Um Framework Baseado em Metadados para Análise de Degradação de Atributos de Qualidade em Sistemas Web

Felipe Pinto[1 2], Uirá Kulesza[1], Leo Silva[1 2], Eduardo Guerra[3]

[1]Universidade Federal do Rio Grande do Norte, Natal, Brasil

[2]Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte, Natal, Brasil

[3]Instituto Nacional de Pesquisa Espacial, São José dos Campos, Brasil

felipe.pinto@ifrn.edu.br, uira@dimap.ufrn.br, leopontosilva@gmail.com, guerraem@gmail.com

ABSTRACT

This paper presents a metadata-based framework for software architecture evaluation of quality attributes. It implements 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 web-based systems. The evaluation process has three phases: (i) dynamic analysis that collects information of scenarios in terms of measurable quality attributes; (ii) degradation analysis that processes and compares the results of the dynamic analysis in term of quality attributes for two or more existing releases of a web-based system to identify degraded scenarios considering the desired quality attributes; (iii) repository mining that looks for development issues and commits associated to code assets of the degraded scenarios. The paper also presents and discusses the obtained results of the framework instantiation for the library module of a large-scale web system.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *Frameworks*.

General Terms

Measurement, Performance, Reliability, Security.

Keywords

Metadata-based framework, scenario, quality attributes, dynamic analysis, repository mining.

1. INTRODUÇÃO

Manter e evoluir sistemas web de larga escala é uma tarefa que tem se tornado crítica, devido à complexidade de tais sistemas e do surgimento de novos requisitos de clientes e tecnologias. Atributos de qualidade, como desempenho e confiabilidade, são essenciais para que esses sistemas possam cumprir seus objetivos. Ao evoluir esses sistemas, desenvolvedores precisam estar atentos ao impacto das requisições de mudanças, como correção de *bugs* ou adição de novas funcionalidades, na degradação de algum atributo de qualidade, por exemplo. Dessa forma, pode ocorrer do sistema implementado se afastar do que foi projetado inicialmente em sua arquitetura, ocorrendo o processo de erosão arquitetural [1].

Considerando que a arquitetura de um software é resultado de um conjunto de decisões de projeto [2] que podem afetar diversos elementos do sistema, incluindo sua estrutura e seus atributos de qualidade, podemos afirmar que degradações dos atributos de qualidade contribuem para a erosão arquitetural do sistema. A degradação de um atributo de qualidade ocorre quando, após uma evolução do sistema, o mesmo passa a fornecer experiências de uso inferiores à versão anterior para os mesmos cenários. Um cenário é uma ação de alto nível no sistema, representando a forma que os *stakeholders* esperam que ele seja usado [3].

De forma geral, existem diversas propostas para avaliação de atributos de qualidade em nível arquitetural baseadas em cenários [3][4][5][6] ou que visam lidar com problemas de erosão arquitetural [1]. Entretanto, há uma ausência de trabalhos de pesquisa nessa área que considerem a implementação do sistema para indicar quais mudanças foram responsáveis por degradar os atributos de qualidade durante sua evolução. As abordagens atuais, que trabalham com a implementação, em geral, são voltadas para a verificação da conformidade estrutural do sistema [7][8][9][10]. Outras fazem uso de modelos matemáticos para predição dos atributos de qualidade [11][12] na ausência da implementação. É possível também usar *benchmarks* para análise de atributos de qualidade, como desempenho [13], mas, nesses casos, a análise é feita sem considerar conceitos arquiteturais de mais alto nível, como os cenários relevantes do sistema. Adicionalmente, alguns trabalhos recentes têm dado atenção a determinar como *bugs* de desempenho são tradados no ciclo de desenvolvimento do sistema, ou seja, como eles são descobertos, reportados e corrigidos [14], ou definir e minerar repositórios na tentativa de identificar as causas de variações do desempenho devido às evoluções do sistema [15], ou ainda, supervisionar a evolução da orquestração de serviços web com base na documentação dos requisitos de qualidade [16].

Parte desses trabalhos apresentam aplicações de suas propostas em sistemas web [13][15][16]. Entretanto, nenhum deles aborda uma forma de identificar quais artefatos de código (classes, métodos e pacotes), requisições de mudança (*issues*) e contribuições (*commits*) são responsáveis pelas degradações. Este trabalho apresenta um *framework* baseado em metadados para avaliação de atributos de qualidade através de análise dinâmica e mineração de repositório. O objetivo é revelar de forma automatizada as degradações que ocorreram em cenários relevantes arquiteturalmente em uma nova versão, provendo a avaliação arquitetural contínua em termos de atributos de

qualidade. As principais contribuições deste trabalho são: **(i)** uma abordagem para avaliação de atributos de qualidade guiada por cenários através de técnicas de análise dinâmica e mineração de repositório de software; **(ii)** a implementação de um *framework* extensível para automatizar a abordagem; e **(iii)** um estudo de caso que apresenta uma instanciação do *framework* e os resultados de sua aplicação para o SIGAA, um sistema web real de larga escala para gestão acadêmica.

Este trabalho está organizado como segue: a Seção 2 apresenta uma visão geral da abordagem de avaliação; a Seção 3 detalha o *framework* proposto; a Seção 4 apresenta um estudo de caso para o atributo de qualidade de desempenho e análise de conflitos; a Seção 5 discute os resultados do estudo; a Seção 6 reporta os trabalhos relacionados; e a Seção 7 conclui o trabalho.

2. VISÃO GERAL DA ABORDAGEM

A abordagem de avaliação requer o código fonte do sistema integrado com informações de nível arquitetural fornecidas por metadados. Neste trabalho, anotações de código Java foram usadas como fontes de metadados. A função delas na abordagem é identificar métodos chaves para o processo, incluindo aqueles que representam pontos de entrada para a execução de cenários e aqueles que implementam decisões arquiteturais que podem afetar o atendimento aos atributos de qualidade do sistema.

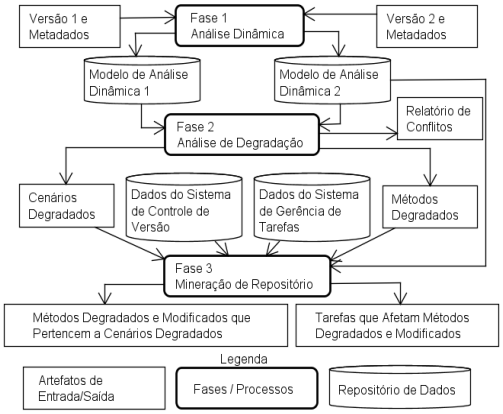

Figura 1. Visão geral da abordagem de avaliação.

A Figura 1 apresenta uma visão geral da abordagem. A primeira fase de análise dinâmica requer a execução dos cenários escolhidos para avaliação através de testes funcionais, manuais ou automatizados. Os métodos de entrada dos cenários e os métodos que implementam decisões arquiteturais que impactam atributos de qualidade são indicados através de anotações. Durante essa fase, é gerado em tempo de execução o modelo de análise dinâmica que deve ser persistido para um banco de dados. Ele contém informações sobre a execução dos cenários modelado através de um grafo de chamadas. Basicamente, o fluxo de execução a partir dos métodos de entrada dos cenários é capturado e mapeado para o grafo, enquanto são calculados os atributos de qualidade. Dessa forma, cada nó no grafo representa a execução de um método (construtor ou método regular).

A análise de degradação é a segunda fase da abordagem. Nesta, ocorre a comparação dos dados extraídos durante a análise dinâmica de duas versões do sistema. Essa comparação revela os cenários e métodos que foram degradados em termos de atributos de qualidade durante a evolução e gera um relatório contendo quais atributos de qualidade afetam os cenários avaliados, com o objetivo de indicar cenários com potenciais conflitos entre os atributos de qualidade. É importante perceber que durante a análise dinâmica, o tempo de execução dos métodos, por exemplo, é influenciado pelos métodos invocados por ele. Neste caso, a degradação de um método irá afetar todos os nós pais dele no grafo. A análise de degradação irá considerar todos esses métodos como degradados, mas apenas um é a real fonte de degradação. A próxima fase da abordagem busca filtrar, dentre os métodos degradados, os que podem ser responsáveis por isso.

A última fase da abordagem é a mineração de repositório. Ela consiste na mineração de dados nos sistemas de controle de versão e gerência de mudanças que estão relacionados especificamente com os métodos identificados como degradados na fase anterior. Basicamente, os *commits* que introduziram mudanças para cada método degradado são recuperados do sistema de controle de versão. A partir daí, o comentário (*log*) do *commit* é analisado em busca dos números das tarefas (requisições de mudanças) associadas. É feita uma consulta no banco de dados do sistema de gerência de mudanças, de forma a recuperar informações da tarefa, como seu tipo e *status*, por exemplo. Nesta fase, os métodos detectados como degradados, mas que não foram modificados, são descartados, porque eles não representam fontes reais de degradação, tendo sido impactados apenas pela degradação de outros métodos. Como a abordagem é guiada por cenários, métodos só serão considerados no conjunto da solução se eles pertencerem a cenários degradados, mesmo que individualmente tenham sido degradados e modificados.

Assim, a última saída tem como principais artefatos: **(i)** a lista de métodos modificados que contribuíram para degradação de cenários; **(ii)** as modificações (*commits*) e tarefas associadas a esses métodos, responsáveis por introduzir tais mudanças. Essas informações permitem que desenvolvedores analisem o conjunto de tarefas cuja implementação causou degradação, para entender o impacto que elas tiveram na arquitetura.

3. FRAMEWORK PROPOSTO

O *framework* foi implementado na linguagem de programação Java, utiliza AspectJ para interceptar a execução do sistema e anotações de código como fonte de metadados. Atualmente, é capaz de automatizar a abordagem proposta (Seção 2, Figura 1) para os atributos de qualidade de desempenho e confiabilidade, considerando as propriedades de tempo de execução e taxa de falhas, além de prover rastreabilidade de código através de anotações para desempenho, confiabilidade e segurança com indicação de potenciais conflitos. A Figura 2 mostra um diagrama parcial com os principais módulos do *framework*: `DynamicAnalyzer`, `DynamicModel`, `RepositoryMiner` e `ChangeModel`. Classes e interfaces na cor branca representam pontos fixos, enquanto a cor cinza indica pontos de extensão. Classes específicas do exemplo de instanciação do estudo apresentado na Seção 4 são indicadas na cor azul, enquanto as classes na cor cinza podem ser reusadas em outras instanciações.

3.1 Módulo DynamicAnalyzer

O módulo `DynamicAnalyzer` contém recursos relacionados aos metadados e à primeira fase da abordagem (análise dinâmica). A anotação `Scenario` possui um atributo `name` que identifica unicamente cada um dos pontos de entrada anotados no código fonte. A anotação `QualityAttribute` é usada nas anotações

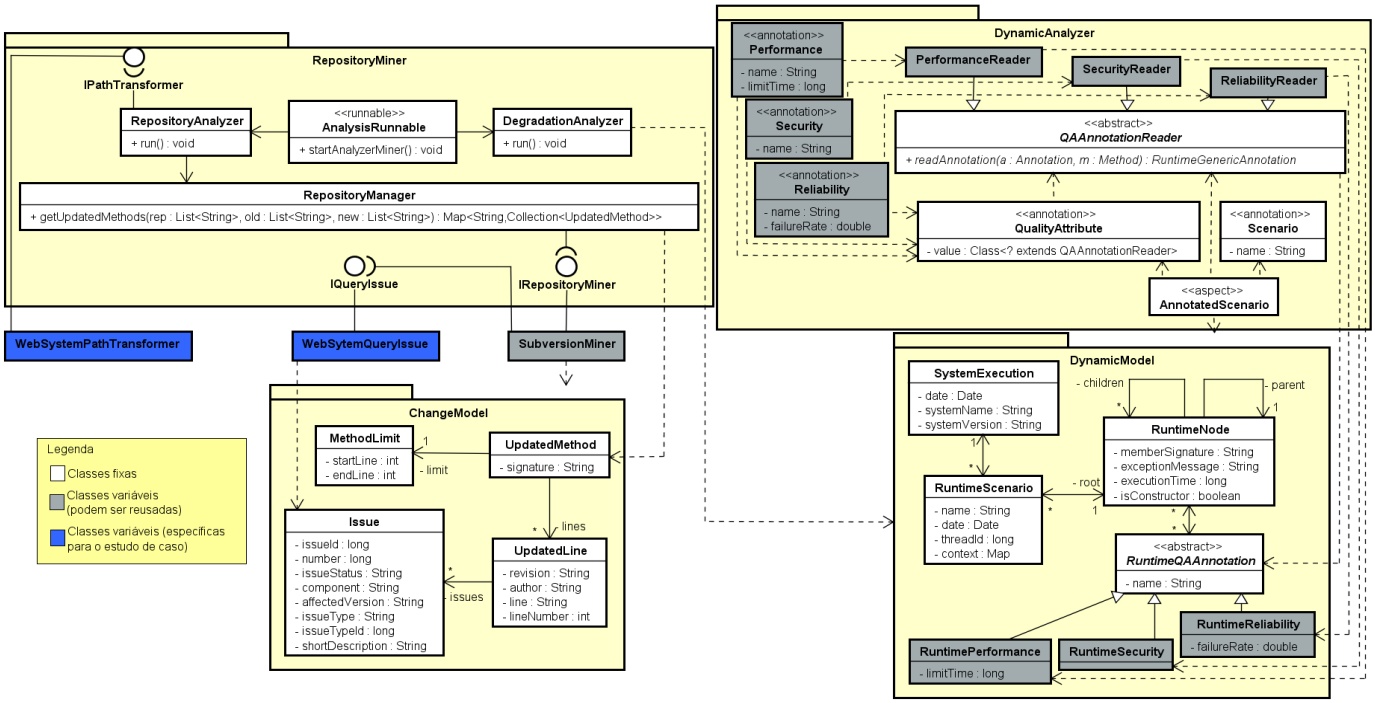

Figura 2. Diagrama de classes parcial do framework proposto.

específicas de atributos de qualidade para indicar a implementação de `QAAnnotationReader` responsável por ler e interpretar aquela anotação. Cada anotação de atributos de qualidade modelada por `Performance`, `Security` e `Reliability` possui outra anotação que indica seu leitor, respectivamente, `PerformanceReader`, `SecurityReader` e `ReliabilityReader`.

A anotação `Performance` tem o atributo `limitTime` usado, opcionalmente, para especificar o tempo máximo esperado para que a execução do método termine. A anotação `Reliability` possui o atributo `failureRate` que especifica a taxa de falhas máxima esperada do método associado. Já a anotação `Security` não possui atributos específicos e é responsável por indicar que a implementação de um determinado método afeta este atributo de qualidade. As anotações de atributos de qualidade são usadas para identificar partes do código que são associadas com eles e que implementam decisões arquiteturais que os afetam. O *framework* pode ser estendido para novas anotações de atributos de qualidade, sendo possível rastrear tais atributos através das anotações, consultando os modelos persistidos e, por exemplo, determinar quais cenários podem potencialmente conter pontos de conflito entre atributos de qualidade, considerando cenários afetados por vários deles. Por exemplo, pode-se supor que em uma rede privada virtual é possível melhorar a segurança do sistema através do aumento da quantidade de bits para encriptação, porém isso requer mais tempo de processamento.

O aspecto `AnnotatedScenario` encontra essas anotações de atributos de qualidade durante a execução do sistema e usa as classes de processamento das anotações para interpretá-las. O aspecto é responsável por popular o modelo de análise dinâmica (módulo `DynamicModel`), durante a fase de análise dinâmica da abordagem, interceptando os fluxos de execução iniciados por métodos marcados como pontos de entrada de cenários, ou seja, anotados com a anotação `Scenario`. Durante a instrumentação do sistema, o aspecto constrói e armazena em banco de dados o modelo de análise dinâmica, incluindo o grafo de chamadas, progressivamente, ao fim da execução de cada cenário. Além de rastrear tais anotações, atualmente, o aspecto implementado também calcula o desempenho e a confiabilidade para todos os cenários anotados com `Scenario` através do tempo de execução e da taxa de falhas, respectivamente.

3.2 Módulo DynamicModel

O módulo `DynamicModel` representa o modelo de análise dinâmica e modela a execução dos cenários de um determinado sistema. Ele é construído e persistido pelo aspecto `AnnotatedScenario` durante a análise dinâmica. O modelo de análise dinâmica é a saída da primeira fase e entrada da segunda fase da abordagem, de acordo com o ilustrado na Figura 1.

A classe `SystemExecution` representa a execução dos cenários de um sistema particular, indicando o nome e a versão do sistema, bem como a data de execução. Esta classe está associada com as execuções de cenários (`RuntimeScenario`). Todo cenário executado possui um nome (configurado pela anotação `Scenario`), a data de execução, o identificador da *thread* que o executou e o contexto que representa uma requisição web. Neste caso de avaliação para sistemas web, armazenar a requisição é útil para ter uma forma de diferenciá-las.

A classe `RuntimeNode` modela os nós do grafo de chamadas para métodos executados dentro dos cenários. Ela mantém informação sobre a assinatura do método, a mensagem caso alguma exceção tenha sido lançada, o seu tempo de execução e a indicação se o nó representa a execução de um método regular ou de um construtor. Adicionalmente, um nó também pode conter atributos de qualidade associados através das anotações inseridas nos métodos. A Figura 2 mostra três possíveis especializações da classe `RuntimeQAAnnotation` para os atributos de qualidade de desempenho, segurança e confiabilidade, respectivamente,

`Performance`, `Security` e `Reliability`, que são instanciadas pelos seus respectivos leitores no módulo `DynamicAnalyzer` durante a análise dinâmica.

3.3 Módulo RepositoryMiner

O módulo `RepositoryMiner` é responsável por acessar os dados dos sistemas de controle de versões e gerência de mudanças, além da análise de degradação, executando a segunda e terceira fases da abordagem. Dois modelos de análise dinâmica (`DynamicModel`) de duas versões de um sistema são comparados (fase 2) gerando a lista de cenários e métodos degradados e o relatório de conflitos, em seguida, os métodos degradados são minerados nos repositórios de dados (fase 3).

A classe `AnalysisRunnable` inicia e coordena os processos de análise de degradação e mineração de repositório. Inicialmente, ela usa uma instância de `DegradationAnalyzer` para executar a análise de degradação através da comparação de dois modelos de análise dinâmica que representam as execuções de duas versões de um mesmo sistema. Ao fim, são produzidas duas listas contendo todos os cenários e métodos degradados para os atributos de qualidade considerados e um relatório contendo os cenários afetados por determinados atributos de qualidade, indicando aqueles com potencias conflitos (Figura 1).

Em seguida, a classe `AnalysisRunnable` usa uma instância de `RepositoryAnalyzer` para minerar dados no sistema de controle de versões relacionados aos arquivos que contém as declarações dos métodos presentes na lista de degradados. Cada linha dos arquivos é verificada para descobrir qual *commit* introduziu a última modificação nela para o intervalo de versões considerada. Uma vez que o *commit* é encontrado procura-se na sua mensagem de *log* (comentário) quais os números das tarefas (requisições de mudança) associadas a ele. Finalmente, os números das tarefas são usados para recuperá-las através de consulta ao sistema de gerência de mudanças. Apenas *commits* associados com os métodos degradados são considerados.

O processo descrito acima é executado através do método `getUpdatedMethods()` da classe `RepositoryManager`. O primeiro parâmetro `rep` indica a lista de arquivos localizados no sistema de controle de versões que deverão ser minerados. O segundo e terceiro parâmetros, `old` e `new`, representam cópias de trabalho dos arquivos em uma versão passada e nova do sistema. O retorno é uma coleção de informações sobre a atualização dos métodos para cada arquivo minerado. Para finalizar o processo de mineração, a classe `RepositoryAnalyzer` verifica, para os métodos retornados por `RepositoryManager`, quais deles pertencem a cenários que foram degradados. Como o processo de avaliação arquitetural é guiada por cenários, não são considerados resultados que não impactam cenários degradados.

As três principais tarefas executadas no módulo `RepositoryMiner` podem variar de acordo com o sistema analisado: **(i)** minerar classes que declaram métodos degradados; **(ii)** descobrir a localização das classes a partir da assinatura dos métodos degradados; e **(iii)** recuperar tarefas a partir de comentários de *commits* e buscá-los no sistema de gerência de mudanças. Por esse motivo, o *framework* provê pontos de extensão que podem ser especializados para cada instanciação.

A primeira tarefa depende do sistema de controle de versões. Na Figura 2, está ilustrada a classe `SubversionMiner` que implementa o suporte para repositórios do tipo Subversion. A classe `RepositoryManager` acessa informações do repositório através da interface `IRepositoryMiner`.

Na segunda tarefa, é preciso descobrir a localização (caminho completo) de cada classe que declara métodos degradados para que elas sejam mineradas. Para alguns sistemas, essa informação não pode ser extraída apenas da assinatura completa do método, pois os pacotes do sistema podem estar organizados em pastas e subpastas de código. Esse mapeamento deve ser feito para cada sistema analisado implementando-se a interface `IPathTransformer`. A Figura 2 ilustra a classe `WebSystemPathTranformer` implementada para o sistema web do estudo que será apresentado.

Na terceira tarefa, para cada *commit* encontrado pela implementação de `IRepositoryMiner`, é preciso descobrir os números das tarefas associadas analisando o comentário do *commit*. Desenvolvedores de sistemas diferentes usam notações diferentes para indicar as tarefas dentro dos comentários. Em geral, os números podem ser recuperados através de uma busca com expressão regular. Para isso, a implementação de `IRepositoryMiner` deve usar a interface `IQueryIssue`. Na Figura 2, a classe `WebSystemQueryIssue` implementa esta interface, provendo suporte para tal operação no sistema web avaliado. Esta classe também deve consultar o banco de dados do sistema de gerência de mudanças usando o número e recuperar as informações da tarefa, incluindo seu tipo e *status*.

3.4 Módulo ChangeModel

O módulo `ChangeModel` representa o modelo de análise de repositório, e é formado por uma estrutura de classes que associam as informações dos *commits* e das tarefas aos métodos degradados. É com base nesse modelo que a classe `RepositoryAnalyzer` gera os relatórios finais do processo de avaliação resultantes da terceira fase da abordagem que contém a lista de métodos que foram degradados e modificados e, além disso, pertencem a cenários degradados durante a evolução. Esse modelo de análise de repositório é construído pelas implementações das interfaces `IRepositoryMiner` e `IQueryIssue`, responsáveis pelo acesso aos sistemas de controle de versões e gerência de mudanças, respectivamente.

A classe `UpdatedMethod` representa métodos degradados que foram modificados. Todo `UpdatedMethod` é limitado pelo intervalo das linhas de código onde a declaração do método inicia e termina, representado pela classe `MethodLimit`. O método pode ter tido mudanças em várias linhas, dessa forma, `UpdatedLine` modela as modificações feitas linha por linha. Finalmente, cada linha pode ter sido modificada por uma ou mais tarefas modeladas pela classe `Issue`.

4. ESTUDO DE CASO

Esta seção apresenta a aplicação do *framework* em um sistema web real de larga escala chamado SIGAA (Sistema Integrado de Gestão de Atividades Acadêmicas) desenvolvido pela superintendência de informática (SINFO) [17] da UFRN.

4.1 Objetivos e Questões de Pesquisa

Este estudo teve como objetivos: **(i)** avaliar a instanciação do *framework* em um sistema web real; **(ii)** analisar a evolução de um sistema web real de larga escala em termos do atributo de

desempenho; **(iii)** analisar a introdução de potenciais novos conflitos durante evolução considerando as anotações de atributos de qualidade providas. Considera-se que o estudo de caso terá sucesso caso seja possível responder as seguintes questões de pesquisa: **(RQ1)** Quais dos cenários analisados sofreram degradação de desempenho? **(RQ2)** Quais módulos/pacotes contém métodos degradados? **(RQ3)** Quais tarefas de desenvolvimento foram responsáveis pelas mudanças nos elementos degradados? **(RQ4)** Houve introdução de potenciais novos conflitos nos cenários analisados?

4.2 Exemplo de instanciação

O alvo da avaliação é o SIGAA, um sistema web para gestão acadêmica. Atualmente, mais de 20 universidades no Brasil usam customizações desse sistema. Sua implementação segue a seguinte estrutura em camadas: *Interface Gráfica de Usuário*, *Serviços*, *Negócio* e *Persistência*. Tipicamente, em sistemas web implementados com o *framework Java Server Faces* (JSF), caso deste estudo, as execuções dos cenários iniciam em métodos de *managed beans*. Isso significa que uma vez selecionados os cenários que deseja-se avaliar, deve-se identificar quais métodos nos *managed beans* representam os pontos de entrada de execução das requisições web, marcando-os com a anotação de cenário fornecida pelo *framework*. Para este estudo, os cenários selecionados pertencem ao módulo biblioteca do sistema e foram escolhidos em conjunto com a equipe de desenvolvimento, a qual também auxiliou na identificação dos métodos que representam os pontos de entrada para a execução desses cenários. Os cenários e seus pontos de entrada são mostrados na Tabela 1.

Tabela 1. Cenários selecionados para avaliação.

Nome do Cenário	Ponto de Entrada (Método)
Pesquisa Simples no Acervo	Classe: PublicSearchLibraryMBean Método: simpleSearch()
Emissão de Declaração de Quitação	Classe: VerifyUserSituationLibraryMBean Método: generateStatusCertificate()
Realizar Empréstimo	Classe: CirculationModuleMBean Método: lendItem()
Finalizar Empréstimo	Classe: CirculationModuleMBean Método: returnItem()
Verificar Situação do Usuário	Classe: UserSearchLibraryMBean Método: selectedUser()
Renovar Empréstimo	Classe: CirculationModuleMBean Método: renewItem()

Após anotar os métodos de entrada, alguns outros métodos identificados como críticos para o atendimento de determinados atributos de qualidade foram anotados com as anotações específicas para tal finalidade. Para isso, é necessário estender o *framework* para definir as anotações de atributos de qualidade. Isso é feito através das classes `QAAnnotationReader` e `RuntimeQAAnnotation` de forma a prover um processador (leitor) para a nova anotação e tornar o modelo de análise dinâmica compatível com ela, respectivamente. A nova anotação deve ser anotada com a anotação `QualityAttribute` para indicar seu processador. Um exemplo parcial desse procedimento é mostrado na Figura 3 e na Figura 4 para a anotação `Performance`. Note que a declaração desta anotação (Figura 3) está anotada com `QualityAttribute` que indica qual a classe capaz de processá-la (`PerformanceReader`). O mesmo procedimento foi feito para definir as anotações de `Reliability` e `Security`, como ilustrado na Figura 2. Os métodos associados aos atributos de qualidade foram anotados com o auxílio de um dos desenvolvedores do SIGAA.

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

Figura 3. Definição de anotação para atributo de desempenho.

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

Figura 4. Exemplo de implementação para novo leitor.

Após a anotação do código do sistema, deve-se configurar seu processo de construção para que o aspecto possa interceptar a execução. Em seguida, basta executar o sistema de forma a exercitar os cenários selecionados, por exemplo, através de seus testes, e o aspecto se encarrega do processo de análise dinâmica. Este procedimento precisa ser realizado para as duas versões do sistema, como ilustrado na Figura 1.

A execução da fase de análise de degradação é feita logo após a primeira. Já a fase final de mineração exige que sejam fornecidas implementações para as interfaces `IPathTransformer`, `IQueryIssue` e `IRepositoryMiner` como explicado na Seção 3.3. O SIGAA usa um repositório de código do tipo Subversion e um sistema de gerência de mudanças proprietário da SINFO. As classes `WebSystemPathTransformer`, `WebSystemQueryIssue` e `SubversionMiner` foram implementadas para atender essas necessidades (Figura 2). Vale lembrar que `SubversionMiner` pode ser reusada para qualquer sistema que use Subversion. Essas classes são especificadas em um arquivo de propriedades para cada instância do *framework* e automaticamente instanciadas por ele durante a mineração.

4.3 Procedimentos de execução

As versões analisadas foram 3.11.24 de agosto de 2013 e 3.12.18 de dezembro de 2013. Os cenários selecionados pertencem ao módulo biblioteca do SIGAA. Na versão 3.11.24 o sistema completo e seu módulo biblioteca tem aproximadamente 673.610 e 98.041 linhas de código. Já na versão 3.12.18, os valores são 701.257 e 98.201. A análise dinâmica foi conduzida em um computador Intel Core i7, 16GB de RAM, rodando Windows 7, JBoss SA 4 e Java 7. A análise de degradação foi configurada para considerar degradações de desempenho quando o tempo de execução do cenário ou método analisado na nova versão é maior que 5% do valor na versão antiga. Cada conjunto de testes de execução de cenários foi repetida em torno de 10 vezes para obter uma boa precisão da média do desempenho. Como o sistema web não tinha testes automatizados, a alternativa foi registrar requisições manuais com o JMeter [18] e, em seguida, usá-lo para reproduzir sua execução um certo número de vezes.

4.4 Resultados

(RQ1) Quais dos cenários analisados sofreram degradação de desempenho? O uso do *framework* detectou que todos os seis cenários avaliados foram degradados. Este resultado pode ser visto no gráfico da Figura 5 que mostra o tempo médio de resposta para os cenários nas versões antes e depois da evolução analisada. A maioria dos cenários apresentou degradação de desempenho variando entre 78% e 125%. Apenas *Pesquisa Simples no Acervo* apresentou um pequeno aumento de 13%.

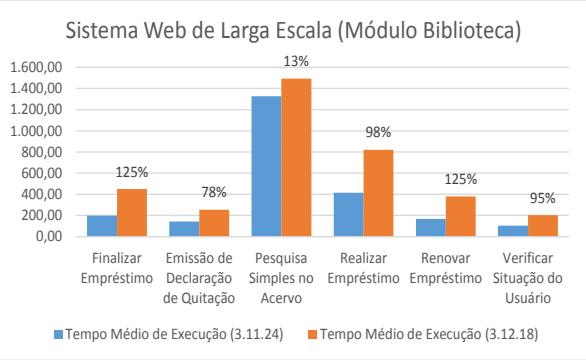

Figura 5. Degradação dos cenários analisados.

(RQ2) Quais módulos/pacotes contém os métodos degradados? A fase de mineração encontrou nove métodos que foram degradados e modificados durante a evolução do sistema. A Tabela 2 apresenta os métodos degradados, os cenários que eles afetam, o número de outras execuções de métodos que eles impactam e o aumento de seu tempo de execução individual. Já a distribuição desses métodos dentro dos pacotes do sistema é mostrada na Tabela 3. Nesta, pode-se observar o total de métodos degradados em cada pacote e o somatório dos aumentos do tempo de execução de cada método degradado do pacote.

Tabela 2. Impacto de métodos degradados para seus cenários.

Classe e Método Degradado	Cenário Impactado	Métodos Impactados	Tempo (ms)
PerformLoanService execute()	Realizar Empréstimo	6	221,04
RenewLoanService execute()	Renovar Empréstimo	6	160,70
LibraryServiceUtil generateExtendedLoan()	Realizar Empréstimo, Renovar Empréstimo	8	16,55
GenericDAOImpl findByExactField()	Realizar Empréstimo, Finalizar Empréstimo	10	7,54
GenericDAOImpl getSession()	Todos	76	5,32
SessionLogger registerCaller()	Todos	120	5,31
Utils toMD5()	Realizar Empréstimo, Renovar Empréstimo	5	0,81
Utils stackTraceInvocador()	Todos	121	0,22
Course getDescription()	Emissão de Declaração de Quitação, Realizar Empréstimo, Verificar Situação do Usuário	6	0,09

Tabela 3. Distribuição dos métodos degradados nos pacotes do sistema.

Pacote	Número de Métodos	Soma dos Aumentos dos Tempos de Execução (ms)
websystem.core.dao	2	12,87
websystem.core.security	1	5,31
websystem.core.util	2	1,03
websystem.library	3	398,29
websystem.domain	1	0,09

(RQ3) Quais tarefas de desenvolvimento foram responsáveis pelas mudanças nos elementos degradados? Foram encontradas duas tarefas. Uma do tipo correção de *bug* e outra do tipo melhoria. A primeira modificou o método `generateExtendedLoan()` da classe `LibraryServiceUtil` da camada de negócio para reduzir os acessos ao banco de dados durante o cálculo dos dias extras para retornar um item emprestado considerando finais de semana e feriados, devendo a data ser estendida para o próximo dia útil. Isso implicou alterações nos métodos `execute()` das classes `RenewLoanService` e `PerformLoanService` para manter a compatibilidade. Essas mudanças afetaram os cenários *Realizar Empréstimo* e *Renovar Empréstimo*. Apesar da tentativa de otimizar o método, ele foi impactado por outros métodos degradados. Neste caso, a segunda tarefa modificou o método `getSession()` da classe `GenericDAOImpl` que introduziu uma nova chamada para o método `registerCaller()` da classe `SessionLogger`. Essa simples modificação foi responsável por impactar vários outros métodos e cenários. O método `registerCaller()` foi introduzido para implementar um serviço de auditoria para o sistema, sendo invocado muitas vezes durante a execução dos cenários. Ele chama o método `UtilService.stackTraceInvoker()` responsável por armazenar a pilha de execuções do sistema.

Finalmente, é importante perceber que a introdução de um método, tal como `registerCaller()`, pode impactar o desempenho de muitos outros métodos que dependem dele e mesmo seu aumento de tempo de execução tendo sido pequeno, como indicado na Tabela 2 (5.31ms), ele causa degradações mais severas do que métodos que tiveram aumento de tempo maior, pois a quantidade de outros métodos que dependem de `registerCaller()` e são afetados por ele é bem maior (120 execuções de métodos impactadas).

(RQ4) Houve introdução de potenciais novos conflitos nos cenários analisados? A Tabela 4 mostra o resultado da análise de conflitos obtida através de consultas às anotações armazenadas nos modelos de análise dinâmica. De forma geral, o *framework* considera que cenários que possuem mais de um método cuja implementação afeta um atributo de qualidade podem conter algum tipo de conflito. O objetivo desse tipo de análise é identificar para os desenvolvedores cenários que eles deveriam ter atenção especial quando evoluindo. Em particular, *Realizar Empréstimo* está associado com três atributos de qualidade e teve alta degradação de desempenho (Figura 5). Durante a evolução, não houve modificação nos cenários, em termos de quais atributos de qualidade os afetam.

Tabela 4. Mapeamento entre métodos anotados e cenários.

Métodos Associados com Atributos de Qualidade	Anotações
Cenário: Finalizar Empréstimo	
FacadeDelegate.execute()	Performance
Cenário: Emissão de Declaração de Quitação	
FacadeDelegate.execute()	Performance
ProcessorGeneratorEmission.createRecord()	Security
Cenário: Pesquisa Simples no Acervo	
CatalographicTitle.advancedSearch()	Performance
Cenário: Realizar Empréstimo	
FacadeDelegate.execute()	Performance
Utils.checkRole()	Security
AbstractController.getMBean()	Reliability
Cenário: Renovar Empréstimo	
FacadeDelegate.execute()	Performance
Utils.checkRole()	Security
Cenário: Verificar Situação do Usuário	
FacadeDelegate.execute()	Performance
Utils.checkRole()	Security

5. DISCUSSÃO E LICÕES APRENDIDAS

Adequabilidade em relação aos objetivos propostos. O estudo de caso demonstrou que o *framework* proposto foi estendido com

sucesso provendo os pontos de extensão necessários para se adaptar ao ambiente de desenvolvimento. Novos estudos serão realizados em outras aplicações com ambientes distintos para procurar identificar novos pontos onde podem haver questões específicas que podem requerer outros tipos de especializações.

Avaliação arquitetural. O *framework* foi instanciado e aplicado com sucesso para identificar fontes de degradação de desempenho em um sistema web de larga escala. Tal avaliação pode ser entendida como arquitetural uma vez que a análise é baseada em cenários arquiteturalmente relevantes para o sistema (Figura 5). Também é possível verificar quais classes e pacotes contém o maior número de métodos degradados (Tabela 2 e Tabela 3). Tais elementos de código, em geral, podem ser mapeados para componentes, identificando-se os artefatos de mais alto-nível que apresentam degradação. A possibilidade de avaliar os atributos de qualidade considerando conceitos de nível arquitetural, como os cenários e módulos do sistema, difere a abordagem proposta de outras que usualmente usam *benchmarks* para avaliar elementos de código isolados.

Precisão na Detecção das Origens das Degradações. A integração das técnicas de análise dinâmica e mineração de repositórios revelou-se uma estratégia interessante para aumentar a precisão na detecção das origens das degradações. A análise dinâmica mostrou que muitos métodos apresentaram degradação do desempenho devido ao impacto de outros métodos. A combinação desses resultados com a mineração do repositório de códigos possibilitou indicar os métodos potencialmente responsáveis pela origem das degradações detectadas, o que representa uma quantidade reduzida deles (Tabela 2).

Análise da Confiabilidade. O estudo também coletou informações sobre o atributo de confiabilidade, através do monitoramento de exceções lançadas pela aplicação para se calcular a taxa de falhas. Apenas o cenário *Realizar Empréstimo* falhou uma vez em cada repetição do conjunto de testes. Como uma repetição do conjunto de testes executa 13 vezes esse cenário, a taxa de falhas foi de 7.69% em ambas as versões. Os métodos indicados pelo *framework* como responsáveis pelas exceções foram `executeTransactionTemplate()` e `equalByPolicyData()` das classes `SystemFacadeBean` e `LoanPolicy`, respectivamente, que sempre falham no empréstimo seguinte, após um empréstimo personalizado. A mineração mostrou que não houve alterações em tais métodos na evolução e, de fato, o estudo descobriu um *bug* do sistema neste caso. Apesar disso, acreditamos que execuções pré-definidas, sejam testes manuais ou automatizados, não são a forma mais adequada de medir tal atributo. Como consequência, novos estudos estão sendo conduzidos considerando a análise de confiabilidade a partir de *logs* da execução dos cenários em ambientes de produção.

Limitações da Abordagem. A anotação manual dos pontos de entrada dos cenários considerados relevantes acaba sendo uma limitação pois requer um conhecimento arquitetural prévio do sistema. Essas anotações deveriam evoluir junto com a implementação permitindo a execução da abordagem em novas versões. Uma possibilidade menos intrusiva para o código fonte seria usar XML como fonte externa de metadados. Outro requisito é o armazenamento e versionamento de todos os artefatos relacionados às versões do sistema, bem como a rastreabilidade entre tarefas de desenvolvimento e as revisões do repositório de códigos, o que pode requerer grande disponibilidade de meios de armazenamento de dados. Finalmente, também é importante a disponibilidade de testes automatizados que permitam promover a execução dos cenários.

Limitações do Estudo. Apesar da quantidade de informação coletada pela análise dinâmica e pela mineração, não há garantia que os mesmos tipos de tarefas encontrados no estudo sejam os tipos usualmente responsáveis por degradações de desempenho que possam ocorrer em outras evoluções do sistema web analisado ou de outros sistemas do mesmo domínio do analisado.

6. TRABALHOS RELACIONADOS

Esta seção descreve trabalhos de análise de evolução de atributos de qualidade, particularmente o desempenho, e que apresentam estudos para sistemas web, assim como o presente trabalho.

Malik et al [13] propõe estratégias para ajudar analistas de desempenho a comparar testes de carga, permitindo-os encontrar variações de desempenho mais facilmente. Um conjunto de contadores filtrados (medições para uso de processador, memória e outras propriedades) é disponibilizado aos analistas para que eles possam se concentrar apenas nos casos relevantes e detectar as causas da variação de desempenho. Essa filtragem é realizada comparando os resultados de um teste de carga anterior com os resultados do novo. A abordagem avaliou um sistema industrial de larga escala e o sistema *open source* Dell DVD Store (DS2), um protótipo de website para *e-commerce*. Os dados gerados foram fornecidos pela empresa para o sistema industrial, e obtidos com um *benchmark* para o sistema web *open source*. O trabalho trata variações de desempenho em um nível de abstração menor que o presente trabalho, uma vez que apresentam um número gerenciável de contadores de desempenho para serem analisados, mas não indicam as fontes da variação no nível de cenários. Também não apontam *commits* ou tarefas de desenvolvimento responsáveis por introduzir mudanças que potencialmente tenham causado o problema.

Tibermacine & Zernadji [16] propõem uma abordagem para supervisionar a evolução da orquestração de serviços web. A proposta é baseada na análise de documentação de decisões de projeto que afetam atributos de qualidade. Isso permite entender quais atributos de qualidade são conflitantes, facilitando a análise de como evoluções na arquitetura que afetam determinados atributos podem conflitar com outros existentes. A principal diferença entre esse trabalho e a nossa abordagem é que o primeiro considera apenas os conflitos entre atributos de qualidade em uma evolução, mas não realiza nenhuma medição no sistema, uma vez que a abordagem não monitora a execução do sistema e nem analisa seu código. Nossa abordagem permite indicar automaticamente potenciais cenários com conflitos rastreando os locais das anotações de atributos de qualidade.

Nguyen et al [15] propõem a criação e mineração de repositórios de causas de regressão para auxiliar a equipe de desempenho a identificar causas que levaram o sistema a uma regressão de desempenho. O repositório de causas de regressão contém os resultados dos testes de desempenho e das causas de regressões passadas. A abordagem usa técnicas de aprendizado de máquina para determinar causas de novas regressões analisando o repositório. Entretanto, as causas identificadas são limitadas a um conjunto de situações pré-definidas a partir de relatórios de

bugs, que representam ações que, frequentemente, causam regressão de desempenho. Elas incluem, por exemplo, adição de lógica executada com frequência, adição de estratégias de entrada/saída bloqueantes, entre outras. O estudo aplicou a proposta para um sistema comercial e também para o sistema *open source* de *e-commerce* Dell DVD Store (DS2). O trabalho de pesquisa de tais autores não indica que artefatos de código, *commits* ou tarefas foram responsáveis pela regressão, da forma como é contemplada na nossa abordagem.

7. CONCLUSÃO

Este trabalho apresentou um *framework* baseado em metadados que automatiza uma abordagem para revelar degradações de cenários arquiteturais em sistemas web com base em técnicas de análise dinâmica e mineração de repositório de software. A descrição da abordagem enfatizou suas três fases: análise dinâmica, análise de degradação e mineração de repositório. Os módulos do *framework* responsáveis pela implementação de cada uma das fases foram detalhados, destacando os pontos de extensão que o adaptam a um dado ambiente de desenvolvimento. Também foi apresentado um estudo de caso para um sistema web de larga escala de gestão acadêmica. Oportunamente, o estudo apresentou as etapas de instanciação do *framework* para tal sistema e os resultados obtidos de sua aplicação, os quais foram posteriormente discutidos.

É interessante reforçar que o *framework* é extensível para permitir a inclusão de novas anotações de atributos de qualidade que podem ser usadas para rastreabilidade de partes do código fonte do sistema dentro dos cenários executados através de consultas aos modelos de análise dinâmica. Igualmente, pode-se incluir suporte para outros sistemas de controle de versões e gerência de mudanças de acordo com a configuração do ambiente do sistema a ser analisado.

Em estudos mais recentes, o *framework* foi instanciado para dois outros sistemas, sendo que uma das instanciações inclui a implementação do minerador para GitHub. Novos estudos estão sendo conduzidos com sistemas web de larga escala para analisar, além do desempenho, o atributo de qualidade de confiabilidade a partir da análise de *logs* de execução. A precisão do *framework* na detecção das origens das degradações também está sendo analisada nesses novos estudos.

Agradecimentos. Este trabalho foi parcialmente financiado pelo National Institute of Science and Technology for Software Engineering (INES), financiado pelo CNPq, processos 573964/2008-4 e Casadinho/Procad 552645/2011-7, e por SINFO/UFRN.

8. REFERÊNCIAS

[1] Silva, L. and Balasubramaniam, D. 2012. Controlling software architecture erosion: A survey. Journal of Systems and Software. 85, 1 (January 2012), 132-151.

[2] Taylor, R. N., Medvidovic, N. and Dashofy, E. M. 2009. Software Architecture: Foundations, Theory, and Practice. Wiley Publishing.

[3] Clements, P., Kazman, R. and Klein, M. 2002. Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley.

[4] Ali Babar, M. and Gorton, I. 2004. Comparison of Scenario-Based Software Architecture Evaluation Methods. In Proceedings of APSEC'04. Washington, DC, USA, 600-607.

[5] Kazman, R., Abowd, G., Bass, L. and Clements, P. 1996. Scenario-Based Analysis of Software Architecture. IEEE Softw. 13, 6 (November 1996), 47-55.

[6] Roy, B. and Graham, T. C. N. 2008. Methods for Evaluating Software Architecture: A Survey. Technical Report No. 2008-545, School of Computing, Queen's University at Kingston. Ontario, Canada.

[7] Ganesan, D., Lindvall, M., Cleaveland, R., Jetley, R., Jones, P. and Zhang, Y. 2011. Architecture Reconstruction and Analysis of Medical Device Software. In Proceedings of WICSA'11. Washington, DC, USA, 194-203.

[8] Ganesan, D., Keuler, T. and Nishimura, Y. 2009. Architecture compliance checking at run-time. Information and Software Technology. 51, 11 (November 2009), 1586-1600.

[9] Abi-Antoun, M and Aldrich, J. 2009. Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations. SIGPLAN Not. 44, 10 (October 2009), 321-340.

[10] Ciraci, S., Sozer, H. and Tekinerdogan, B. 2012. An Approach for Detecting Inconsistencies between Behavioral Models of the Software Architecture and the Code. In Proceedings of COMPSAC'12. IEEE Computer Society, Washington, DC, USA, 257-266.

[11] Gokhale, S. S. 2007. Architecture-Based Software Reliability Analysis: Overview and Limitations. IEEE Ns On Dependable And Secure Computing. 4, 1 (January 2007), 32-40.

[12] Williams, L. G. and Smith, C. U. 2002. PASASM: a method for the performance assessment of software architectures. In Proceedings of WOSP'02. ACM, New York, USA, 179-189.

[13] Malik, H., Hemmati, H. and Hassan, A. E. 2013. Automatic detection of performance deviations in the load testing of large scale systems. In Proceedings of ICSE'13. IEEE Press, Piscataway, NJ, USA, 1012-1021.

[14] Nistor, A., Jiang, T. and Tan, L. 2013. Discovering, reporting, and fixing performance bugs. In Proceedings of MSR'13. IEEE Press, Piscataway, NJ, USA, 237-246.

[15] Nguyen, T. H. D., Nagappan, M., Hassan, A. E., Nasser, M. and Flora, P. 2014. An industrial case study of automatically identifying performance regression-causes. In Proceedings of MSR'14. ACM, New York, NY, USA, 232-241.

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

[17] SINFO: http://www.info.ufrn.br/wikisistemas/doku.php (June 2014).

[18] Apache JMeter: http://jmeter.apache.org (June 2014).