; Jones, P.; Zhang, Y. (2011). **Architecture Reconstruction and Analysis of Medical Device Software**. In Proceedings of the 2011 Ninth Working IEEE/IFIP Conference on Software Architecture (WICSA '11). IEEE Computer Society, Washington, DC, USA, 194-203.

[16] Ganesan, D.; Keuler, T.; Nishimura, Y. (2009). **Architecture compliance checking at run-time**. Inf. Softw. Technol. 51, 11 (November 2009), 1586-1600.

[17] Gokhale, S. S. (2007). **Architecture-Based Software Reliability Analysis: Overview and Limitations**. IEEE Trans. Dependable Secur. Comput. 4, 1 (January 2007), 32-40.

[18] G. Williams, L. G.; Smith, C. U. (2002). **PASASM: a method for the performance assessment of software architectures**. In Proceedings of the 3rd international workshop on Software and performance (WOSP'02). ACM, New York, NY, USA, 179-189.

[19] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. **Discovering, reporting, and fixing performance bugs**. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13). IEEE Press, Piscataway, NJ, USA, 237-246.

[20] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. 2014. **An industrial case study of automatically identifying performance regression-causes**. In Proceedings of the 11th Working Conference on Mining Software Repositories (MSR 2014). ACM, New York, NY, USA, 232-241. DOI=10.1145/2597073.2597092 http://doi.acm.org/10.1145/2597073.2597092.

[21] **JUnit Framework**. Available at http://junit.org (August 2015).

[22] **T.J. Watson Libraries for Analysis (WALA)**. Available at http://wala.sourceforge.net (Augusto 2015).

[23] **Eclipse Java development tools (JDT)**. Available at http://www.eclipse.org/jdt (Augusto 2015).

[24] **Apache Subversion**. Available at http://subversion.apache.org (Augusto 2015).

[25] **Git**. Available at http://git-scm.com (Augusto 2015).

[26] **PostgreSQL Database**. Available at http://www.postgresql.org (Augusto 2015).

[27] **Informatics Superintendence from Federal University of Rio Grande do Norte (UFRN)**. Available at http://www.info.ufrn.br/wikisistemas (August 2015).

[28] **Netty**. Available at http://netty.io (August 2015).

[29] **ArgoUML**. Available at http://argouml.tigris.org (August 2015).

[30] **Apache Wicket**. Available at http://wicket.apache.org (August 2015).

[31] **Jetty**. Available at http://www.eclipse.org/jetty (August 2015).

[32] Len Bass, Paul Clements, and Rick Kazman. 2003. **Software Architecture in Practice.** 2 ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[33] Kazman, R.; Klein, M.; Clements, P. 2000. **ATAM: Method for Architecture Evaluation**. Technical Report. CMU/SEI-2000-TR-004.

[34] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. 1982. **Gprof: A call graph execution profiler**. In Proceedings of the 1982 SIGPLAN symposium on Compiler construction (SIGPLAN'82). ACM, New York, NY, USA, 120-126. DOI=10.1145/800230.806987 http://doi.acm.org/10.1145/800230.806987.

[35] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. **Call graph construction in object-oriented languages**. In Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), A. Michael Berman (Ed.). ACM, New York, NY, USA, 108-124. DOI=10.1145/263698.264352 http://doi.acm.org/10.1145/263698.264352.

[36] David Grove and Craig Chambers. 2001. **A framework for call graph construction algorithms**. ACM Trans. Program. Lang. Syst. 23, 6 (November 2001), 685-746. DOI=10.1145/506315.506316 http://doi.acm.org/10.1145/506315.506316.

[37] Reid Holmes and David Notkin. 2011. **Identifying program, test, and environmental changes that affect behaviour**. In Proceedings of the 33rd International Conference on Software Engineering (ICSE'11). ACM, New York, NY, USA, 371-380. DOI=10.1145/1985793.1985844 http://doi.acm.org/10.1145/1985793.1985844.

[38] Selim Ciraci, Hasan Sozer, and Bedir Tekinerdogan. 2012. **An Approach for Detecting Inconsistencies between Behavioral Models of the Software Architecture and the Code**. In Proceedings of the 2012 IEEE 36th Annual Computer Software and Applications Conference (COMPSAC'12). IEEE Computer Society, Washington, DC, USA, 257-266. DOI=10.1109/COMPSAC.2012.36 http://dx.doi.org/10.1109/COMPSAC.2012.36.

[39] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. 2009. **Inferred call path profiling**. In Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09). ACM, New York,

NY, USA, 175-190. DOI=10.1145/1640089.1640102 http://doi.acm.org/10.1145/1640089.1640102.

[40] K.K. Chaturvedi, V.B. Singh, and Prashast Singh. 2013. **Tools in Mining Software Repositories**. In proceedings of the 13th International Conference on Computational Science and Its Applications (ICCSA'13). IEEE. 89-98. DOI=10.1109/ICCSA.2013.22 http://dx.doi.org/10.1109/ICCSA.2013.22.

[41] Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. 2007. **A survey and taxonomy of approaches for mining software repositories in the context of software evolution**. J. Softw. Maint. Evol. 19, 2 (March 2007), 77-131. DOI=10.1002/smr.344 http://dx.doi.org/10.1002/smr.344.

[42] Ahmed E. Hassan. 2008. **The Road Ahead for Mining Software Repositories**. In proceedings of the Frontiers of Software Maintenance (FoSM 2008). IEEE, Beijing, 48-57, Sept. 28 2008-Oct. 4 2008. DOI=10.1109/FOSM.2008.4659248 http://dx.doi.org/10.1109/FOSM.2008.4659248.

[43] Kwan, I.; Damian, D. 2011. **A Survey of Techniques in Software Repository Mining**. Technical Report DCS-340-IR, University of Victoria.

[44] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. **Who should fix this bug?**. In Proceedings of the 28th international conference on Software engineering (ICSE'06). ACM, New York, NY, USA, 361-370. DOI=10.1145/1134285.1134336 http://doi.acm.org/10.1145/1134285.1134336.

[45] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. 2007. **Mining Software Evolution to Predict Refactoring**. In Proceedings of the First International Symposium on Empirical Software Engineering and Measurement (ESEM'07). IEEE Computer Society, Washington, DC, USA, 354-363. DOI=10.1109/ESEM.2007.63 http://dx.doi.org/10.1109/ESEM.2007.63.

[46] Gregorio Robles, Jesus M. Gonzalez-Barahona, Martin Michlmayr, and Juan Jose Amor. 2006. **Mining large software compilations over time: another perspective of software evolution**. In Proceedings of the 2006 international workshop on Mining software repositories (MSR'06). ACM, New York, NY, USA, 3-9. DOI=10.1145/1137983.1137986 http://doi.acm.org/10.1145/1137983.1137986.

[47] Morrison, R.; Kirby, G.; Balasubramaniam, D.; Mickan, K. 2004. **Support for evolving software architectures in the ArchWare ADL**. In Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04). IEEE Computer Society, Washington, DC, USA.

[48] Roshanak Roshandel, André Van Der Hoek, Marija Mikic-Rakic, and Nenad Medvidovic. 2004. **Mae – A system model and environment for managing architectural evolution**. ACM Trans. Softw. Eng. Methodol. 13, 2 (April 2004), 240-276. DOI=10.1145/1018210.1018213 http://doi.acm.org/10.1145/1018210.1018213.

[49] Eugen C. Nistor, Justin R. Erenkrantz, Scott A. Hendrickson, and André van der Hoek. 2005. **ArchEvol: versioning architectural-implementation relationships**. In Proceedings of the 12th international workshop on Software configuration management (SCM'05). ACM, New York, NY, USA, 99-111. DOI=10.1145/1109128.1109136 http://doi.acm.org/10.1145/1109128.1109136.

[50] University of California Irvine. **Software and Systems Architecture Development Environment**. Available at http://isr.uci.edu/projects/archstudio (September 2015).

[51] Nguyen, T. N.; Munson, E. V.; Boyland, J. T.; Thao, C. (2004). **Architectural Software Configuration Management in Molhado**. In Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04). IEEE Computer Society, Washington, DC, USA, 296-305. DOI= 10.1109/ICSM.2004.1357815 http://dx.doi.org/10.1109/ICSM.2004.1357815.

[52] Ciaran O'Reilly, Philip Morrow, and David Bustard. 2003. **Lightweight Prevention of Architectural Erosion**. In Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE'03). IEEE Computer Society, Washington, DC, USA.

[53] **Apache JMeter**. Available at http://jmeter.apache.org (August 2015).

[54] **Automating the Assessment of the Performance Quality Attribute for Evolving Software Systems: An Exploratory Study**. Available at https://sites.google.com/site/experfstudy (August 2015).

[55] **Automating the Performance Deviation Analysis for Multiple System Releases: an Evolutionary Study**. Available at https://sites.google.com/site/perfevolutionarystudy (August 2015).

[56] Christensen, H. B.; Hansen, K M. 2011. **Towards architectural information in implementation** (NIER track). In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 928-931. DOI=10.1145/1985793.1985948 http://doi.acm.org/10.1145/1985793.1985948.

[57] Mirakhorli, M.; Shin, Y.; Cleland-Huang, J.; Cinar, M. 2012. **A tactic-centric approach for automating traceability of quality concerns**. In Proceedings of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, Piscataway, NJ, USA, 639-649.

[58] Tibermacine, C.; Zernadji, T. 2011. **Supervising the evolution of web service orchestrations using quality requirements**. In Proceedings of the 5th European conference on Software architecture (ECSA'11), Ivica Crnkovic, Volker Gruhn, and Matthias Book (Eds.). Springer-Verlag, Berlin, Heidelberg, 1-16.

[59] Foo, K. C.; Jiang, Z. M.; Adams, B.; Hassan, A. E.; Zou, Y.; Flora, P. 2010. **Mining Performance Regression Testing Repositories for Automated Performance Analysis**. In Proceedings of the 10th International Conference on Quality Software (QSIC 2010). IEEE Computer Society, Washington, DC, USA, 32-41.

[60] Ghaith, S.; Wang, M.; Perry, P.; Murphy, J. 2013. **Profile-Based, Load-Independent Anomaly Detection and Analysis in Performance Regression Testing of Software Systems**. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13). IEEE Computer Society, Washington, DC, USA, 379-383. DOI=10.1109/CSMR.2013.54 http://dx.doi.org/10.1109/CSMR.2013.54.

[61] Abram Hindle. 2012. **Green mining: a methodology of relating software change to power consumption**. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12). IEEE Press, Piscataway, NJ, USA, 78-87.

[62] R. Perez-Castillo, M. Piattini. 2014. **Analyzing the Harmful Effect of God Class Refactoring on Power Consumption**. IEEE Software. 31, 3 (April 2014), 48-54.

[63] Heiko Koziolek, Bastian Schlich, Steffen Becker, and Michael Hauck. 2013. **Performance and reliability prediction for evolving service-oriented software systems**. Empirical Softw. Eng. 18, 4 (August 2013), 746-790. DOI=10.1007/s10664-012-9213-0 http://dx.doi.org/10.1007/s10664-012-9213-0.

[64] Jon Eyolfson, Lin Tan, and Patrick Lam. 2011. **Do time of day and developer experience affect commit bugginess?**. In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11). ACM, New York, NY, USA, 153-162. DOI=10.1145/1985441.1985464 http://doi.acm.org/10.1145/1985441.1985464.

[65] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. **When do changes induce fixes?**. In Proceedings of the 2005 international workshop on Mining software repositories (MSR '05). ACM, New York, NY, USA, 1-5. DOI=10.1145/1082983.1083147 http://doi.acm.org/10.1145/1082983.1083147.

[66] Daryl Posnett, Raissa D'Souza, Premkumar Devanbu, and Vladimir Filkov. 2013. **Dual ecological measures of focus in software development**. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 452-461.

[67] Foyzur Rahman and Premkumar Devanbu. 2011. **Ownership, experience and defects: a fine-grained study of authorship**. In Proceedings of the 33rd International Conference on Software Engineering (ICSE '11). ACM, New York, NY, USA, 491-500. DOI=10.1145/1985793.1985860 http://doi.acm.org/10.1145/1985793.1985860.

[68] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. 2008. **Classifying Software Changes: Clean or Buggy?**. IEEE Trans. Softw. Eng. 34, 2 (March 2008), 181-196. DOI=10.1109/TSE.2007.70773 http://dx.doi.org/10.1109/TSE.2007.70773.

[69] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. **A qualitative study on performance bugs**. In Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR '12). IEEE Press, Piscataway, NJ, USA, 199-208.

[70] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. 2009. **Software Dependencies, Work Dependencies, and Their Impact on Failures**. IEEE Trans. Softw. Eng. 35, 6 (November 2009), 864-878. DOI=10.1109/TSE.2009.42 http://dx.doi.org/10.1109/TSE.2009.42.

[71] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. 2000. **Predicting Fault Incidence Using Software Change History**. IEEE Trans. Softw. Eng. 26, 7 (July 2000), 653-661. DOI=10.1109/32.859533 http://dx.doi.org/10.1109/32.859533.

[72] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. 2008. **A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction**. In Proceedings of the 30th international conference on Software engineering (ICSE '08). ACM, New York, NY, USA, 181-190. DOI=10.1145/1368088.1368114 http://doi.acm.org/10.1145/1368088.1368114.

[73] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. 2004. **Securing web application code by static analysis and runtime protection**. In Proceedings of the 13th international conference on World Wide Web (WWW '04). ACM, New York, NY, USA, 40-52. DOI=10.1145/988672.988679 http://doi.acm.org/10.1145/988672.988679.

[74] Prasanth Anbalagan and Mladen Vouk. 2009. **An empirical study of security problem reports in Linux distributions**. In Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM '09). IEEE Computer Society, Washington, DC, USA, 481-484. DOI=10.1109/ESEM.2009.5315985 http://dx.doi.org/10.1109/ESEM.2009.5315985.

[75] Wilson, C. E.; Coyne, K. P. 2001. **The whiteboard: Tracking usability issues: to bug or not to bug?**. Interactions 8, 3 (May 2001), 15-19. DOI=10.1145/369825.369828 http://doi.acm.org/10.1145/369825.369828.

[76] Florian Heller, Leonhard Lichtschlag, Moritz Wittenhagen, Thorsten Karrer, and Jan Borchers. 2011. **Me hates this: exploring different levels of user feedback for (usability) bug reporting**. In CHI '11 Extended Abstracts on Human Factors in Computing Systems (CHI EA '11). ACM, New York, NY, USA, 1357-1362. DOI=10.1145/1979742.1979774 http://doi.acm.org/10.1145/1979742.1979774.

[77] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. 2008. **What's a Typical Commit? A Characterization of Open Source Software Repositories**. In Proceedings of the 2008 The 16th IEEE International Conference on Program

Comprehension (ICPC '08). IEEE Computer Society, Washington, DC, USA, 182-191. DOI=10.1109/ICPC.2008.24 http://dx.doi.org/10.1109/ICPC.2008.24.

[78] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. **Performance regression testing target prioritization via performance risk analysis**. In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014). ACM, New York, NY, USA, 60-71. DOI=10.1145/2568225.2568232 http://doi.acm.org/10.1145/2568225.2568232.

[79] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. **Producing wrong data without doing anything obviously wrong!**. In Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS XIV). ACM, New York, NY, USA, 265-276. DOI=10.1145/1508244.1508275 http://doi.acm.org/10.1145/1508244.1508275.

[80] Frank E. Harrell, Jr.. 2006. **Regression Modeling Strategies**. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[81] Efron, B.; Tibshirani, R. 1986. **Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy**. Statistical Science, vol. 1, pp. 54-75, 1986.

[82] Markus Neuhäuser. **International Encyclopedia of Statistical Science: Wilcoxon–Mann–Whitney Test**. Lovric, Miodrag, 2014. ISBN 978-3-642-04897-5. Pages: 1656-1658.

[83] Jacek Sliwerski, Thomas Zimmermann, Andreas Zeller. **When do Changes Induce Fixes?**. In Proceedings of the Second International Workshop on Mining Software Repositories (MSR 2005), St. Louis, MO, USA, May 2005, pp. 24-28.

[84] Jacek Sliwerski, Thomas Zimmermann, Andreas Zeller. **Don't Program on Fridays! How to Locate Fix-Inducing Changes**. In Proceedings of the 7th Workshop Software Reengineering (WSR 2005), Bad Honnef, Germany, May 2005. Proceedings also appeared in Softwaretechnik-Trends (25:2), published by the Gesellschaft für Informatik (GI).

[85] Felipe Pinto, Uirá Kulesza, Eduardo Guerra. (2013). **Automating the Architecture Evaluation of Enterprise Information Systems**. In Proceedings of the 15th International Conference on Enterprise Information Systems, ISBN 978-989-8565-61-7, pages 333-340.

[86] Felipe Pinto, Uirá Kulesza, Eduardo Guerra, João Maria Júnior, and Leo Silva. **Automated architectural evaluation of web information systems**. In Proceedings of the 19th Brazilian symposium on Multimedia and the web (WebMedia'13). 2013, Salvador, Brazil, p. 225-232. DOI = http://dx.doi.org/10.1145/2526188.2526193.

[87] Felipe Pinto, Uirá Kulesza, Leo Silva, and Eduardo Guerra. **A Metadata-based Framework for Quality Attribute Degradation Analysis in Web Systems**. In Proceedings of the 20th Brazilian Symposium on Multimedia and the Web (WebMedia'14). 2014, João Pessoa, Brazil, p. 171-178. DOI = http://dx.doi.org/10.1145/2664551.2664570.

[88] Felipe Pinto, Uirá Kulesza, Eduardo Guerra. **An Automated Architectural Evaluation Approach Based on Metadata and Code Analysis**. Lecture Notes in Business Information Processing. 1ed.: Springer International Publishing, 2014, v. 190, p. 490-505.

[89] Felipe Pinto, Uirá Kulesza, Leo Silva, and Eduardo Guerra. **Automating the Assessment of the Performance Quality Attribute for Evolving Software Systems: An Exploratory Study**. In proceedings of 48th Hawaii International Conference on System Sciences (HICSS), 2015, Kauai, HI, p. 5144-5153, DOI = http://dx.doi.org/10.1109/HICSS.2015.608.

[90] Felipe Pinto, Uirá Kulesza, and Christoph Treude. **Automating the Performance Deviation Analysis for Multiple System Releases: an Evolutionary Study**. In proceedings of 15th IEEE International Working Conference on Source Code Analysis and Manipulation, 2015, Bremen, p. 201-210. DOI = http://dx.doi.org/10.1109/SCAM.2015.7335416.

APPENDIX A – MODULES RESPONSIBLE FOR PERFORMANCE DEGRADATION

This appendix is complementary to the study presented in **Chapter 5** entitled *"Evaluation for Performance Deviation Analysis"*. Table 14, Table 15 and Table 16 show how the methods flagged by the framework as potential responsible for performance degradations are spread over the classes and packages of each target system, considering SIGAA, ArgoUML and Netty, respectively.

Table 14. SIGAA: classes that contain degraded and changed methods.

#	Classes	Number of Methods
1	websystem.core.dao.GenericDAOImpl	2
3	websystem.core.util.Utils	2
2	websystem.core.security.SessionLogger	1
4	websystem.sig.library.PerformLoanService	1
5	websystem.sig.library.RenewLoanService	1
6	websystem.sig.library.LibraryServiceUtil	1
7	websystem.sig.domain.Course	1
	TOTAL:	9

Table 15. ArgoUML: classes that contain degraded and changed methods.

#	Classes	Number of Methods
1	org.argouml.profile.UserDefinedProfile	10
2	org.argouml.model.mdr.XmiReferenceResolverImpl	7
3	org.argouml.model.mdr.CoreHelperMDRImpl	6
4	org.argouml.model.mdr.XmiReaderImpl	6
5	org.argouml.notation.NotationProvider	6
6	org.argouml.persistence.PGMLStackParser	6
7	org.argouml.persistence.UmlFilePersister	4
8	org.argouml.uml.diagram.ui.FigCompartmentBox	4
9	org.argouml.model.mdr.CollaborationsFactoryMDRImpl	3
10	org.argouml.model.mdr.DataTypesFactoryMDRImpl	3
11	org.argouml.model.mdr.FacadeMDRImpl	3
12	org.argouml.model.mdr.MDRModelImplementation	3
13	org.argouml.model.mdr.UmlFactoryMDRImpl	3
14	org.argouml.persistence.ModelMemberFilePersister	3
15	org.argouml.profile.internal.ProfileManagerImpl	3
16	org.argouml.profile.internal.ProfileMeta	3
17	org.argouml.profile.internal.ProfileUML	3
18	org.argouml.kernel.ProjectImpl	2
19	org.argouml.model.mdr.CommonBehaviorHelperMDRImpl	2
20	org.argouml.model.mdr.CoreFactoryMDRImpl	2
21	org.argouml.model.mdr.ModelEventPumpMDRImpl	2
22	org.argouml.notation.NotationProviderFactory2	2
23	org.argouml.notation.providers.uml.AttributeNotationUml	2

24	org.argouml.notation.providers.uml.NotationUtilityUml	2
25	org.argouml.uml.diagram.static_structure.ui.FigPackage	2
26	org.argouml.uml.diagram.ui.FigAssociation	2
27	org.argouml.uml.diagram.ui.FigEdgeModelElement	2
28	org.argouml.application.helpers.ApplicationVersion	1
29	org.argouml.application.helpers.ResourceLoader	1
30	org.argouml.i18n.Translator	1
31	org.argouml.kernel.ProfileConfiguration	1
32	org.argouml.kernel.ProjectManager	1
33	org.argouml.kernel.ProjectSettings	1
34	org.argouml.model.mdr.MDRModelImplementation$Extent	1
35	org.argouml.model.mdr.StateMachinesFactoryMDRImpl	1
36	org.argouml.model.mdr.UndoCoreHelperDecorator	1
37	org.argouml.notation.providers.ActionStateNotation	1
38	org.argouml.notation.providers.AssociationEndNameNotation	1
39	org.argouml.notation.providers.AssociationNameNotation	1
40	org.argouml.notation.providers.AssociationRoleNotation	1
41	org.argouml.notation.providers.AttributeNotation	1
42	org.argouml.notation.providers.ClassifierRoleNotation	1
43	org.argouml.notation.providers.MessageNotation	1
44	org.argouml.notation.providers.ModelElementNameNotation	1
45	org.argouml.notation.providers.NodeInstanceNotation	1
46	org.argouml.notation.providers.ObjectNotation	1
47	org.argouml.notation.providers.OperationNotation	1
48	org.argouml.notation.providers.StateBodyNotation	1
49	org.argouml.notation.providers.TransitionNotation	1
50	org.argouml.notation.providers.uml.ModelElementNameNotationUml	1
51	org.argouml.notation.providers.uml.TransitionNotationUml	1
52	org.argouml.persistence.XmiFilePersister	1
53	org.argouml.persistence.ZargoFilePersister	1
54	org.argouml.profile.Profile	1
55	org.argouml.profile.ProfileMother	1
56	org.argouml.profile.URLModelLoader	1
57	org.argouml.profile.internal.ocl.uml14.ModelAccessModelInterpreter	1
58	org.argouml.uml.StereotypeUtility	1
59	org.argouml.uml.UUIDHelper	1
60	org.argouml.uml.cognitive.critics.CrNoAssociations	1
61	org.argouml.uml.cognitive.critics.ProfileCodeGeneration	1
62	org.argouml.uml.cognitive.critics.ProfileGoodPractices	1
63	org.argouml.uml.diagram.activity.ui.FigActionState	1
64	org.argouml.uml.diagram.collaboration.ui.FigClassifierRole	1
65	org.argouml.uml.diagram.deployment.ui.AbstractFigNode	1
66	org.argouml.uml.diagram.deployment.ui.FigObject	1
67	org.argouml.uml.diagram.state.ui.FigBranchState	1
68	org.argouml.uml.diagram.state.ui.FigForkState	1
69	org.argouml.uml.diagram.state.ui.FigHistoryState	1
70	org.argouml.uml.diagram.state.ui.FigInitialState	1
71	org.argouml.uml.diagram.state.ui.FigJoinState	1
72	org.argouml.uml.diagram.state.ui.FigJunctionState	1
73	org.argouml.uml.diagram.state.ui.FigState	1
74	org.argouml.uml.diagram.static_structure.ui.FigClass	1
75	org.argouml.uml.diagram.static_structure.ui.FigComment	1
76	org.argouml.uml.diagram.static_structure.ui.FigEdgeNote	1

77	org.argouml.uml.diagram.static_structure.ui.FigLink	1
78	org.argouml.uml.diagram.static_structure.ui.FigPackage$PackageBackground	1
79	org.argouml.uml.diagram.ui.FigSingleLineTextWithNotation	1
80	org.argouml.uml.diagram.use_case.ui.FigActor	1
81	org.argouml.uml.diagram.use_case.ui.FigUseCase	1
82	org.argouml.uml.ui.ActionNewDiagram	1
83	org.argouml.uml.ui.ActionStateDiagram	1
84	org.argouml.util.CustomSeparator	1
	TOTAL:	153

Table 16. Netty: classes that contain degraded and changed methods.

#	Classes	Number of Methods
1	io.netty.channel.DefaultChannelHandlerContext	4
2	io.netty.channel.DefaultChannelConfig	3
3	io.netty.channel.ChannelOutboundBuffer	2
4	io.netty.channel.SingleThreadEventLoop	2
5	io.netty.channel.nio.NioEventLoop	2
6	io.netty.buffer.AbstractByteBufAllocator	1
7	io.netty.buffer.PooledByteBufAllocator	1
8	io.netty.buffer.PooledByteBufAllocator$PoolThreadLocalCache	1
9	io.netty.channel.AbstractChannel	1
10	io.netty.channel.ChannelOption	1
11	io.netty.channel.DefaultChannelHandlerContext$AbstractWriteTask	1
12	io.netty.channel.DefaultChannelHandlerContext$WriteAndFlushTask	1
13	io.netty.channel.ThreadPerChannelEventLoopGroup	1
14	io.netty.channel.socket.DefaultSocketChannelConfig	1
15	io.netty.channel.socket.nio.NioServerSocketChannel	1
16	io.netty.channel.socket.nio.NioServerSocketChannel$NioServerSocketChannelConfig	1
17	io.netty.channel.socket.nio.NioSocketChannel	1
18	io.netty.channel.socket.nio.NioSocketChannel$NioSocketChannelConfig	1
	TOTAL:	26

APPENDIX B – COMMITS AND TYPE OF ISSUES RELATED TO SOURCES OF PERFORMANCE DEVIATION

This appendix is complementary to the study presented in **Chapter 6** entitled *"Evolutionary Analysis"*. It presents the commits, issue numbers and type of issues related to sources of performance deviation which were found by the framework during the automated analysis. Table 17, Table 18 and Table 19 show the types of issues found for Netty, Wicket and Jetty, respectively.

Table 17. Type of issues for Netty.

Type	NE1	NE2	NE3	NE4	NE5	NE6	Total (Type)
Unlabeled	0	2	1	0	0	0	**3**
Improvement	1	1	0	0	2	0	**4**
Bug/Defect	0	0	2	0	0	0	**2**
New Feature	0	0	0	0	1	0	**1**
Total (Evolution):	**1**	**3**	**3**	**0**	**3**	**0**	**10**

Netty Evolution 1

```
Improvement: 1606
Number of revisions (commits): 4
Revision (Issues):
```

- 4cd7e625556f86eb25a4a4cfae20ed0d34aaf5ed(0)
- 25f96b164498988439c38baf6ff4c57763464824(1606)
- d5e202d7554f79a7a7d670e371486d66a9f75c9d(0)
- d4aa5b53d6410d543834f3d54841bd5eaf95dbee(0)

Netty Evolution 2

```
Improvement: 1707
Unlabeled: 1697;1832
Number of revisions (commits): 5
Revision (Issues):
```

- 013ac44d3a64d53ab9e131cb43124fcbc0873caf(1832)
- b934b6009c9e7d2ed6df400698d6e6e4b550291d(1707)
- fb619f23947aab158a11730e00f4dbd490093669(0)
- fc6213604dc3a1a596b5a6fb25fd9cf0759e0fc7(0)
- 48a7a21541b96eba74b493d8a2ce4a678a6bc9db(1697)

Netty Evolution 3:

```
Bug/Defect: 1908;2060
Unlabeled: 1947
Number of revisions (commits): 7
Revision (Issues):
```

- e4358ae6b8a8028de980f4fd41f0c93d3dbb40b2(1947)
- 65b522a2a75c78c977a65e15663509509b5f12e0(0)
- 8986245b47e738c004e9aa9bba56b8f973dd3dc4(0)
- 89a7cb8e710952e76a3a09b113fcb6ebe17acb12(2060)
- c9b7f1f1b570eab9eb6b61df0e94cd4dbcc5e2ec(1908)
- ba3bc0c0205ed37f0541e43dd747ebcc103f8ed8(0)

- 7dddbbb2bdc1a2ab5648e7d4b4b24d6f328eac6a(0)

Netty Evolution 5:

Improvement: 808;2264
New feature: 2311
Number of revisions (commits): 2
Revision (Issues):

- 68670ba19506f2f3ef24181ecc4fbb4961c7d8a8(2311)
- 13fd69e8712acd337570aad2c153cf4b2bc9b586(2264;808)

Netty Evolution 6:

Number of revisions (commits): 1
Revision (Issues):

- 9b468bc2758ead86690f633ee071e396c482c489(0)

Table 18. Type of issues for Wicket.

Type	WE1	WE2	WE3	WE4	WE5	WE6	Total (Type)
Bug/Defect	0	0	0	7	0	2	**9**
Improvement	0	0	0	4	0	1	**5**
Unlabeled	0	0	0	0	0	0	**0**
New Feature	0	0	0	1	0	0	**1**
Total (Evolution):	**0**	**0**	**0**	**12**	**0**	**3**	**15**

Wicket Evolution 4

New Feature: 3335
Improvement: 3347;5215;5410;5459
Bug/Defect: 4964;5294;5309;5398;5426;5484;5522
Number of revisions (commits): 38
Revision (Issues):

- fea2b8e4e0feb6a7374bcfe6de2da5d1a2feba20(0)
- fb45a781c24f17d7a44658061c4f17e95c4cd1ef(5426)
- ec84bb57fbfadce914a1af79b1c47c3267de5d4b(3335)
- b891bb8ddff1fcd16b3deb1044f9753bd878ce36(0)
- 4ab506431371e78ee5d1f5000200c74f89a97a94(5309)
- 28a1637874300d3d8feefecffdb9e8565e3171e9(0)
- da98a830ce75b94bd7885a759659d1bd7abcf193(3335)
- bd9bcd34d7356757fcd24a1f1a9f0520f1ea7b3f(5294)
- 2e57d8db33b88889413f1396dbd4389904cf7e7f(3335)
- 1c8dc6f77027a3b3bcc68e24b427daa7effd2aa2(0)
- fd7ab849202144b183c3cbd69c81b7e4e433be99(0)
- 6971b62d1a788e251e3aaabcf461fdd3e4a0eb4a(5215)
- f81ad2a53a7bcba321d3faa7f0b484b0bb431da9(0)
- 61b01295d0d52929178d058d48456cae4acbc3e7(3335)
- 9759e26e7558b713ed082e349cba2a6ed437725a(4964)
- 19e7c1cdc1acca652bd1b38e11db00d2f0302d61(5398)
- ac6688794a0d74a8787814370ee803c3b0efbaaf(0)
- 5b730c0b41d6261be5bdd7810cdc59ebebf1d1ef(5522)
- 0fbd643621f99f1ff03ddd834723b8d53cfd798a(0)
- abb316cea0a31edea61a25ea4c192507d5c7b390(0)
- 914b18d0f81e0df9d2f02fd7f2d3d71a49c2e22a(0)
- 22f1e048923cf5b6e020a81b66e0a8512c24fe79(3347)
- 58daafa556ccf334bd7d03c83eab9973d83fb497(0)
- 723d7d3c9ebe8336897d75d3754685384f017b3a(3335)

- ed8d1258740c859a3f00c08d395366e09198b6fa(0)
- 2008dfb7044f544d7b112cf4666dfacf42406b89(3335)
- 6f9821b81e2afeedebbe66496b85009c0c62f74c(5426)
- 612f6fb41322fa97547d02cfb4fdd4f20ba6892c(3335)
- ecdfc124453ca614178d33d0c119d4e71872cfef(5484)
- b5e4574b1b135249dee7860e1f52b2a2af34b641(5410)
- b54c663d01e37c51b6e773033705ab50d0a3fa77(3335)
- b9ddb88862f15be28dd0a74869489a804e4cda43(3335)
- acb4360abb8398d39f36f6f94b1711a98f072fbc(0)
- 3da1d656cd4c85efe0919d1a32c33b300ef35490(0)
- ffd7e27401d851337e1c08324c94a6f9611ded59(3335)
- 4950a773dc7e3d286c67cdc4611f3b9bd2b060af(0)
- 156dea5b3479805eec964b905847ce8e177cd560(3335)
- eba961e585fd0fef9002a04588a96b68dc9035b5(5459)

Wicket Evolution 6

Improvement: 5713
Bug/Defect: 5689;5730
Number of revisions (commits): 4
Revision (Issues):

- db5be6ab05545e1bb95f566c3aeb23e05cf93437(5730)
- bb9c1044e5f1ba8902aa69073ef9fb236802d277(0)
- e4262674d6dd347fb51a1454c63e5f03ed5f135e(5713)
- 2ac29d3065a448c00a2ba7ac4b8cd53ae6c2b91b(5689)

Table 19. Type of issues for Jetty.

Type	JE1	JE2	JE3	JE4	JE5	JE6	Total (Type)
Bug/Defect	0	0	0	0	1	0	**1**
Total (Evolution):	**0**	**0**	**0**	**0**	**1**	**0**	**1**

Jetty Evolution 5

Bug/Defect: 439375
Number of revisions (commits): 7
Revision (Issues):

- 6048343fee5f5fd9b845ccf12d285c69d5aae182(0)
- 17f46665df2e0b066b40a3889dbf3f56173e0554(0)
- aaa2e5c6c1d6d34f8068dc9a056a6b446fd05603(0)
- d45a5c861c88e35513b79e97fc56fd3a1fa5731d(0)
- 4d2a580c2c8ca054841697a9d8848bdceb1c97f9(439375)
- 33435fad3d8e1e12ab85873d842f2f8d580b567a(0)
- 38501a9e978335c63ed52555f8b6d2d8e4f138f8(0)

APPENDIX C – PACKAGES AND CLASSES THAT CONTAIN SOURCES OF PERFORMANCE DEVIATION

This appendix is complementary to the study presented in **Chapter 6** entitled *"Evolutionary Analysis"*. It presents the packages and classes that contain sources of performance deviation which were found by the framework during the automated analysis. Table 20, Table 21, Table 22, Table 23, Table 24 and Table 25 show how the methods flagged by the framework as potential responsible for performance deviations are spread over the classes and packages of each target system, considering Netty, Wicket and Jetty, respectively.

Table 20. Netty: number of methods responsible for performance deviation by packages.

Package	NE1	NE2	NE3	NE4	NE5	NE6	Total
io.netty.channel	2	2	4	0	0	2	**10**
io.netty.buffer	0	0	2	0	5	0	**7**
io.netty.channel.socket.nio	0	0	0	0	3	0	**3**
io.netty.channel.nio	0	0	1	0	0	0	**1**
Total:	**2**	**2**	**7**	**0**	**8**	**2**	**21**

Table 21. Netty: number of methods responsible for performance deviation by classes.

Class	NE1	NE2	NE3	NE4	NE5	NE6	Total
io.netty.channel.DefaultChannelHandlerContext	1	1	1	0	0	0	**3**
io.netty.channel.ChannelOutboundBuffer	1	1	0	0	0	0	**2**
io.netty.buffer.AbstractByteBufAllocator	0	0	1	0	0	0	**1**
io.netty.buffer.PooledByteBuf	0	0	1	0	0	0	**1**
io.netty.channel.ChannelOption	0	0	1	0	0	0	**1**
io.netty.channel.DefaultChannelPipeline	0	0	0	0	0	2	**2**
io.netty.buffer.PoolArena	0	0	0	0	1	0	**1**
io.netty.buffer.PoolThreadCache	0	0	0	0	3	0	**3**
io.netty.buffer.PoolThreadCache$MemoryRegionCache	0	0	0	0	1	0	**1**
io.netty.channel.socket.nio.NioDatagramChannel	0	0	0	0	1	0	**1**
io.netty.channel.socket.nio.NioServerSocketChannel	0	0	0	0	1	0	**1**
io.netty.channel.socket.nio.NioSocketChannel	0	0	0	0	1	0	**1**
io.netty.channel.nio.NioEventLoop	0	0	1	0	0	0	**1**
io.netty.channel.SingleThreadEventLoop	0	0	1	0	0	0	**1**
io.netty.channel.ThreadPerChannelEventLoopGroup	0	0	1	0	0	0	**1**
Total:	**2**	**2**	**7**	**0**	**8**	**2**	**21**

Table 22. Wicket: number of methods responsible for performance deviation by packages.

Package	WE1	WE2	WE3	WE4	WE5	WE6	Total
org.apache.wicket	0	0	0	12	0	3	**15**
org.apache.wicket.request	0	0	0	5	0	1	**6**
org.apache.wicket.markup	0	0	0	5	0	0	**5**

org.apache.wicket.jmx	0	0	0	3	0	0	**3**
org.apache.wicket.session	0	0	0	3	0	0	**3**
org.apache.wicket.serialize	0	0	0	1	0	0	**1**
Total:	**0**	**0**	**0**	**29**	**0**	**4**	**33**

Table 23. Wicket: number of methods responsible for performance deviation by classes.

Class	WE1	WE2	WE3	WE4	WE5	WE6	Total
org.apache.wicket.MarkupContainer	0	0	0	8	0	2	**10**
org.apache.wicket.request.handler.render.WebPageRenderer	0	0	0	5	0	1	**6**
org.apache.wicket.jmx.Initializer	0	0	0	3	0	0	**3**
org.apache.wicket.session.DefaultPageFactory	0	0	0	3	0	0	**3**
org.apache.wicket.Component	0	0	0	2	0	0	**2**
org.apache.wicket.markup.MarkupCache	0	0	0	2	0	0	**2**
org.apache.wicket.Page	0	0	0	2	0	0	**2**
org.apache.wicket.Application	0	0	0	0	0	1	**1**
org.apache.wicket.markup.html.list.ListView	0	0	0	1	0	0	**1**
org.apache.wicket.markup.parser.XmlPullParser	0	0	0	1	0	0	**1**
org.apache.wicket.markup.repeater.AbstractRepeater	0	0	0	1	0	0	**1**
org.apache.wicket.serialize.java.JavaSerializer	0	0	0	1	0	0	**1**
Total:	**0**	**0**	**0**	**29**	**0**	**4**	**33**

Table 24. Jetty: number of methods responsible for performance deviation by packages.

Package	JE1	JE2	JE3	JE4	JE5	JE6	Total
org.eclipse.jetty.http	0	0	0	0	1	0	**1**
org.eclipse.jetty.server	0	0	0	0	1	0	**1**
org.eclipse.jetty.util.resource	0	0	0	0	1	0	**1**
Total:	**0**	**0**	**0**	**0**	**3**	**0**	**3**

Table 25. Jetty: number of methods responsible for performance deviation by classes.

Class	JE1	JE2	JE3	JE4	JE5	JE6	Total
org.eclipse.jetty.http.MimeTypes$Type	0	0	0	0	1	0	**1**
org.eclipse.jetty.server.ServerConnector	0	0	0	0	1	0	**1**
org.eclipse.jetty.util.resource.PathResource	0	0	0	0	1	0	**1**
Total:	**0**	**0**	**0**	**0**	**3**	**0**	**3**

APPENDIX D – ONLINE SURVEY FEEDBACK

This appendix is complementary to the study presented in **Chapter 6** entitled *"Evolutionary Analysis"*. It presents the feedback we have obtained from eight developers of Netty, Wicket and Jetty through an online survey. Each question is going to be presented and followed by the answer of each developer. First, we show answers for Netty (four developers), then the answers for Wicket (three developers) and, finally, the answers for Jetty (one developer).

Multiple choice questions were mandatory, while writing questions were optional. The word *"Empty"* in the answer of a question means that the developer did not answer it. We did a separated survey for each system, but the questions are presented together in order to avoid repetitions. Here, extra comments to explain parts of the survey are between brackets.

Netty/Wicket/Jetty Framework Questionnaire

We are developing an open-source tool to help developers manage execution time (response time) variations in their software. We are currently conducting a study on the Netty/Wicket/Jetty framework and would be grateful if you could answer the following questions. On the last page, we summarize some of our results so far, and we will be happy to send you the full results of our study after we conclude the analysis.

Page 1/5

In which year did you start contributing to the Netty/Jetty/Wicket project?

Developer 1 (Netty): *2013*
Developer 2 (Netty): *2014*
Developer 3 (Netty): *2013*
Developer 4 (Netty): *2008*

Developer 5 (Wicket): *2009*
Developer 6 (Wicket): *2010*
Developer 7 (Wicket): *2008*

Developer 8 (Jetty): *2009*

Page 1/5

How familiar are you with the Netty/Wicket/Jetty project?
a) Very unfamiliar
b) Unfamiliar

c) Neutral
d) Familiar
e) Very familiar

Developer 1 (Netty): *Unfamiliar*
Developer 2 (Netty): *Very familiar*
Developer 3 (Netty): *Very familiar*
Developer 4 (Netty): *Very familiar*

Developer 5 (Wicket): *Very familiar*
Developer 6 (Wicket): *Familiar*
Developer 7 (Wicket): *Very familiar*

Developer 8 (Jetty): *Very familiar*

Page 2/5

In your opinion, how important is execution time (response time) for the Netty/Wicket/Jetty framework?
a) Not important
b) Not very important
c) Neutral
d) Somewhat important
e) Very important

Developer 1 (Netty): *Very important*
Developer 2 (Netty): *Very important*
Developer 3 (Netty): *Somewhat important*
Developer 4 (Netty): *Very important*

Developer 5 (Wicket): *Somewhat important*
Developer 6 (Wicket): *Very important*
Developer 7 (Wicket): *Neutral*

Developer 8 (Jetty): *Very important*

Page 2/5

Assume you have just released a new version of Netty/Wicket/Jetty. What do you do to ensure that the execution time of Netty's/Wicket's/Jetty's main functionalities is acceptable compared to other releases?

Developer 1 (Netty):
Empty

Developer 2 (Netty):
Same thing for any java project: JMH for microbenchmarks Higher level generalized benchmarks Down stream project testing and profiling (eg yourkit, jfr)

Developer 3 (Netty):

Nothing specific. I have a lot of unit tests that exercise Netty heavily, so if I saw the run time for builds going up dramatically, I might dig into it and see what happened.

Developer 4 (Netty):
Automatic benchmark and compare.

Developer 5 (Wicket):
Usually I use the SNAPSHOT version of Wicket for developing my current project. If I notice regression in the performance I fix it before the next release.

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
Run JMeterbased tests.

Developer 8 (Jetty):
We have several sets of benchmarks that we run, not on every minor release, but on major releases and whenever we have made a significant change. We also utilize micro benchmarks on units of critical code when we wish to evaluation alternate approaches.

Page 2/5

Assume you have changed some important methods used by several communication scenarios in Netty/Wicket/Jetty. Imagine that these changes have caused an increase in execution time that has affected many other methods. How would you go about finding the sources of the execution time variation (methods, commits, or development issues)? Would you use a specific tool?

Developer 1 (Netty):
Empty.

Developer 2 (Netty):
See previous.

Developer 3 (Netty):
If there's something obvious that could be affecting the time, I'd test the theory that that was the cause by replacing pieces of it with the old implementation and try to nail down the culprit. I'd probably start by writing a test that exercises that part of the code for an hour or two and compare GC logs old code vs. new, to get an idea if the memory profile has changed. And take periodic thread dumps and compare the stacks I see most commonly that's poorman's profiling you can do on a production system without the act of measuring it altering the behavior too much.

Developer 4 (Netty):
Depends, mainly YourKit.

Developer 5 (Wicket):
I use YourKit and JProfiler for measuring the performance of my main application.

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
JMeter reports + Jenkins.

Developer 8 (Jetty):
The jetty code base is such that straight forward analysis of execution time is rarely enough to diagnose performance issues. Problems are most likely to be cause by mechanical sympathetic type issues, contention of locks or just latency from too frequent dispatch. The approach taken will vary. We have tried a binary chop of commits to find a performance issue, but it is rare that we have so little idea as to what change caused the problem. Our problem is more often finding out why a change has caused a performance issue rather than which change. We will try micro benchmarks on some changes, but that rarely helps much with the kind of issues we encounter. So we typically have to resort to visual code analysis, often in pair programming mode, to walk through possible scenarios, devise tests to prove/deny theories and use logging/debuggers/custom code to check how our theories apply in benchmark runs.

Page 3/5

Netty question:
Are you aware of any execution time problem in any of the Netty releases between 4.0.0.Final and 4.0.21.Final?

Wicket question:
Are you aware of any execution time variation (increase or decrease) in any of these Wicket releases: 6.15.0, 6.16.0, 6.17.0, 6.18.0, 7.0.0M1, 7.0.0M2, 7.0.0M4?

Jetty question:
Are you aware of any execution time variation (increase or decrease) in any of these Jetty releases: 9.2.6, 9.2.7, 9.2.8, 9.2.9, 9.2.10, 9.3.0.M0, 9.3.0.M1?

Options:
a) No, I am not aware of any execution time problem.
b) Yes, I know that some of these releases had execution time variation.

Developer 1 (Netty): *No*
Developer 2 (Netty): *Yes*
Developer 3 (Netty): *No*
Developer 4 (Netty): *No*

Developer 5 (Wicket): *Yes*
Developer 6 (Wicket): *No*
Developer 7 (Wicket): *No*

Developer 8 (Jetty): *Yes*

Page 3/5

If you answered yes to the previous question, which releases and functionalities were affected? Please provide any additional information that you find relevant.

Developer 1 (Netty):
Empty.

Developer 2 (Netty):
A bunch - just check the release notes. Most of my personal troubles are not directly wasted CPU though. I personally had the most trouble with a change that introduced a bug that doubled memory usage. Somewhere in the 4.0.teens I think.

Developer 3 (Netty):
Empty.

Developer 4 (Netty):
Not using 4.0 branch.

Developer 5 (Wicket):
This is a top secret :p.

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
Empty.

Developer 8 (Jetty):
We made a major change to our threading policy in 9.3.0M1. Rather than a producer/consumer model where one thread (eg a selector or parser) might dispatch tasks to a thread pool, we instead used a more mechanical sympathetic approach that we have nicknamed eat-what-you-kill: ie don't produce (kill) a task that you don't intent to consume (eat) yourself as you have a hot cache and can avoid dispatch latency. While this approach has given us a few correctness issues (fixed in next release), it has demonstrated much better throughput AND improved request latency.

Page 3/5

If you answered no, are you aware of other execution time variations in any of the <u>Netty/Wicket/Jetty</u> releases? If yes, which functionalities were affected? Are you able to indicate the causes (methods, commits, or development issues) for the execution time increase or decrease?

Developer 1 (Netty):
Empty.

Developer 2 (Netty):
It affected buffer pooling and allocation. Yeah, I found the cause and offered a fix.

Developer 3 (Netty):
None that have affected me enough to notice.

Developer 4 (Netty):
Yes in 3.9 due to increasing (statically) assigned threads at startup instead of dynamically assigned threads. Static is faster but could lead to some issues when badly configured, while dynamic was ok since it never reaches the high number of threads. Fixing configuration in final program fix the issue.

Developer 5 (Wicket):
Empty.

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
Empty.

Developer 8 (Jetty):
Empty.

Page 4/5

Netty question:
We noticed that the execution time increased for several test cases between releases 4.0.10 and 4.0.15. After that, the execution time decreased again between releases 4.0.17 and 4.0.18. This was the case for scenarios tested by classes such as WriteBeforeRegisteredTest, SocketEchoTest, and SocketObjectEchoTest. Are you aware of these execution time issues?

Wicket question:
We noticed that the execution time increased for several test cases between releases 6.18.0 and 7.0.0M1. After that, the execution time decreased again between releases 7.0.0M2 and 7.0.0M4. This was the case for scenarios tested by classes such as HangManTest, NiceUrlTest, and LinkomaticTest. Are you aware of these execution time variation?

Jetty question:
We noticed a small increase of the execution time for some test cases between releases 9.2.10 and 9.3.0.M0. This was the case for scenarios tested by classes from servlet project such as AsyncServletLongPollTest, AsyncIOServletTest, and DispatcherForwardTest. Are you aware of these execution time variations?

Options:
a) I'm aware of these execution time variations.
b) I'm not aware of these execution time variations.
c) I'm not aware of these execution time variations, and I disagree with these results.

Developer 1 (Netty):
I'm not aware of these execution time variations.

Developer 2 (Netty):
I'm not aware of these execution time variations.

Developer 3 (Netty):
I'm not aware of these execution time variations.

Developer 4 (Netty):
I'm not aware of these execution time variations.

Developer 5 (Wicket):
I'm not aware of these execution time variations, and I disagree with these results.

Developer 6 (Wicket):
I'm not aware of these execution time variations.

Developer 7 (Wicket):
I'm not aware of these execution time variations.

Developer 8 (Jetty):
I'm not aware of these execution time variations.

Page 4/5

Considering the examples of the previous question, what methods, commits, or development issues caused the increase in execution time?

Developer 1 (Netty):
Empty.

Developer 2 (Netty):
That sounds vaguely familiar, but I'm not really involved at that level.

Developer 3 (Netty):
I'd need to look at the code in these tests to give a sane answer. As they say, "put micro trust in microbenchmarks". Most unit tests amount to microbenchmarks. If you show me that the number of requests per second a server under constant heavy load went down, or the response time went up, with a real workload, over a period of hours or days, that's one thing. It's a mistake to fixate on individual tests, especially small ones, because they magnify small differences that may have no practical impact, or may even have a positive impact, in real world use. In short, the premise of this question that the execution time of certain tests says something meaningful about application behavior is naive and mistaken.

Developer 4 (Netty):
Empty.

Developer 5 (Wicket):
No idea.

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
Empty.

Developer 8 (Jetty):
These tests involve some waiting and contrived situations to attempt to create particular sequences of asynchronous events. The tests themselves are extremely short and often only single iterations. There is no warmup time for the JIT or JVM. Performance results from these tests are pretty much meaningless They are correctness tests and any variation in execution time is more likely to be a slight change in the test rather than any indication of jetty's performance.

Page 5/5

Common comments:
We have implemented a tool that automatically compares the execution time of scenarios (such as test cases) between different releases of a software system. Given two or more releases of a software system and Java annotations for the scenarios (methods) whose runtime is of interest, our tool provides a comparison of the execution time of these scenarios and it automatically identifies commits and development issues associated with such variation.

Netty specific comments:
When we applied our tool to different versions of Netty (4.0.0.Final, 4.0.6.Final, 4.0.10 Final, 4.0.15.Final, 4.0.17.Final, 4.0.18.Final, and 4.0.21.Final), our tool automatically identified 21 methods with increased execution time that affected 13 different test cases. Our tool also automatically linked these changes in execution time to 9 development issues. For example, let's consider the scenario "Send Simple Echo Message with Socket" based on the test class SocketObjectEchoTest from release 4.0.10.Final to 4.0.15.Final. Comparing the difference in execution time, our tool indicated that this scenario executed three additional methods in release 4.0.15.Final that were not executed in release 4.0.10.Final, which are: DefaultChannelHandlerContext.safeExecute(), NioEventLoop.openSelector(), and ChannelOption.valueOf(). In release 4.0.15.Final, these methods had an average execution time of 0.031ms, 3.248ms and 0.059ms, respectively (based on 10 runs). In each run, these methods were executed 6162, 5, and 28 times, respectively. Furthermore, the method DefaultChannelHandlerContext.write() increased its execution time between these two releases by 0.0375ms, and it was executed 6144 times in each run.

Below, the table [this table was a partial sample of Table 9 for the given example] shows the issues and commits responsible for changing these methods as identified by our tool. These were the four main methods that increased the execution time of "Send Simple Echo Message with Socket" based on the test class SocketObjectEchoTest from release 4.0.10.Final to 4.0.15.Final. Other methods were also executed only in release 4.0.15.Final, but in the context of this scenario, they had an execution time average close to zero, thus we are not reporting them here. Other methods that increased the execution time for this scenario were inside the test suite package, which we are also not reporting.

Wicket specific comments:
When we applied our tool to different releases of Wicket (6.15.0, 6.16.0, 6.17.0, 6.18.0, 7.0.0-M1, 7.0.0-M2, and 7.0.0-M4), our tool automatically identified 29 methods with increased execution time that affected 7 different test cases and 4 methods with decreased execution time that affected 12 test cases. Our tool also automatically linked these changes in execution time to 15 development issues and other 17 commits. Some of these results are summarized in the table below [this table was the same of Table 10] for the evolutions we found performance

variation (WE4 – From 6.18.0 to 7.0.0-M1 and WE6 – From 7.0.0-M2 to 7.0.0-M4). All indicated methods of WE4 had an increase of the execution time, while all of WE6 had a decrease of execution time. We did not detect significant variation for the other releases. To get these results, each test case was repeated 10 times in random order, each one in a different JVM and we only considered average impacts greater than 10ms.

Jetty specific comments:
When we applied our tool to the Jetty-Servlet project in different releases of Jetty (9.2.6, 9.2.7, 9.2.8, 9.2.9, 9.2.10, 9.3.0.M0, 9.3.0.M1), our tool automatically identified 3 methods with small increases of their execution time that affected scenarios tested by 6 different test classes. Our tool also automatically linked these changes to 1 development issues and other 7 commits. Some of these results are summarized in the table below [this table was the same of Table 11] for the evolutions we found variation (JE5 – From 9.2.10 to 9.3.0.M0). We did not detect significant variations for the other releases. To get these results, each test case was repeated 30 times in random order, each one in a different JVM and we configured our tool to consider average impacts greater than 1ms.

Common comments:
The table [the specific table indicated as example in each survey] is sorted by the average impact measured in milliseconds. The average impact is calculated by multiplying the delta average execution time of the method (the difference between the average execution time of the method in both releases) by the number of times that it was executed for a specific scenario (test case) and divided by the number of repetitions of the scenario. For those methods that impacted more than one scenario, the average impact is their arithmetic mean of the average impact in each impacted scenario.

Question:
Do you think such a tool for automatically analyzing releases and indicating execution time variations as well as their potential sources (development issues, commits, and methods) would be useful?

Options:
a) Not useful at all
b) Not very useful
c) Neutral
d) Somewhat useful
e) Very useful

Developer 1 (Netty): *Very useful.*
Developer 2 (Netty): *Neutral.*
Developer 3 (Netty): *Neutral.*
Developer 4 (Netty): *Very useful.*

Developer 5 (Wicket): *Not very useful.*
Developer 6 (Wicket): *Somewhat useful.*
Developer 7 (Wicket): *Somewhat useful.*

Developer 8 (Jetty): *Somewhat useful.*

Considering the tool summarized above, would you suggest to use a different tool or strategy instead of our approach? Are they ways to complement our tool?

Developer 1 (Netty):
Empty.

Developer 2 (Netty):
If it is very low effort, then it might be worth running, but I wouldn't trust it for much more than leads. To be reliable for benchmarking, tests pretty much have to be designed using JMH (and even then they are frequently erroneous). It might do a decent job of catching "obviously bad things" that otherwise pass unit tests, but have to be careful with the noise level. That said, everyone enjoys faster test (and therefore build) speeds. So even if a change in test performance ends up not being representative of real usages, it isn't entirely useless (assuming it can be fixed without significant effort).

Developer 3 (Netty):
*To give some background, I do a fair amount of performance consulting work for a living. This sort of tool *might* be useful as a way of detecting certain kinds of performance regressions, BUT the results will be very fragile.*

Developer 4 (Netty):
Empty.

Developer 5 (Wicket):
Microbenchmarking is a tough business! Take a look at http://openjdk.java.net/projects/codetools/jmh/ and read its author's blog http://shipilev.net/. It explains a many things which would be in benefit for you.

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
Empty.

Developer 8 (Jetty):
Hard for me to comment in this case as the changes from 9.2.10 to 9.3.0.M0 were largely to include significant new functionality (HTTP/2), so we were initially not focused on performance. From M1 onwards we are looking more at performance as well. Having said that, it is interesting enough that I will be looking at MimeType$Type and checkAliasPath changes to double check we've not done anything stupid. But my previous comments still apply, the vast majority of our unit tests are not going to warm up the JIT enough to be valid comparisons of performance. Typically we have no unit tests that truly test performance because for us, such tests tend to stress out CI machines too much and we get complaints from their providers!

Page 5/5

Would you be willing to be contacted for a follow up interview?
a) Yes
b) No

Developer 1 (Netty): *No*
Developer 2 (Netty): *No*
Developer 3 (Netty): *Yes*
Developer 4 (Netty): *No*

Developer 5 (Wicket): *Yes*
Developer 6 (Wicket): *Yes*
Developer 7 (Wicket): *No*

Developer 8 (Jetty): *Yes*

Page 5/5

Do you have any further comments for us?

Developer 1 (Netty):
Empty.

Developer 2 (Netty):
I've read, analyzed, and sometimes offered suggestions on, most of the codebase. However, I'm not actually much involved with the project outside of a couple issues and being a heavy user. So I have little familiarity with the test suite, or build system. You may want to downgrade the weight of my responses accordingly, sorry. Good luck with your analysis, and let me know what your conclusions are if you have a chance.

Developer 3 (Netty):
*As part of a continuous build, this might be vaguely useful, but you'd need to: Compensate for machine load (in a cloud environment it may be your neighbor causing the load with no way to detect that fact) if not run on bare metal, you'll be measuring *something*, but you don't know what Run the test many times, to weed out classloading and hotspot compilation Ensure you're not swapping Compensate for OS level memory management Throw away any results that don't show a very significant, consistent change in execution time In short, this could be cool, but it's going to turn out to be less useful than it sounds at first blush.*

Developer 4 (Netty):
Nice tool if it is really "automatic" since currently comparison is a manual process.

Developer 5 (Wicket):
Good luck!

Developer 6 (Wicket):
Empty.

Developer 7 (Wicket):
Empty.

Developer 8 (Jetty):
Your first question did not give enough years. I started working on Jetty in 1995 so yes 20 years this December!!!!