An Automated Architectural Evaluation Approach Based on Metadata and Code Analysis

Felipe Pinto[1,2](✉), Uirá Kulesza[1], and Eduardo Guerra[3]

[1] Federal University of Rio Grande do Norte (UFRN), Natal, Brazil
felipe.pinto@ifrn.edu.br, uira@dimap.ufrn.br
[2] Federal Institute of Education, Science and Technology of Rio Grande do Norte (IFRN), Natal, Brazil
[3] National Institute for Space Research (INPE), São José dos Campos, Brazil
guerraem@gmail.com

Abstract. Traditional methods of scenario-based software architecture evaluation rely on manual review and advanced skills from architects and developers. They are used when the system architecture has been specified, but before its implementation has begun. When the system implementation evolves, code analysis can enable the automation of this process and the reuse of architectural information. We propose an approach that introduces metadata about use case scenarios and quality attributes in the source code of the system in order to support automated architectural evaluation through of static and dynamic code analysis and produce reports about scenarios, quality attributes, code assets, and potential tradeoff points among quality attributes. Our work also describes the implementation of a code analysis tool that provides support to the approach. In addition, the paper also presents the preliminary results of its application for the architectural analysis of an enterprise web system and an e-commerce web system.

Keywords: Architectural evaluation · Source code analysis

1 Introduction

Over the last decade, several software architecture evaluation methods based on scenarios and quality attributes have been proposed [1, 2]. These methods use scenarios in order to exercise and evaluate the architecture of software systems. A scenario represents the way in which the stakeholders expect the system to be used [1]. The methods allow the gain of architectural-level understanding and of predictive insight to achieve desired quality attributes [3].

Traditional scenario-based methods produce a report as output of the process that contains information about risk analysis regarding architecture decisions. Existing methods, such as ATAM (Architecture Tradeoff Analysis Method) [1], produce information about sensitivity and tradeoff points. Risks [1, 4] are architecturally important decisions that have not been made, for example, when the development team has not decided which scheduling algorithm will be used, or if they are going to

S. Hammoudi et al. (Eds.): ICEIS 2013, LNBIP 190, pp. 490–505, 2014.
DOI: 10.1007/978-3-319-09492-2_29

use a relational or object-oriented database. Risks can also happen when decisions have been made, but their consequences are not completely understood. One example of such case is when the architecture team has decided to include an operating system portability layer, but they are not sure which functions will part of it or how it will affect the system performance [1].

Sensitivity points [1, 4] are architectural decisions involving one or more architectural elements that are critical for achieving a particular quality attribute. In that case, the response measure is sensitive to changing the architectural decision. For example, the level of confidentiality in a virtual private network might be sensitive to the number of bits of encryption. Sensitivity points indicate to software architects and developers where they should focus attention when trying to understand the achievement of some quality attribute. They have to be careful when changing those properties of the architecture because particular values of sensitivity points might become risks. Finally, a tradeoff point [1, 4] is a sensitivity point for more than one quality attribute. For example, changing the level of encryption could have impact on both security and performance. If the time of processing a message has hard real-time requirements, the level of encryption could be a tradeoff point because it improves the security, but it requires more processing time affecting the system performance.

Existing architecture evaluation methods are applied manually and rely on manual review-based evaluation that requires advanced skills from architects and developers. They are applied when the system architecture has already been specified, but before its implementation has begun. The system implementation is one additional element that can be useful when suitably analyzed, for example, when a software evolves causing critical architectural erosion [5] implying on the need of executing the process of evaluation again because the architecture design has several differences to the architecture implemented [6]. In this case, the implementation can provide information to help the automation of the architectural evaluation, enabling information reuse from previous manual evaluations.

In this context, this paper proposes an approach that introduces metadata about use case scenarios and quality attributes in the source code of the system, which ideally should come from traditional architecture evaluation methods. The main aim is to allow automated static and dynamic code analysis in order to produce reports with information, such as: (i) the scenarios affected by particular quality attributes; (ii) the scenarios that potentially contain tradeoff points and should have more attention from the development team; (iii) the execution time of a particular scenario; and (iv) if the scenario was successfully executed or not. In our approach, when the system implementation evolves, it is possible to keep or adjust the metadata information and automatically generate a new updated evaluation report. The approach does not aim to replace traditional architecture evaluation methods, but it complements them by promoting the continuous architecture evaluation during the implementation and evolution of software systems.

The rest of this paper is organized as follows: Sect. 2 introduces the approach; Sect. 3 presents the tool developed; Sect. 4 shows two case studies where we have applied our approach; Sect. 5 discusses some related works and, finally, Sect. 6 concludes the paper.

2 Approach Overview

This section presents an overview of our approach. The main goal is to automate the architecture evaluation by adding extra information with metadata to the application source code. The approach presented here is independent of programming language or platform. Figure 1 presents an overview of the approach showing input and output artifacts of each step.

Next subsections detail the steps shown in Fig. 1 to prepare information systems to be analyzed using our approach, which are: (i) choosing scenarios from the target architecture to be evaluated; (ii) identifying starting methods from evaluation scenarios; (iii) identifying and annotating quality attributes in the source code of the target system; and (iv) executing our analysis tool that uses all the provided information to perform automated scenario-based software architecture evaluation.

The approach is not limited to particular quality attributes, although the developed tool is currently addressing only the performance and robustness quality attributes. It is important to realize that the dynamic analysis is feasible only for quality attributes that can be quantified during the system execution. On the other hand, the static analysis can be used to perform traceability of any quality attribute that has associated source code.

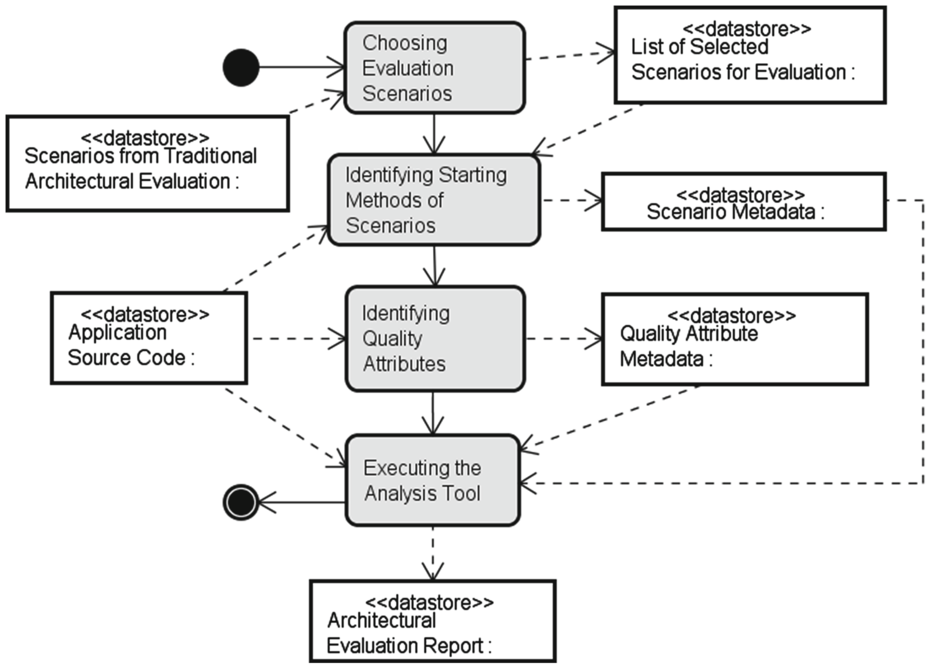
Choosing Evaluation Scenarios
<<datastore>> List of Selected Scenarios for Evaluation :
<<datastore>> Scenarios from Traditional Architectural Evaluation :
Identifying Starting Methods of Scenarios
<<datastore>> Scenario Metadata :
<<datastore>> Application Source Code :
Identifying Quality Attributes
<<datastore>> Quality Attribute Metadata :
Executing the Analysis Tool
<<datastore>> Architectural Evaluation Report :

Fig. 1. Approach overview.

2.1 Choosing Evaluation Scenarios

The first step of the approach is to choose the scenarios from the target architecture to be evaluated. In order to perform this step, we can reuse information produced by previous activities from the development process. In particular, the elicited relevant scenarios gathered during the application of traditional architecture evaluation methods, such as ATAM or others [7], might be reused during this step.

2.2 Identifying Starting Methods of Scenarios

In this step, we identify the starting points of the execution of the chosen scenarios in the application source code under evaluation. A scenario execution defines paths of execution which can be abstracted to a call graph, where each node represents a method and each edge represents possible invocations.

Our challenge in this step is to define how identifying scenarios or paths of execution in the application source code. A simple solution to associate scenarios to the source code is to identify the methods that begin the use case scenarios execution, considering them as the call graph root nodes. In order to allow the introduction of this information in the source code, our approach defines the `Scenario` metadata, which defines a name attribute to identify it uniquely.

2.3 Identifying Quality Attributes

The identification of quality attributes in the application source code is similar to the identification of scenario starting methods. We have to add the metadata to the code element that we are interested. The approach is not limited to particular quality attributes, but the tool currently defines metadata considering performance, security, reliability and robustness.

Figure 2 shows the metadata definition with their respective attributes. `Performance` metadata has two attributes: name and limit time. Name is a string that uniquely identifies it, and limit time is a long integer that specifies a maximum time expected in milliseconds. The related method must complete its execution in a shorter time compared to the limit time value. Consequently, we can monitor if particular methods related to this metadata are executing according to the expected time or if they have improved or decreased their performance in the context of an evolution between two different releases of the system.

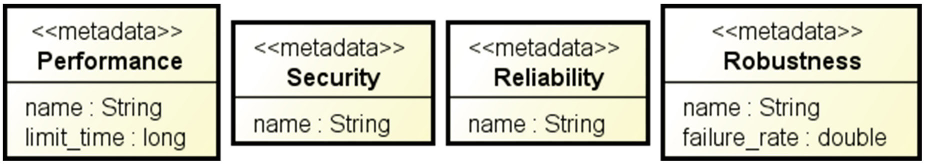

Fig. 2. Approach metadata for quality attributes.

`Robustness` metadata has a string name attribute that is unique and a double attribute that specifies the failure rate. It 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 cases. Similarly to performance, we can monitor if the method fails more times than the value specified in the attribute. In this case, a warning should alert the developer team.

`Security` and `Reliability` metadata have currently one attribute, a string that uniquely identifies each one. The main aim is to annotate source code elements where these quality attributes are critical. This is useful because it enables traceability providing the possibility of determining statically, which use case scenarios are affected by quality attributes or which ones have potential to contain tradeoff points. For example, increasing the level of encryption could improve the security of the system, but it requires more processing time. That is, if a path of scenario execution is associated to more than one quality attribute, we need to observe and monitor it carefully because it has potential to contain tradeoff points.

2.4 Executing the Analysis Tool

The last step of the approach involves the execution of static and dynamic analysis implemented in a tool. This tool parses the metadata from the source code and performs analysis in order to proceed with the automated architecture evaluation based on the chosen scenarios and specified quality attributes.

During the static analysis, the tool parses the source code and metadata and builds a call graph of the methods. The root nodes of the call graph are those methods indicated by the `Scenario` metadata as scenarios starting points. After that, the tool uses the call graph: (i) to discover the quality attributes associated to a particular scenario or which scenarios have potential to have tradeoff points; (ii) to find out which methods, classes or scenarios could be affected due to a particular quality attribute; and (iii) to perform source code traceability of scenarios and quality attributes.

The dynamic analysis also benefits from the metadata information in order to perform the architecture evaluation during the system execution. It allows monitoring the performance and robustness quality attributes. In addition, dynamic reflective calls are captured only by dynamic analysis. The analysis currently accomplished by the tool allows: (i) calculating the performance time or failure rate from a scenario or a particular method; (ii) verifying if the constraints defined by quality attribute metadata are respected during the system execution; (iii) logging several information captured during the runtime; and (iv) adding more useful information to detect and analyze tradeoff points.

3 Approach Tool Support

This section introduces a tool that we have developed to support our approach. It has been accomplished as two independent components: (i) the static analysis is implemented as an Eclipse plugin; and (ii) the dynamic analysis is made available as a JAR file. The tool implements the metadata by using Java annotations.

3.1 Tool Support for Static Analysis

The static analysis tool allows executing the architecture evaluation over Eclipse projects. It currently parses source code from Java projects. Figure 3 shows a partial class diagram of the tool.

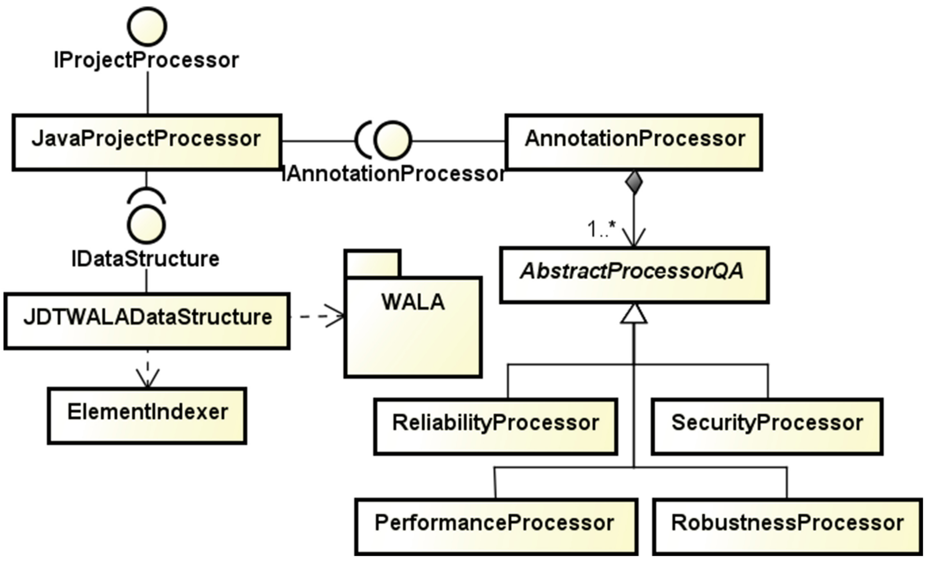

Fig. 3. UML class diagram showing tool processors.

The `JavaProjectProcessor` class calls other classes in order to build the call graph of the system under architectural evaluation. We have used the CAST (Common Abstract Syntax Tree) front-end of WALA (Watson Libraries for Analysis) static analysis framework [8] to build the call graph of the scenarios of interest. `AnnotationProcessor` class aggregates a set of different concrete strategy classes to process the different quality attribute annotations. Each one of them is responsible for the processing of a particular kind of annotation. During the annotation parsing, the `AnnotationProcessor` class also builds the list of scenarios annotated to complement the data structures built previously.

`JavaProjectProcessor` class also uses the `JDTWALADataStructure` to access and manipulate the application call graph and the indexes. The `JDTWALA-DataStructure` class uses `ElementIndexer` to build indexes of methods, classes and annotations to be used during the analysis. Actually, the annotation index is created by the `AnnotationVisitor` class that reads the source code looking for annotations.

Figure 4 summarizes the static analysis process. `JavaProjectProcessor` uses `JDTWALADataStructure` to build the call graph and the indexes. `Ele-mentIndexer` is used to build the method index and the annotation index, but it creates an `AnnotationVisitor` object that parsers the source code looking for

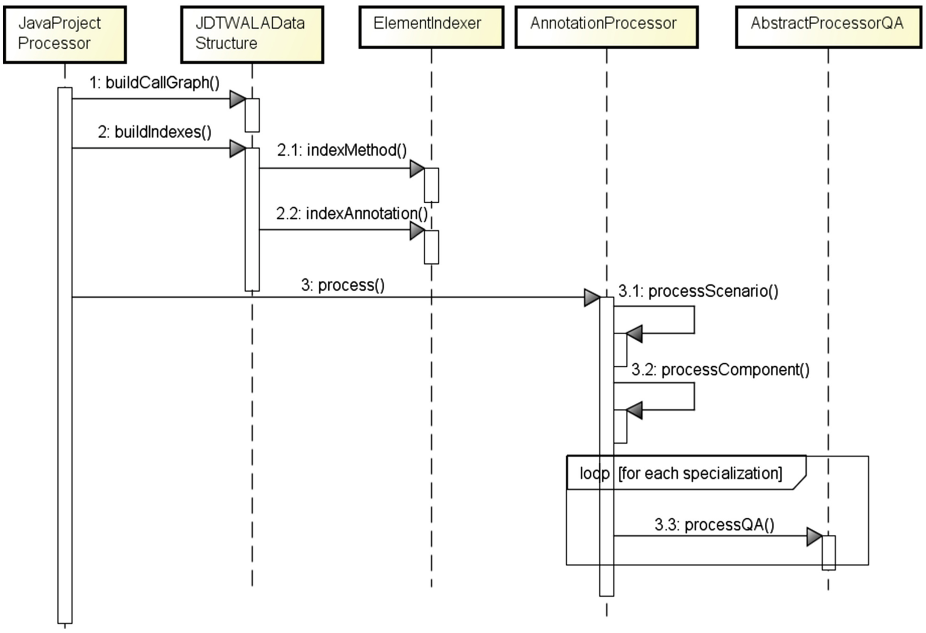

Fig. 4. UML sequence diagram to static analysis.

annotations. Then, `AnnotationProcessor` processes the scenario annotations and builds a list of scenarios. Finally, it processes each quality attribute annotation calling every `AbstractProcessorQA` subclasses.

The static analysis tool uses a model to represent the relationships among the system assets, such as classes, methods, scenarios and quality attributes. Figure 5 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 method node in the WALA call graph.

3.2 Tool Support for Dynamic Analysis

The dynamic analysis tool has been implemented using the AspectJ language. It defines aspects to monitor the execution of annotated methods. Essentially, the tool builds a dynamic call graph during application execution by using aspects to intercepting the approach annotations. The dynamic analysis should ideally be executed during the tests of the system. The quality of the analysis is directly related to the tests used to monitor the system execution because if they do not stress the system enough, the system failures will never happen, and the aspects will not be able to detect them.

In this way, the tool intercepts and monitors the scenarios execution by using the aspects, in other words, methods annotated with the scenario annotation.

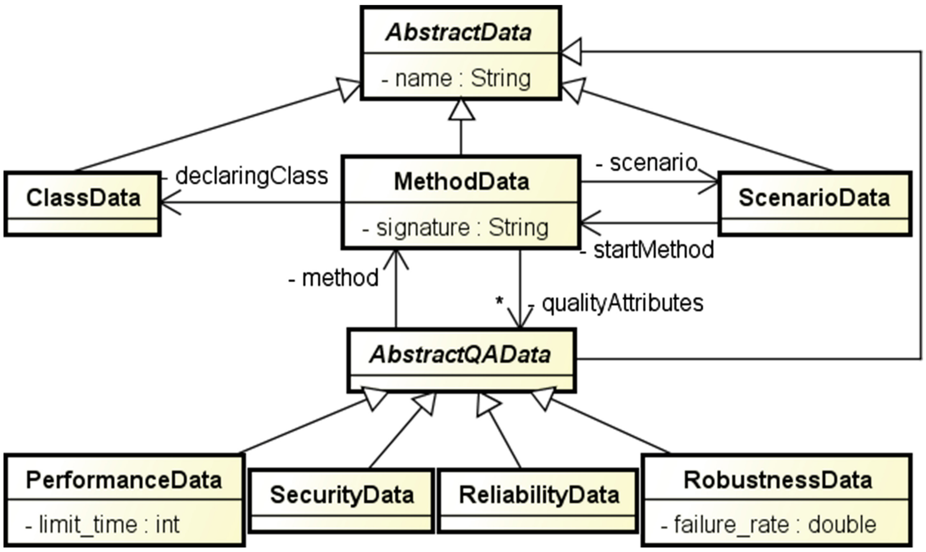

Fig. 5. UML class diagram of the static analysis model.

The execution flow is followed from the annotated method while the tool builds a dynamic call graph that accurately reflects the path executed in a particular scenario. In this graph, each node represents a method or constructor and keeps information of the quality attributes of interest, such as the execution time and if it ran successfully or not.

The current version of our tool has implemented aspects to intercept scenarios and quality attributes annotations for performance, security, robustness and reliability. In addition, the aspects can quantify values for the performance and robustness quality attributes. For performance, the tool calculates the time to execute a particular method or a full scenario depending on the analysis focus. For robustness, the tool indicates if the method or the full scenario was successfully executed or not. In this case, the tool considers that the method failed when it throws some exception. Similarly, the scenario fails when one or more of its methods fail. For security and reliability, our approach cannot provide any mechanism to quantify them. Thus, they are used to the traceability of important points in the application source code helping in the analysis and detection of tradeoff points among quality attributes.

The main aspect implemented in our approach is the `Scenario` aspect. It intercepts method invocations annotated with the scenario annotation from a particular point in order to determine the path of the scenario execution. The aspect follows the execution from the scenario starting method and builds a call graph that represents a particular execution of the scenario. Figure 6 shows the class structure for the dynamic call graph support built by scenario aspect.

A dynamic call graph (`RuntimeCallGraph`) is created for each new scenario execution and stored inside a list of execution paths (`ExecutionPaths`). Essentially, the `ExecutionPaths` class contains a list of dynamic call graphs represented

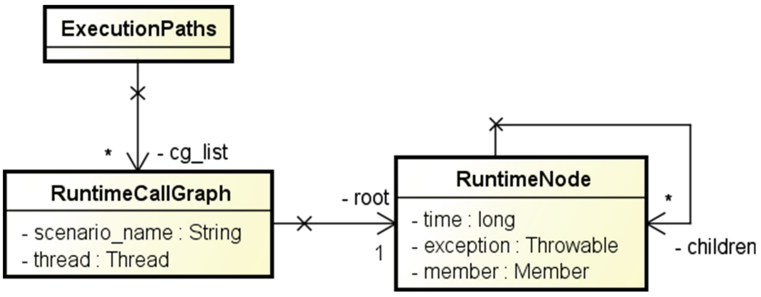

Fig. 6. Class model for dynamic call graph support.

by the class `RuntimeCallGraphGraph`, which maintains the executed scenario name, the thread that starts this scenario execution, and the root node representing the scenario starting method. Nodes (`RuntimeNode`) have `children` that represent other method invocations, and the following attributes: `time`, `exception` and `member`. The `time` and `exception` attributes indicate, respectively, the time to execute the node (method or constructor) and the exception thrown if its execution has failed (it will be null otherwise). The `member` attribute represents the node element, in other words, the method or constructor executed. The time of the full scenario execution is equal to its root node execution time.

The tool also implements default strategies to gather and store information about the execution of the relevant architecture scenarios. In addition, we have also implemented specific strategies for our case studies, which will be presented in the next section (Sect. 4).

4 Approach Evaluation

The approach has been applied to two different systems. In the first one, we have explored the static analysis in an academic enterprise large-scale web system developed by our institution. In addition, we have also applied the dynamic and static analysis to an e-commerce web system. Our main goal was to conduct an initial evaluation of the approach in order to verify its feasibility and how the developed tool behaves in practice.

4.1 Academic Enterprise Web System

We have applied the static analysis tool of our approach to enterprise web systems from SINFO/UFRN. SINFO is the Informatics Superintendence at Federal University of Rio Grande do Norte (UFRN), Brazil. It has developed several enterprise large-scale information systems [9], which perform full automation of university management activities. Due to the quality of these systems, several Brazilian government and education institutions have licensed and extended them to address their needs.

Our main goal was to verify the approach feasibility of static analysis in practice when used to a large-scale software system. In this sense, the tool should extract useful information in order to help developers answering some questions, such as: (i) what scenarios does a specific method belong to? (ii) what kinds of quality attributes can affect a specific scenario? (iii) what are the scenarios that contain potential tradeoff points among quality attributes? Next subsections describe the results of applying the approach to the academic management web system developed by SINFO/UFRN.

4.1.1 Choosing Evaluation Scenarios

In the first step, we have chosen some specific scenarios from the chosen analyzed system: (i) sending message – scenario used for sending messages (e-mails); (ii) authenticated document generation – scenario used to generate authenticated documents; (iii) user authentication – scenario used to authenticate users in the web application; and (iv) mobile user authentication – scenario used to authenticate users from a mobile device.

4.1.2 Identifying Scenarios

In this step, the starting execution methods for each chosen scenario were identified. They are, respectively, in the same order of the scenarios:

(i) `Mail.sendMessage()`: the class Mail contains some methods for sending messages;
(ii) `ProcessorDocumentGeneration.execute()`: the class `ProcessorDocumentGeneration` is a processor responsible for generating authenticated documents;
(iii) `DatabaseAuthentication.userAuthentication()`: the class `DatabaseAuthentication` implements user authentication strategy by using database;
(iv) `DatabaseAuthentication.mobileUserAuthentication()`: the class `DatabaseAuthentication` also supports mobile user authentication by using database strategy.

4.1.3 Identifying Quality Attributes

In the third step, we need to identify the quality attributes of interest in the application source code. The following methods and quality attributes were identified:

(i) `DatabaseAuthentication.getJdbcTemplate()` with `@Performance` – it was considered to be relevant for performance requirements because it is accessed by several database operations;
(ii) `Mail.enqueue()` with `@Security` – it is used by the system to enqueue messages that will be sent over the network;
(iii) `ProcessorDocumentGeneration.createRegistry()` with `@Security` – it is used to create the registry of an authenticated document to ensure its legitimacy;

(iv) `UFRNUtils.toMD5()` with `@Security` – it is used to create an MD5 hashing of strings, for example, passwords;

(v) `Database.initDataSourceJndi()` with `@Reliability` – it is used to initialize the access to the database and was considered critical for reliability because if the database initialization fails, the system is not going to work adequately.

4.1.4 Executing the Static Analysis Tool: Preliminary Results

The tool execution has extracted useful and interesting information in order to help us answering the questions highlighted in Sect. 4.1. Considering the first question – (i) what scenarios does a specific method belong to? – the tool can determine that the `getJdbcTemplate()` method, for example, belongs to the following scenarios: user authentication, mobile user authentication, and authenticated document generation. This is possible because the tool builds a static call graph of each scenario and calculates if a call to a particular method exists in some of the possible paths of execution.

Regarding the second question – (ii) what kinds of quality attributes can affect a specific scenario? – the tool verifies all the paths for a specific scenario checking which ones have any quality attribute. The tool has identified, for example, all the quality attributes related to the User Authentication scenario: (i) performance quality attribute – because the `getJdbcTemplate()`method belongs to a possible path; (ii) the reliability quality attribute because the method `initDataSourceJndi()` also belongs to a possible path; and (iii) finally, the security quality attribute for the same reason, the method `toMD5()` is used to encrypt the user password.

Finally, for answering the third question – (iii) what are the scenarios that contain potential tradeoff points among quality attributes? – the tool looks for scenarios affected by more than one quality attribute because they potentially contain tradeoff points. The tool has identified: (i) user authentication and mobile user authentication are potential scenarios to have tradeoff because they are affected by performance, security and reliability; (ii) authenticated document generation scenario is another potential tradeoff point because it addresses the reliability and security quality attributes; and, on the other hand, (iii) the sending message scenario does not represent a tradeoff point because it is only affected by the security quality attribute. Table 1 summarizes these results.

The information automatically identified by our tool is useful to indicate to the architects and developers which specific scenarios and code assets they need to give more attention when evaluating or evolving the software architecture through the conduction of code inspections, or during the execution of manual and automated testing. In that way, our preliminary evaluation in a large-scale enterprise system has allowed us to answer the expected questions previously highlighted showing the feasibility of our static analysis approach.

4.2 e-Commerce Web System

The evaluation of the dynamic analysis was performed by applying our tool to the EasyCommerce web system [10, 11], which is an e-commerce system that has been

Table 1. Some information about tradeoffs in scenarios.

User Authentication		Mobile User Authentication	
Performance:	getJdbcTemplate()	Performance:	getJdbcTemplate()
Security:	toMD5()	Security:	toMD5()
Reliability:	initDataSourceJndi()	Reliability:	initDataSourceJndi()
Tradeoff:	Potential	Tradeoff:	Potential
Authenticated Document Generation		Sending Message	
Performance:	–	Performance:	–
Security:	createRegistry()	Security:	enqueue()
Reliability:	initDataSourceJndi()	Reliability:	–
Tradeoff:	Potential	Tradeoff:	No

developed by graduate students from our research group. It implements a concrete product of an e-commerce software product line described in [12]. During the evaluation process, we also have applied the static analysis to this system. This section shows both results.

This evaluation has two main aims: (i) finding out if there is potential scenarios that contain tradeoff points between their quality attribute; and (ii) understanding how quality attributes affect each scenario, for example, analyzing the execution time of scenarios and if they have failed or not. The static analysis has been used to explore the first aim. On the other hand, to the second aim, we need to use the dynamic analysis. Next subsections describe the steps followed.

4.2.1 Choosing Evaluation Scenarios

We have chosen some of the scenarios that represent the main features of Easy-Commerce: (i) registration of login information – it records the user information about login, such as user name and password; (ii) registration of personal information – it records personal information about the user, such as name, address, birthday, document identification; (iii) registration of credit card information – it records information about users credit card, such as card number and expiration date; (iv) searching for products – it allows searching for products by its name, type or features; and (v) inserting product item to cart – it allows users adding a product item to their shopping cart.

4.2.2 Identifying Scenarios

In this step, the starting execution methods for each chosen scenario were identified. They are, respectively, in the same order of the scenarios: (i) `registerLogin()`, (ii) `registerUser()` and (iii) `registerCreditCard()` from `RegistrationMBean` class that contains methods responsible for registering of user information; (iv) `searchProducts()` from `SearchMBean` class that includes methods for basic and advanced search of products; and (v) `includeItemToCart()` from `CartMBean` class that manages the virtual user-shopping cart.

4.2.3 Identifying Quality Attributes

We have chosen some methods belonging to the scenarios that appear to have potential to be relevant to specific quality attributes. The selected ones were:

(i) `GenericDAOImpl.save()` with `@Performance`, `@Reliability` and `@Robustness` – it is used by the system to save all its objects, because of that it should run as fast as possible and it also represents a critical action;
(ii) `RegistrationMBean.registerLogin()`, `RegistrationMBean.registerUser()`, and `RegistrationMBean.registerCreditCard()` with `@Security` – these methods manipulate user confidential information and they are in some way related to security.

4.2.4 Executing Static and Dynamic Analysis: Preliminary Results

First, we have executed the static analysis. Table 1 presents an overview of the generated report. The tool has detected for each scenario the relevant quality attributes. For example, all scenarios of registration of information are related to security because their starting method was annotated with @Security. The registration of credit card scenario also has reliability, performance and robustness quality attributes because some of the methods inside this scenario has specific annotations, in this case, the `save()` method. Looking at Table 1, we can identify potential trade-off points between quality attributes. The registration of credit card scenario, for example, is associated to four quality attributes, where security and performance are classical examples of quality attribute conflict. Because of that, this scenario was identified as a potential tradeoff point.

In order to evaluate the behavior of the performance and reliability quality attributes, we have used the dynamic analysis to quantify them. The system was executed to exercise the chosen scenarios thus enabling the monitoring by the aspects.

Table 2 shows some information collected by the aspect, which were extracted during the execution of the following scenarios: register of login, register of personal information, and register of credit card information. By executing these scenarios, we have one occurrence of performance, reliability and robustness in `save()` method and three occurrences of security in `registerLogin()`, `registerUser()` and `registerCreditCard()`. For each scenario, the tool also calculated the execution time and the failure rate. Table 3 shows some examples.

Table 2. Scenarios and quality attributes which affect them.

Scenario	Quality Attribute
Registration of login information	Security
Registration of personal information	Security
Registration of credit card	Reliability Security Performance Robustness
Search for products	–
Include product item to cart	–

Table 3. Sample of data collected by dynamic analysis.

Register of login	Register of personal information	Register of credit card information
Execution time: 4 ms	Execution time: 3 ms	Execution time: 152 ms
Failure rate: 0 %	Failure rate: 0 %	Failure rate: 0 %
@Performance: -	@Performance: -	@Performance: save()
@Security: registerLogin()	@Security: registerUser()	@Security: registerCreditCard()
@Reliability: -	@Reliability: -	@Reliability: save()
@Robustness: -	@Robustness: -	@Robustness: save()

The dynamic analysis process in this study has met our expectations because it has allowed us extracting useful information of the execution context, such as, monitoring of scenarios and quality attributes, calculating the execution time and failure rate of scenarios and last, but not least, helping the detection of executed paths with potential tradeoff points.

5 Related Work

To the best of our knowledge, there is no existing proposal that looks for the automation of architecture evaluation methods using annotation and code analysis as we have proposed in this paper. In this section, we summarize some research work that address architectural evaluation or propose analysis strategies similar to ours.

Over the last years, several architecture evaluation methods, such as ATAM, SAAM, ARID [1] and ALMA [2] have been proposed. They rely on manual reviews before the architecture implementation. Our approach complements these existing methods by providing automated support to static and dynamic analysis over the source code of the software system. It contributes to the continuous evaluation of the software architecture during the system implementation and evolution.

Some recent research work have proposed adding extra architectural information to the source code with the purpose of applying automated analysis or document the software architecture. In that way, Christensen et al. [13] use annotations to add information about components and design patterns with the purpose of documenting the architecture. On the other hand, Mirakhorli et al. [14] present an approach for tracing architecturally significant concerns, specifically related to architectural tactics which are solutions for a wide range of quality concerns. These recent research work, however, do not explore the integrated usage of adding information related to scenarios or quality attributes with dynamic and static code analysis.

The approach presented in [15] proposes to identify when the developer changes to the program source code, tests, or environment affect the system behavior. That approach does not address scenarios or quality attributes in order to provide any kind of architectural evaluation, but parts of the technique that has been applied is similar with ours. In the same way of our approach, it builds the static graph through the source code analysis and the dynamic graph with AspectJ during the tests execution.

It can be used, for example, to confront the graphs and determine if changes in the system (static graph) affect the system behavior (dynamic graph), or vice versa.

6 Conclusions

We presented an approach to automating the software architecture evaluation using the source code as input of this process. The approach proposes: (i) to add metadata to the source code in order to identify relevant scenarios and quality attributes for the architectural evaluation; and (ii) to execute static and dynamic code analysis that supports the automated architectural evaluation based on the annotated scenarios and quality attributes. The paper has also presented an implementation of a tool that supports the static and dynamic analysis of the approach for the architectural evaluation of systems implemented in the Java language. Finally, we have also described the application of the approach in two existing systems: a large-scale enterprise information system and an e-commerce web system. The preliminary obtained results of the approach usage have allowed us to provide and quantifying several and useful information about architecture evaluation based on scenarios and quality attributes.

The presented approach is still under development and we are currently evolving it in order to apply to other large-scale enterprise information systems. The tool might be used to check missing paths [16], which happens when a path exists in the static call graph and it does not exist in the dynamic call graph meaning a not tested path or dead code.

In addition, we are also evolving the approach in order to analyze the architecture erosion of existing software systems by allowing the execution of the static and dynamic code analysis over different versions of the same software system. The main aim is to observe how quality attributes evolve when the scenarios implementation are changed and evolved. In this new implementation, our tool is used: (i) to persist the information extracted and quantified from each version of the system; and (ii) to compare the obtained results for the different versions of the system in order to verify if the code changes have caused degradation of the system quality attributes. Finally, the tool is also been extended to mine the source code repository of the system (e.g. subversion repository system), to find out the tasks that are responsible to introduce these changes, and to indicate which kind of tasks are more likely to cause degradation to specific quality attributes.

Acknowledgements. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES) - CNPq under grants 573964/2008-4 and CNPq 560256/2010-8, and the Informatics Superintendence (SINFO) from Federal University of Rio Grande do Norte (UFRN), Brazil.

References

1. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, MA (2002)
2. Bengtsson, P., Lassing, N., Bosch, J., Vliet, H.: Architecture-level modifiability analysis (ALMA). J. Syst. Softw. **69**, 1–2 (2004)

3. Kazman, R., Abowd, G., Bass, L., Clements, P.: Scenario-based analysis of software architecture. IEEE Softw. **13**(6), 47–55 (1996)
4. Kazman, R., Klein, M., Clements, P.: ATAM: Method for Architecture Evaluation. Technical report, CMU/SEI-2000-TR-004, ESC-TR-2000-004, Software Engineering Institute, August 2000
5. Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: a survey. J. Syst. Softw. **85**(1), 132–151 (2012)
6. Abi-Antoun, M., Aldrich, J.: Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. SIGPLAN Not. **44**, 321–340 (2009)
7. Babar, M.A., Gorton, I.: Comparison of scenario-based software architecture evaluation methods. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04), pp. 600–607. IEEE Computer Society, Washington, DC (2004)
8. Wala, T.J.: Watson Libraries for Analysis, September 2013. http://wala.sourceforge.net
9. SINFO/UFRN (Informatics Superintendence), September 2013. http://www.info.ufrn.br/wikisistemas
10. Torres, M.: Systematic Assessment of Product Derivation Approaches. MSc Dissertation, Federal University of Rio Grande do Norte (UFRN), Natal, Brazil (2011) (in Portuguese)
11. Aquino, H.M.: A Systematic Approach to Software Product Lines Testing. MSc Dissertation, Federal University of Rio Grande do Norte (UFRN), Natal, Brazil (2011) (in Portuguese)
12. Lau, S.Q.: Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates, MSc Dissertation, University of Waterloo (2006)
13. Christensen, H.B., Hansen, K.M.: Towards architectural information in implementation (NIER track). In: Proceedings of the 33rd International Conference on Software Engineering, (ICSE '11), pp. 928–931. ACM, New York (2011)
14. Mirakhorli, M., Shin, Y., Cleland-Huang, J., Cinar, M.: A tactic-centric approach for automating traceability of quality concerns. In: Proceedings of the 2012 International Conference on Software Engineering (ICSE 2012), pp. 639–649. IEEE Press, Piscataway (2012)
15. Holmes, R., Notkin, D.: Identifying program, test, and environmental changes that affect behaviour. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pp. 371–380. ACM, New York (2011)
16. Liu, S. Zhang, J.: Program analysis: from qualitative analysis to quantitative analysis (NIER track). In: Proceedings of the 33rd International Conference on Software Engineering (ICSE '11), pp. 956–959. ACM, New York (2011)