Automating the Assessment of the Performance Quality Attribute for Evolving Software Systems: An Exploratory Study

Felipe Pinto¹ ², Uirá Kulesza¹, Leo Silva¹ ², Eduardo Guerra³
¹Federal University of Rio Grande do Norte, Natal, Brazil
²Federal Institute of Education, Science and Technology of Rio Grande do Norte, Natal, Brazil
³National Institute for Space Research, São Paulo, São José dos Campos, Brazil
felipe.pinto@ifrn.edu.br, uira@dimap.ufrn.br, leopontosilva@gmail.com, guerraem@gmail.com

Abstract
This paper describes an exploratory study for the evaluation of the performance quality attribute for releases of the same system. The main aim is to reveal performance degradations of architectural scenarios and their possible causes. Three software systems from different domains are used in our study, including a large-scale web system (SIGAA), a UML modeling tool (ArgoUML), and a client-server framework for development of network applications (Netty). The data collection of the study is accomplished using a scenario-based approach that uses dynamic analysis and code repository mining to provide an automated way to reveal degradations of scenarios on releases of software systems. The results of our study show the feasibility of the approach to determine the causes of the performance degradations of scenarios, including the degraded and changed methods of scenarios, and the issues that have affected them.

1. Introduction

The maintenance and evolution of large-scale systems has become a critical and hard task over the last years, due to the complexity to manage and adapt them to new customer requirements and technologies. When evolving software systems, developers must be aware to maintain their quality attributes according to the architectural and design decisions taken during the development and maintenance of the systems. The ability to understand and analyze the impact of new introduced changes for the degradation of quality attributes is fundamental to avoid the architectural erosion [1] of the system. Quality attributes degradation happens when a new release of a software system exhibits inferior measurements of the quality attributes than previous releases. In our work, we focus on the quality attribute of performance considering the property of execution time.

Software architecture can be seen as a result of a set of design decisions made about the system [1]. Design decisions might affect many system elements, including its structure and quality attributes. Existing architectural evaluation methods, such as SAAM, and ATAM [3] [4], propose the investigation of quality attributes through the manual analysis and review of the design decisions addressing relevant usage scenarios. A scenario is a high-level action to the system and represents the way in which the stakeholders expect the system to be used [3]. The degradation of quality attributes over the execution of the same scenarios for different releases of the system can also be seen as a way of architectural erosion [1].

There are many research works that explore the architectural analysis of quality attributes based on scenarios [4] [4] [6]. However, existing work do not focus on the identification of the code assets or development issues (change requests) responsible for the degradation of specific quality attributes. Current approaches that perform code analysis usually focus on the structural compliance of the system [7] [8] [9] [10]. Other proposals use mathematical models for predicting quality attributes [11] [12], but they do not assess the real system implementation thus preventing to determine the real impact of the system evolution on the quality attributes. In addition, there are specific studies that use benchmarks for quality attribute analysis (e.g. performance [13]), which focus on the analysis at a lower abstraction level without considering architectural scenarios. Finally, some recent research studies has addressed on determine how performance bugs are usually discovered, reported to developers and fixed by developers [14] or to create and mine repositories of performance regression-causes to identify the causes of new performance regressions [15]. In both cases, the studies do not show code assets or changes that are the source of the performance problem.

In this context, this paper describes an exploratory study for the evaluation of the performance quality attribute for releases of the same system. The main aim of our study is to reveal performance degradations of architectural scenarios and their possible causes. Three software systems from different domains are used in
our study, including a large-scale web-based system (SIGAA [16]), a UML modeling tool (ArgoUML [17]) and a client-server framework for development of network applications (Netty [18]). The data collection of the study is accomplished using an automated scenario-based architecture evaluation approach for performance considering the property of execution time. The first step of the approach consists on the monitoring of the system execution using dynamic analysis to detect performance degradation of architecturally relevant scenarios. After that, the degraded methods (and constructors) collected from the dynamic analysis for two existing releases of each analyzed software system are mined from repositories (version control and issue track systems). The repository mining contributes to identify code assets, commits and development issues responsible for the performance degradations of the scenarios.

The main contributions of this work are: (i) the proposal of an automated architectural approach for the analysis of the performance degradation of system scenarios; and (ii) the conduction of an exploratory study that evaluates the proposed approach and reveal performance degradations of evolution releases of existing software systems. Overall, our 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 evolution of each system; and (ii) 2, 31 and 29 development issues were pointed out to be the main sources of performance degradation for SIGAA, Netty and ArgoUML, respectively.

The rest of this paper is organized as follows: Section 2 presents an overview of the evaluation approach. Section 3 describes our study design. Section 4 presents the results of the study; Section 5 discusses its results, while Section 6 discusses some threats to validity; Section 7 reports some related work; and, finally, Section 8 concludes the paper.

2. Approach Overview

Our exploratory study uses a framework that implements an automated architectural evaluation approach of quality attributes for scenarios using dynamic analysis and software repository mining techniques.

The approach requires the system source code and additional metadata information in order to proceed with the evaluation process. The metadata (code annotations) are used to indicate architecturally relevant scenarios that will be monitored by the dynamic analysis. The relevant scenarios must be selected by specialists of the system or reused from previous architectural evaluation processes. The metadata identifies methods from existing classes, which represent execution entry points of the scenarios of interest. Figure 1 gives an overview of the different phases of our approach: (i) dynamic analysis, (ii) degradation analysis and (iii) repository mining. In addition, it shows the elements of input and output for each phase.

The first phase is the dynamic analysis. It requires the execution of the scenarios through a manual or automated test suite. This phase generates the dynamic analysis model at runtime, which is persisted to 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 release. Our framework defines an aspect using the AspectJ language, which instruments the executions of the scenarios, intercepting each method execution to build the call graph and calculate the quality attributes of interest during the process. The methods, which represent entry points for scenarios, are marked at the source code with an annotation (@Scenario) also defined by the framework.

Regarding the quality attributes calculated by the aspect, it is important to emphasize that our study only considers the performance quality attribute through the computation of the execution time. The methods selected as execution entry points for scenarios represent root nodes in the graph and each node represents a method execution, which can be a regular method or constructor.

![Figure 1. Approach phases and their inputs and outputs.](image-url)
methods of the system that have been degraded over the releases. The framework calculates the average execution time for each stored method execution and compares the values for two versions of the system. If the value in the newer version has increased 5% or more, it considers that happened a performance degradation in the method. The percentage for considering degradation can be modified, if it is necessary. The same strategy is used to determine if there are degradations in any of the target scenarios.

During the dynamic analysis, the execution time of some methods can be affected by the execution of other methods that are directly or indirectly called by them. In this way, the increasing of the execution time of one method will affect all its parent nodes in the call graph. In this context, the degradation analysis will consider all these methods as performance degraded, but, actually, just one method will be blamed for the degradation. Next phase addresses this situation.

The last phase is the repository mining. It mines data from the version control and issue track systems to find out which specific code changes and development issues have affected the degraded methods identified in the previous phase. Basically, our framework retrieves commits from the version control system for each class that contains methods detected as degraded. If the commit changed lines inside the degraded method, the framework searches inside the commit log for issue numbers, which are then used to search more accurate information about the development issue in the issue track system.

Our mining process consists of the analysis of the results of several queries to the version control and issue track repositories. During this process, the framework uses regular expression to match and discover issue numbers inside commit logs. Usually, each system can use a particular notation to highlight the issue numbers inside the logs. The framework also implements a parser to statically analyze Java classes and determine their methods boundaries. It discovers the limits of each method declaration inside classes that contain degraded methods, in other words, the numbers of the first and last lines, what is useful to check if the lines that the commits affected were inside or not a degraded method in order to determine if the method changed or not during the evolution.

The methods that have been identified as degraded on the previous phases, but have not changed during the evolution, are ignored because they do not represent real degradation sources. They are only impacted by other methods because of the call method hierarchy. Since our approach is guided by scenarios, the framework only considers degraded and changed methods that impact at least one of the target scenarios.

If the method does not affect any degraded scenario, it is not considered.

The final output contains: (i) degraded and changed methods that are responsible for the degradation of scenarios; and (ii) associated code changes (commits) and development issues to each degraded method of the system. That information allows the developers analyzing the development issues in order to understand the modifications and the reasons why they have introduced any performance degradation.

The framework developed to support our approach was implemented in the Java programming language. It uses AspectJ to intercept and instrument the system execution, and Java annotations as source of metadata. Thus, it defines a @Scenario annotation to mark methods that are representative entry points for scenarios. Currently, it supports the quality attributes of performance and reliability, measuring them respectively by the properties of execution time and failure rate. The framework can automatically execute the three phases illustrated in Figure 1. As said before, the study presented here focuses only on the quality attribute of performance by using the property of execution time.

3. Study Design

This section describes the design of our exploratory study. Next subsections present the goals and research questions (Section 3.1), the target systems (Section 3.2) and the study methodology (Section 3.3).

3.1. Goals and Research Questions

The main goal of our study was to investigate the performance degradation of architecturally relevant scenarios, considering the properties of execution time as said before. 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 our study were:

(RQ1) How many and which scenarios were degraded considering the performance quality attribute? We use our automated 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.

(RQ2) Can the mining process improve the discovering of degraded methods resulted from the dynamic analysis? The dynamic analysis stage of our approach detects many class 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.

(RQ3) What are the modules of the system responsible for the performance degradation? It is also interesting to explore which specific system modules were identified to have performance degradation considering the analyzed scenarios.

(RQ4) Which kinds of issues are associated to performance degradation? Last, but not least, our study also investigates the kinds of issues that are responsible for the degradation in the studied systems.

3.2. Target Systems

The software systems evaluated in our study were an enterprise large-scale web-based system (SIGAA) and two open source systems (ArgoUML and Netty). We have chosen different systems in order to analyze and evaluate our approach 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 release, and the time frame between them. An interesting observation is that a longer time frame does 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>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Information System</td>
<td>673,610 701,257</td>
<td>160,066 156,788</td>
<td>69,281 82,466</td>
<td>August 2013  December 2013</td>
<td>Subversion  GitHub</td>
</tr>
</tbody>
</table>

The web system for academic management was developed by using Java mainstream technologies. Different customizations of this are currently used by more than 20 universities in our country. 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.

3.3. Mining and Assessment Procedures

Two computers were used to run the first phase of the approach: (i) an Intel Core i7 processor with 16GB of RAM running JBoss server application 4 for SIGAA; and (ii) an AMD Phenom II processor with 8GB of RAM memory for ArgoUML and Netty. Both of them running the Windows 7 operating system and Java version 7.

The first step in our 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 our dynamic analysis 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, we have chosen and annotated six scenarios from the library management module. This module has been selected because we have support from its original developers. Thus, the scenarios and entry points in the source code were also chosen together with members of the development team. ArgoUML and Netty use JUnit3 and JUnit4 tests, respectively, so we decided to exercise their scenarios by running their test suite.

After that, the automated test suite of each release was repeated ten times in the same conditions for each system in order to have an accurate average of the performance quality attribute. SIGAA was an exception because the system does not provide automated tests. To work around, we created and executed a series of manual testing requests to the web system, which exercise the chosen scenarios. These requests were then recorded and automatic repeated ten times using JMeter tool [19].

During our study, we executed the test suite 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. The comparison considered increasing of more than 5% of the execution time to determine if a scenario was degraded or not. Finally, we mined in the code repositories and issue tracker systems the commits and development issues to analyze the introduced changes responsible for scenario performance degradation.

4. Study Results

In this section, we present and discuss the results of the application of our approach (Section 2) to existing
software systems. The discussion is organized in terms of the research questions of our study (Section 3.1).

(RQ1) How many and which scenarios were degraded considering the performance quality attribute? 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 significant performance degradation.

Table 2. Total of analyzed and degraded scenarios.

<table>
<thead>
<tr>
<th>Number of Scenarios</th>
<th>SIGAA</th>
<th>Netty</th>
<th>ArgoUML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyzed</td>
<td>6</td>
<td>21</td>
<td>63</td>
</tr>
<tr>
<td>Degraded</td>
<td>6 (100%)</td>
<td>5 (24%)</td>
<td>12 (19%)</td>
</tr>
</tbody>
</table>

The individual scenarios and how they have degraded during the evolution for SIGAA, ArgoUML and Netty are shown in the graphics illustrated by Figure 2, Figure 3, and Figure 4, respectively. Our 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 2 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 3, we can see that the ArgoUML project has 4 scenarios with performance degradation superior to 74%. For Figure 4, 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. While answering the next research question we indicate code assets and modifications responsible for some of the detected scenario degradations.

(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. This 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.

<table>
<thead>
<tr>
<th>Total of Degraded Scenarios</th>
<th>SIGAA</th>
<th>Netty</th>
<th>ArgoUML</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>Degraded Methods Found Before Mining</td>
<td>229 (100%)</td>
<td>349 (100%)</td>
<td>1,181 (100%)</td>
</tr>
<tr>
<td>Degraded Methods Found After Mining</td>
<td>9 (4%)</td>
<td>100 (29%)</td>
<td>230 (19%)</td>
</tr>
<tr>
<td>Degraded Methods of Degraded Scenarios</td>
<td>9 (4%)</td>
<td>26 (7%)</td>
<td>153 (12%)</td>
</tr>
</tbody>
</table>
The mining phase helps to decrease the number of performance-degraded methods indicated by the dynamic analysis. For example, one of the 229 methods found in the web system before mining was QueryLoanDAO.findPersonInfoNoLink(). This method was found by the degradation analysis because it has increased more than 5% its execution time. However, it was ignored from the 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. Our framework has decreased the amount of methods to nine for SIGAA after mining only the changed methods from the degraded set of methods.

As result, we had a reduction of 96% (from 229 to 9) in the number of the degraded methods for SIGAA. Netty had a reduction of 71% (from 349 to 100) after mining and 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). We conclude 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.

**(RQ3) What are the modules of the system responsible for the performance degradation?** Our study found 9, 26 and 153 methods (Table 3) 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. Due to space constraints, we do not list all the degraded methods in the paper. Additional details about the study can be found in [20].

Table 4 shows all packages that contain degraded and changed methods. It 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 variation of the average execution time of all 43 methods was 1680.24 milliseconds. The affected packages represent points with execution time degradation.

Most of the blamed methods of Netty belong to the io.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 used to handle channels and abstract the communication. The API is used in many situations and we realized that small variations of performance in methods of classes from this package might cause a considerable impact because this API is used several times during a communication. The sum of the average execution time of the degraded and changed methods of the io.netty.channel package was 4.28ms. It means that this package has caused more impact than the io.netty.buffer package with 1.58ms. Our 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 just contains five degraded methods, while 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 XmReferenceResolverImpl.toURL(), UserDefinedProfile.getAllCritiquesInModel() and UmlFilePersisters.doLoad(), respectively, from the model, profile and persistence packages. ArgoUML had several 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

<table>
<thead>
<tr>
<th>System</th>
<th>Package Prefix</th>
<th>Total of Methods</th>
<th>Sum of Execution Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGAA</td>
<td>webservice.core.dao</td>
<td>2</td>
<td>12.87</td>
</tr>
<tr>
<td></td>
<td>webservice.core.security</td>
<td>1</td>
<td>5.31</td>
</tr>
<tr>
<td></td>
<td>webservice.core.util</td>
<td>2</td>
<td>1.03</td>
</tr>
<tr>
<td></td>
<td>webservice.sig.library</td>
<td>3</td>
<td>398.29</td>
</tr>
<tr>
<td></td>
<td>webservice.sig.domain</td>
<td>1</td>
<td>0.09</td>
</tr>
<tr>
<td>Netty</td>
<td>io.netty.buffer</td>
<td>3</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>io.netty.channel</td>
<td>23</td>
<td>4.28</td>
</tr>
<tr>
<td>Argo UML</td>
<td>org.argouml.application</td>
<td>2</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>org.argouml.18n</td>
<td>1</td>
<td>0.11</td>
</tr>
<tr>
<td></td>
<td>org.argouml.kernel</td>
<td>5</td>
<td>26.44</td>
</tr>
<tr>
<td></td>
<td>org.argouml.model</td>
<td>43</td>
<td>1680.24</td>
</tr>
<tr>
<td></td>
<td>org.argouml.notation</td>
<td>27</td>
<td>19.38</td>
</tr>
<tr>
<td></td>
<td>org.argouml.persistence</td>
<td>15</td>
<td>1753.69</td>
</tr>
<tr>
<td></td>
<td>org.argouml.profile</td>
<td>23</td>
<td>1781.14</td>
</tr>
<tr>
<td></td>
<td>org.argouml.uml</td>
<td>36</td>
<td>19.62</td>
</tr>
<tr>
<td></td>
<td>org.argouml.util</td>
<td>1</td>
<td>0.01</td>
</tr>
</tbody>
</table>
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 systems that use the common core module. The strategy has affected the library module and all analyzed scenarios. The results were presented and discussed with developers from SIGAA in order to validate them.

(RQ4) Which kinds of issues are associated to performance degradation? Mining development issues related to the changes applied to the degraded methods is useful because it can help 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, we have performed a manual inspection considering the issues mined by the automated analysis.

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 the attempt to optimize the scenarios, the performance decreased because of code changes related to the second issue. It was an enhancement issue. The GenericDAOImpl.getSession() method introduced the new method SessionLogger.registerCaller(), which caused several performance degradation for all the investigated scenarios. The registerCaller() method is responsible for implementing an auditing service for 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 trace of the invoker.

The analysis of the Netty framework found 13 bug fixing, 9 enhancements, 1 new feature, and 8 not labeled issues. The last ones are issues that were not associated to a specific type. Again, we have accomplished some manual inspection of these issues to try to understand important modifications. A not labeled 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 issues, 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 channel is canceled. They have contributed to degrade some scenarios, including sending messages with sockets and writing data before connection.

Finally, we also found a new feature issue that causes performance degradation 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 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 enhancements and 25 bug fixing issues. Two enhancement 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 user profile that depends on another user 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 on the other hand has been initialized before. The solution has performed several modifications, including 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 our study. Although, we cannot generalize the results for other systems and releases than the ones we have investigated. The low number of issues for SIGAA is justified by the reduced number of changes in the analyzed releases.

<table>
<thead>
<tr>
<th>Issue Type</th>
<th>SIGAA</th>
<th>Netty</th>
<th>ArgoUML</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enhancement</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>Bug Fixing</td>
<td>1</td>
<td>13</td>
<td>25</td>
<td>39</td>
</tr>
<tr>
<td>New Feature</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Not Labeled</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td><strong>TOTAL:</strong></td>
<td>2</td>
<td>31</td>
<td>29</td>
<td>62</td>
</tr>
</tbody>
</table>

5. Discussions

**Architecture Evaluation Effectiveness.** Our exploratory study has successfully identified sources of performance degradation for the evolution of three real systems by using an approach based on scenarios. We consider our analysis as an architectural evaluation because it is guided by architectural relevant scenarios. As illustrated in our study, the approach can indicate the sources of degradation in different granularity, for example, methods, classes and packages. In general, we can map classes or packages to components and identify the high-level assets of the system that contain quality attributes degradations. The ability to find out the source of the degradations and its support for such high-level concepts distinguishes our approach from other ones that just use benchmarks to test isolated source code elements.

**Architectural Compliance Issues.** For the exploratory study presented in this paper, we assume that the collected values of the performance quality attribute for the initial releases address the need of the software architecture. However, even in the cases when it is not true, it should not avoid the evolution analysis because it should successfully determine if the scenarios were degraded or not. Whether the initial release is already degraded compared to expected values for the software architecture, we will just discover that the next release is even more degraded, but there is also the opportunity to analyze previous versions to understand and recover the code changes responsible for the found degradations. It is also possible to check out if the values of the supported quality attributes belong 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.

**Precision Detection of Source Degradations.** The integration of dynamic analysis and repository mining has revealed to be an interesting approach to improve the detection precision of the degradation sources for architectural quality attributes. Dynamic analysis identified several performance-degraded methods for all the investigated systems. The combination of those results with repository mining techniques allowed us to identify more accurately the methods blamed for degradations, which represents a reduced number of those ones captured by the dynamic analysis. The strategy is to verify which degraded methods were changed during the evolution. Those ones that have not changed are not source of degradation and they were just impacted by others. However, the set of degraded and changed methods do not have an architectural meaning if the analysis does not consider high-level concepts such as scenarios. We are interested in methods blamed for degradation of scenarios. In order to address that, our approach reduces even more the set of degraded and modified methods to consider those ones belonging to degraded scenarios.

**Approach Execution Challenges.** Our study also revealed some execution challenges. The first one is the manual annotation of the systems, which requires appropriate knowledge. Another requirement is the storage and versioning of all code artifacts related to the system versions, as well as the traceability data between development tasks and code revisions. That could not be possible for legacy systems that are running for many years. Finally, it is also important the availability of automated tests to execute the scenarios. The absence of tests requires a substantial effort to create them. That was the main reason why we selected only six scenarios from SIGAA.

**Execution time property.** Our approach works with the execution time property. Other possible properties for performance are memory consumption, disk activity and CPU usage, for example. New performance properties are being analyzed to be considered for next studies. In order to address it, the aspect that instruments the system execution and calculates the execution time could be adapted to estimate new properties. It is also interesting because for some systems, memory might be more important than execution time, for example. In addition, new lines of code to adding new features or correcting bugs might potentially increase the execution time. It will be detected and the development team needs to decide whether the time increment represents or not a performance degradation considering the changes.

6. Threats to Validity

This section presents and discusses the threats to validity of our study.
6.1. Threats to Construct Validity

Architectural knowledge. The identification of scenarios executed by the automated tests and the manual selection of the entry methods of relevant scenarios require an appropriate knowledge of the software system architecture. In order to deal with that, we performed the following procedure: (i) for SIGAA, the developers helped us to identify these elements and validate the results; and (ii) for ArgoUML and Netty, we studied the software structure and associated documentation to perform these tasks without help.

Discovering issues. We search for issues number in logs of commit. Usually, developers indicate the issue number using a pre-defined notation and it can be match 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 verify manually the commits in the analyzed period to certificate the issue numbers were correctly indicated.

6.2. Threats to Internal Validity

Rename problem. One internal threat is regarding the method rename. The framework considers it as a new element that potentially will increase the scenario execution time, considering performance. We could execute a semantic comparison algorithm to verify whether the old and new method are the same element with a different name or not, even if one of them has been changed. Our manual inspection to interpret the results showed us that in this study, they are consistent and this problem was not critical.

6.3. Threats to External Validity

Issue generalization. Despite of the amount of information collected for both dynamic analysis and repository mining, the results of our study cannot be generalized for other systems or domains in terms of which kinds of development issues usually contribute to performance degradation. We intend to conduct new replications of our study with more releases of the same systems to analyze if we can found predominant kind of issues that contribute to system performance degradation of specific systems. For example, our study cannot conclude that bug fixing usually contribute more to performance degradation than new feature or enhancement issues (Table 5). In order to test this hypothesis new studies need to be conducted.

7. Related Work

There are a few research works focusing on the conduction of studies to detect and analyze the performance degradation of software systems, but none of them has focused on the automated identification of the discovering of the causes of such degradations. Our study addresses this issue by using an approach that integrates dynamic analysis and repository mining.

Malik et al. [13] propose strategies to help performance analysis to more effectively compare results of load tests to find performance deviation in large-scale systems. Their approach provide a reduced and manageable set of important performance counters to assist in root-cause analysis of the detected deviations. These performance counters are measures, including CPU utilization, memory and others. The latest results are compared to the previous one to determine if there was any performance deviation. The case study is based on load tests from an industrial system and an open source system by using the company data and benchmark application, respectively. Their study focused in a lower level of abstraction than ours did. They present a reduced set of performance counters for the system that helps analysts to determine degradation causes, but they do not indicate sources in the level of scenarios, commits or development issues.

Koziolek et al. [21] present a methodology to predict the quality attributes of performance and reliability using the response time and the failure rate. They evaluate a large-scale control process system. The evaluation goal was to quantitatively predict the quality attributes for different architectural alternatives and then to pick the alternative with the best trade-off among the quality attributes (performance and reliability). It differs from our study that focuses on the analysis of system releases to detect existing performance degradations and their causes.

Nistor et al. [14] investigate how performance bugs are discovered, reported to developers, and fixed by developers and compare the results with those for non-performance bugs. Nguyen et al. [15] propose the mining of a regression-causes repository to assist the performance team in identifying causes of new performance regressions. The regression-causes repository contains the results of performance tests and causes of past regressions. They use machine-learning techniques to determine the causes of new regressions by using the data from the regression-causes repository. However, the identified causes are limited to a pre-defined set of situations defined from bug reports that represent actions that usually cause performance regression, including adding frequently executed logic, adding blocking I/O access, and others. There are no indications of the code assets or commits that represent the source of the performance regression.

8. Conclusion

In this paper, we presented an exploratory study that aimed to perform a scenario-based architecture
evaluation of the quality attribute of performance. The study consisted on the analysis of the evolution of the performance, considering the property of execution time, for two releases of three different software systems. 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 enhancement (14 issues). Some explanation for the high number of bug fixing issues are that they usually add new code to existing scenarios to execute new validations or to cover possibilities that were not addressed before.

We are currently working on several directions to improve the results presented in this paper such as: (i) developing a new algorithm to calculate the individual influence of each degraded method in the call graph in order to improve the precision of the methods detected as sources of the scenario degradations; (ii) preparing new studies to analyze the quality attribute of performance and reliability, and to determine how accurate is our automated analysis when considering the new individual influences of each method; and (iii) investigating and incorporating other techniques in our approach that consider new performance properties besides execution time.

Acknowledgments. This work was partially supported by National Institute of Science and Technology for Software Engineering (INES) under grants CNPq 573964/2008-4 and 552645/2011-7 and SINFO/UFRN.

9. References


