Detecção de Violações Arquiteturais usando Histórico de Versões

Cristiano Amaral Maffort[1,2], Marco Tulio Valente[2], Mariza A. S. Bigonha[2]

[1]Departamento de Computação, CEFET – MG

[2]Departamento de Ciência da Computação, UFMG

cristiano@decom.cefetmg.br, {mtov,mariza}@dcc.ufmg.br

Resumo. *Este artigo descreve uma metodologia para detecção de violações arquiteturais em um dado produto de software. O objetivo principal é coletar informações históricas sobre como as relações de dependência entre classes que compõem o sistema se comportam no decorrer do tempo de modo a identificar padrões de uso e desuso nessas relações para, então, detectar violações arquiteturais presentes no código-fonte do sistema. O artigo apresenta também resultados da aplicação da metodologia proposta em um sistema de gestão de uma instituição de ensino superior, composto por 1852 classes e interfaces, o qual é mantido em um repositório de controle de versões contendo 4923 versões. Como resultado, a metodologia proposta detectou 92% das divergências arquiteturais existentes nesse sistema (recall), com uma precisão de cerca de 67%.*

Abstract. *This paper proposes an approach to detect architectural violations in object-oriented software products. Basically, the proposed approach collects historical information on the inter-class dependency relations of an existing system in order to identify usage and disuse patterns that can suggest the existence of architectural violations in the source code. To evaluate the proposed approach, the paper also presents results of its application in the main administrative information system of a university, with 1852 classes and interfaces. For this purpose, we have collected information on 4923 revisions stored in this system version control platform. As a result, the proposed approach detected 92% of the existing architectural divergences (recall), with a precision of 67%.*

1. Introdução

A especificação do modelo arquitetural de um software é uma tarefa que, via de regra, antecede as atividades de codificação. Nesta etapa, arquitetos de software definem como será a organização modular do sistema e quais serão as relações de dependência desses módulos e/ou componentes entre si e com sistemas externos, como *frameworks*.

No entanto, durante a evolução de um produto de software podem ser introduzidas anomalias de codificação que não são aderentes ao modelo arquitetural especificado para a aplicação. Essas anomalias são, nesse artigo, classificadas como violações arquiteturais, e representam desvios de implementação em relação à arquitetura planejada para um sistema [1,2].

Na prática, a introdução de violações arquiteturais é bastante comum, devido, principalmente, à falta de conhecimento dos desenvolvedores, impedimentos e/ou dificuldades técnicas, requisitos conflitantes etc [3]. Essas violações tornam mais difíceis as

tarefas de manutenção no software, já que o produto codificado não está aderente à arquitetura planejada e documentada [4].

Sendo assim, este artigo é baseado na observação de que a adição de violações arquiteturais em um produto de software é uma tarefa comum [5] e que parte das violações introduzidas são detectadas e corrigidas em revisões futuras, por meio de atividades de inspeção e/ou manutenção no software. Além disso, observa-se que, seguindo-se boas práticas de desenho de software, classes pertencentes a um mesmo módulo ou componente seguem convenções de programação similares. Por exemplo, suponha um sistema organizado em camadas, o qual inclui uma camada de visão $\mathcal{V}$, uma camada de controle $\mathcal{C}$ e uma camada de persistência $\mathcal{P}$. Nesse sistema, somente $\mathcal{V}$ pode utilizar serviços de $\mathcal{C}$ e somente $\mathcal{C}$ de $\mathcal{P}$. Essa restrição representa um padrão de implementação e qualquer relação de dependência entre classes que viole essa regra está violando a arquitetura. Como outro exemplo, apenas o módulo $\mathcal{P}$ deve possuir relações de dependência com classes pertencentes ao *framework* de persistência utilizado. Entretanto, observando o sistema, verificou-se a existência de dependência entre classes da camada $\mathcal{C}$ com classes do *framework* de persistência, as quais diferem do padrão de implementação frequentemente usado. Além disso, constatou-se que um percentual significativo de tais dependências foi excluído em versões anteriores do sistema. Nesse caso, o fato da relação de dependência ter sido excluída reforça o indício de violação da mesma, e pode ser usado para detectar outras violações semelhantes no código.

Para tratar essas questões, propõe-se neste artigo uma metodologia para detecção de violações arquiteturais em produtos de software. Basicamente, a solução proposta analisa os padrões de uso e desuso (inclusão e exclusão) entre as relações de dependência do sistema no decorrer do tempo, por meio das versões armazenadas no repositório de controle de versões. O objetivo principal da solução proposta é identificar violações arquiteturais a partir tanto dos padrões de similaridade entre as relações de dependência quanto dos padrões de modificações realizadas no histórico de versões.

Com objetivo de avaliar a abordagem proposta, descreve-se também sua aplicação em um sistema real de médio porte utilizado por uma instituição de ensino superior para gestão administrativa. Como resultado, foram identificados 217 indícios de violações nas relações de dependência desse sistema, sendo confirmadas 116 violações, o que implica em uma precisão de 53,45%. Adicionalmente, a metodologia proposta detectou 73 das 79 divergências arquiteturais existentes nesse sistema (*recall*), implicando numa precisão de 92,41%. Além disso, a metodologia de detecção é realizada estaticamente e de forma não-invasiva, de modo que não impacta as atividades normais de programação do sistema.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta uma visão geral da abordagem proposta. As Seções 3 e 4 descrevem as heurísticas para detecção de ausências e divergências, respectivamente. A Seção 5 apresenta uma avaliação de aplicação da abordagem em um estudo de caso envolvendo um sistema real. A Seção 6 descreve alguns trabalhos relacionados e a Seção 7 apresenta as conclusões.

2. Estratégias para Detecção de Violações Arquiteturais

Este artigo propõe uma estratégia para detecção de violações na arquitetura de um software. A abordagem proposta baseia-se tanto nas relações de dependência entre as classes que compõem um sistema quanto no histórico de manutenções realizadas nessas

relações. Tais informações históricas são recuperadas a partir das revisões do sistema mantidas em um repositório de controle de versões. A partir das informações extraídas, um arquiteto de software deve agrupar os módulos do sistema em componentes, as quais constituem um parâmetro de entrada das estratégias de detecção de violações. Por fim, as informações arquiteturais e as relações de dependência extraídas são analisadas por heurísticas para inferir violações arquiteturais, conforme ilustrado na Figura 1.

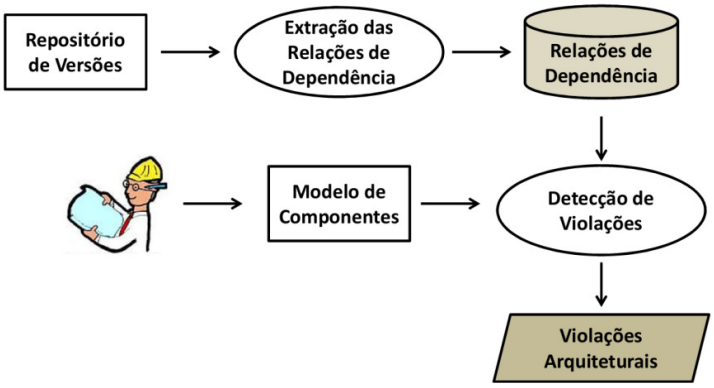

Figura 1: Abordagem proposta

A abordagem proposta considera que os módulos do sistema estejam organizados em estruturas de maior granularidade, chamadas componentes. Por exemplo, em Java, um módulo corresponde a um pacote e um componente a um conjunto de pacotes relacionados.

2.1. Notações

As seguintes notações são utilizadas na definição da abordagem proposta:

- $V = \{v_1, v_2, \ldots, v_N\}$ é o conjunto de todas as versões de um sistema, as quais são mantidas em um repositório de controle de versões.
- $C = \{c_1, c_2, \ldots, c_N\}$ é o conjunto de todas as classes do sistema. A função $first(c)$ retorna a versão na qual a classe c foi inserida no repositório pela primeira vez.
- $M = \{m_1, m_2, \ldots, m_N\}$ é o conjunto de todos os módulos do sistema. A função $mod(c)$ retorna o módulo ao qual uma determinada classe c pertence.
- $COMP = \{l_1, l_2, \ldots, l_N\}$ é o conjunto de todos os componentes do sistema. A função $comp(c)$ informa a qual componente l uma determinada classe c pertence.
- $DRG_v = \langle C, R \rangle$ é um grafo direcionado contendo as relações de dependência entre as classes de uma versão v do sistema. Nesse grafo, os vértices denotam classes e as arestas indicam relações de dependência entre duas classes. Para simplificar, a notação $depends(c_1, c_2, v)$ indica que a classe c_1 depende da classe c_2 na versão v do sistema.
- H é o identificador da versão mais recente (atual) dos artefatos no repositório.

2.2. Extração do Grafo DRG

Para construção do DRG, o algoritmo de extração das relações de dependência recebe como entrada uma referência para o repositório de controle de versões que hospeda o código fonte do sistema analisado. Em seguida, são percorridas todas as versões desse

sistema e, para cada versão, recupera-se o conjunto de todas as relações de dependência sintática R entre as classes do sistema. Particularmente, considera-se que uma relação de dependência sintática denota uma relação de uso entre duas classes. Ou seja, a notação $depends(c_1, c_2, v)$ denota que a classe c_1, na versão v, utiliza de algum modo a classe c_2 — seja como um atributo, uma variável, argumento de um método, anotação, etc.

2.3. Heurísticas

As heurísticas propostas neste artigo destinam-se a detectar dois tipos de violações arquiteturais: (a) dependências que existem no modelo arquitetural mas não existem no código fonte e (b) dependências que existem no código fonte mas não existem no modelo arquitetural. Estes grupos são chamados de *ausências* e *divergências*, respectivamente [6,1].

3. Heurísticas para Detecção de Ausências

As ausências são violações que ocorrem quando uma classe não utiliza um recurso ou serviço provido por outra classe, embora tal utilização esteja prescrita no modelo arquitetural do sistema. Esta anomalia ocorre, principalmente, quando o desenvolvedor negligencia o uso de uma classe especificada no modelo, resultando, frequentemente, em implementação de código redundante.

A identificação das ausências é realizada em dois passos: o primeiro passo consiste em identificar as relações de dependência que frequentemente ocorrem nas classes, mas que não ocorrem em um pequeno conjunto de classes daquele módulo e/ou componente. O segundo passo utiliza informações históricas das manutenções realizadas no sistema, valendo-se do repositório de controle de versões. Nesse caso, procura-se identificar padrões no modo como as relações de dependência são estabelecidas no decorrer do ciclo de desenvolvimento do sistema. Classes que divergem desses padrões são classificadas como indícios de violação.

Esses dois passos são detalhados a seguir:

Passo #1: este passo seleciona classes de um módulo m que não possuem relação de dependência com uma classe particular c, enquanto a maioria das classes de m possui relação de dependência com c. Formalmente, esse passo pode ser expresso como descrito a seguir. Considere $D(m, c)$ como o conjunto das classes do módulo m que possuem relação de dependência com a classe c na versão mais recente do sistema, como definido pela Equação 1:

$$D(m, c) = \{x \in C_{(H)} \mid depends(x, c, H) \wedge mod(x) = m\} \quad (1)$$

Seja $C(m)$ o conjunto de classes do módulo m na versão atual do sistema, como definido na Equação 2:

$$C(m) = \{x \in C_{(H)} \mid mod(x) = m\} \quad (2)$$

Assim, a razão entre o número de classes de um módulo m que possuem relação de dependência com uma determinada classe c pelo número total de classes deste mesmo módulo é definida pela Equação 3.

$$DRU(m, c) = \frac{|D(m, c)|}{|C(m)|} \quad (3)$$

Como exemplo, considere um sistema cuja arquitetura seja baseada no padrão *Model-View-Controller* (MVC). Além disso, considere também que o sistema utiliza JPA como *framework* de persistência, o qual estabelece que classes persistentes da camada $Model$ devem usar a anotação `Entity`. Assim, de acordo com esse passo, a identificação desta violação baseia-se em quantificar o número de classes do módulo m que possuem relação de dependência com `Entity` por meio da função $D(m, Entity)$. Em seguida, obtém-se o número de classes do módulo m via $C(m)$. Por fim, calcula-se a porcentagem de classes do módulo m que dependem da classe `Entity`, conforme definido pela Equação 3. Segundo essa estratégia para detecção de violações, quanto maior for $DRU(m, Entity)$, maior o indício de que as classes de m que não se relacionam com `Entity` estão violando a arquitetura do sistema.

Portanto, para inferir os indícios de ausência em um módulo m, deve-se, primeiramente, selecionar as classes das quais m depende na versão atual do sistema, expresso da seguinte forma:

$$DC(m) = \{c \in C \mid x \in C, depends(x, c, H) \wedge mod(x) = m\} \quad (4)$$

Em seguida, considera-se que uma classe x do módulo m possui uma ausência em relação a uma determinada classe c quando x não possui relação de dependência com c, mas existe pelo menos um percentual de classes k_{mru} localizadas em m que dependem de c, o que pode ser formalmente representado da seguinte forma:

$$AMV(m) = \left\{(x, c) \;\middle|\; \begin{matrix} x \in C \wedge c \in DC(m) \wedge mod(x) = m \wedge \\ \neg depends(x, c, H) \wedge DRU(m, c) \geq k_{mru} \end{matrix}\right\} \quad (5)$$

Ao limitar o escopo de detecção da ausência a um módulo, a heurística proposta torna possível identificar violações que ocorrem em um módulo mesmo que este possua algumas classes com um padrão de dependências atípico. No entanto, caso todas as classes do mesmo módulo possuam a mesma relação ausente, então o agrupamento por módulo não identificará as violações segundo a estratégia proposta.

Para contornar a deficiência apresentada, propõe-se uma ampliação do escopo de agrupamento das classes por módulo para componente e a aplicação de um conjunto de equações idêntico àquele utilizado para o agrupamento por módulo. O agrupamento por componente permite identificar ausências que ocorrem na maioria das classes de m mas que não ocorrem na maioria das classes do componente que contém m. Esta regra pode ser expressa pela Equação 3, substituindo-se o escopo de classes de módulo para componente, de modo que $DRU(l, c)$ passe a representar a porcentagem de classes do componente l que possuem relação de dependência com uma determinada classe c.

Para ilustrar esse cenário, considere o sistema descrito no exemplo anterior. Segundo a especificação JPA, classes persistentes da camada $Model$, anotadas com `Entity`, devem possuir um atributo de tipo inteiro anotado com `Id`. Suponha que um desenvolvedor tenha implementado todas as classes de um mesmo módulo m negligenciando essa convenção de programação. Nesse caso, a estratégia de agrupamento por módulo não identifica a ausência dessa relação, pois essa ausência é comum a todas as classes do módulo. Entretanto, ao analisar as classes desse módulo em conjunto com classes de outros módulos pertencentes ao mesmo componente ($Model$), pode-se constatar que a maioria das classes persistentes da camada $Model$ dependem de `Id`. Como

as classes de m não possuem essa relação de dependência, todas elas representam indícios de violação.

De modo análogo à metodologia de detecção de ausência por módulo, a detecção de indícios de ausências em um componente l consiste em: a) selecionar as classes das quais l depende ($DC(l)$); b) determinar quais classes do componente l não possuem relação de dependência com uma classe c, quando alguma classe de l possui essa relação, assim expresso:

$$ALV(l) = \left\{(x,c) \,\middle|\, \begin{matrix} x \in C \wedge c \in DC(l) \wedge comp(x) = l \wedge \\ \neg depends(x,c,H) \wedge DRU(l,c) \geq k_{lru} \end{matrix} \right\} \quad (6)$$

Em outras palavras, ausências em um componente l são detectadas quando classes de l não dependem de uma classe c quando um percentual alto — acima de uma constante k_{lru} — das classes desse componente dependem de c.

Passo #2: no Passo #1, somente a versão mais recente do sistema (H) é avaliada. O Passo #2 considera informações históricas de manutenções realizadas no sistema, por meio de uma análise no repositório de controle de versões, de modo a melhorar a precisão da heurística de detecção de ausências.

Nesta etapa, as violações são identificadas quando classes de um módulo m, ou componente l, são inseridas no repositório pela primeira vez, sem relação de dependência com uma classe particular c, conforme formalizado na Equação 7.

$$CWD(m,c) = \{x \in C \mid mod(x) = m \wedge \neg depends(x,c,first(x))\} \quad (7)$$

No entanto, em versões seguintes, adiciona-se uma relação de dependência com c, conforme apresentado pela Equação 8.

$$ADL(m,c) = \{x \mid x \in CWD(m,c) \wedge depends(x,c,H)\} \quad (8)$$

Neste caso, a métrica utilizada é representada pela razão entre o número de classes de m que adicionaram uma relação de dependência com a classe c e o número de classes de m que "nasceram" sem essa relação, assim expresso:

$$DHR(m,c) = \frac{|ADL(m,c)|}{|CWD(m,c)|} \quad (9)$$

Por fim, considera-se que representam indícios de violação as classes de m que não possuem relação de dependência com c quando a maioria das classes que "nasceram" sem essa relação, segundo um percentual k_{mhr}, estabeleceram essa relação no futuro, conforme formalizado a seguir:

$$AMHV(m,c) = \left\{(x,c) \,\middle|\, \begin{matrix} x,c \in C \wedge mod(x) = m \wedge \neg depends(x,c,H) \wedge \\ DHR(m,c) \geq k_{mhr} \end{matrix} \right\} \quad (10)$$

Como exemplo, considere o sistema descrito no Passo #1. Suponha que ao analisar o histórico das relações de dependência das classes do módulo m tenha-se verificado que um percentual expressivo de tais classes tenha adicionado relação de dependência com `Entity` a partir da segunda revisão das classes no repositório. Nesse caso, consi-

dera-se que, além da maioria das classes persistentes da camada $Model$ possuírem relação de dependência com `Entity`, a adição dessa relação posteriormente reforça o indício de violação detectado no Passo #1.

De modo análogo ao Passo #1, essa estratégia de detecção também pode se valer dos benefícios de agrupar as classes por componentes devido à baixa granularidade do agrupamento por módulo. Nesta etapa, essa dependência é ainda mais evidente, pois alguns módulos possuem poucas classes, as quais podem ter sofrido pequenas modificações. Assim, o agrupamento por componente torna possível contornar particularidades de um módulo.

4. Heurísticas para Detecção de Divergências

Violações por divergência ocorrem quando existe uma relação de dependência no código fonte que não é prevista no modelo arquitetural do sistema. Para detectar esses desvios, a heurística proposta utiliza o histórico de versões para identificar padrões de modificações nas relações de dependência. Para detecção de divergências são utilizadas três heurísticas: a Heurística #1 identifica classes que possuem relação de dependência com classes raramente utilizadas no sistema e que, quando usadas, frequentemente excluem essa dependência; a Heurística #2 seleciona classes de um componente l que criam relações de dependência com classes de um módulo m, o qual é raramente utilizado em l mas é frequentemente usado em outros componentes do sistema; e a Heurística #3 detecta dependências circulares entre componentes.

4.1. Heurística #1

Frequentemente, durante o ciclo de desenvolvimento de um sistema, desenvolvedores adicionam relações de dependência que violam o modelo arquitetural. Em sistemas maduros, que contam com um repositório com um número expressivo de versões, tais violações são frequentemente corrigidas em atividades de inspeção e/ou manutenção.

Assim, esta heurística seleciona classes de um módulo m que possuem uma quantidade pequena de relações de dependência com uma determinada classe c. Além disso, observando o histórico das relações de dependência das classes do módulo m, verifica-se que as relações de dependência entre classes de m com a classe c foram frequentemente excluídas.

Mais precisamente, essa heurística pode ser expressa como descrito a seguir. Considere $DA(m, c)$ como o conjunto das classes do módulo m que, em alguma versão do sistema, estabeleceram uma relação de dependência com uma determinada classe c, assim definido:

$$DA(m, c) = \{x \in C \mid \exists v \in V \wedge mod(x) = m \wedge depends(x, c, v)\} \quad (11)$$

Em seguida, dentre as classes que estabeleceram relação de dependência com c, selecionam-se aquelas que removeram essa relação e que, portanto, não a possuem na versão corrente do sistema, como definido pela Equação 12:

$$DR(m, c) = \{x \in DA(m, c) \mid mod(x) = m \wedge \neg depends(x, c, H)\} \quad (12)$$

Adicionalmente, calcula-se a razão entre o número de classes do módulo m que removeram uma relação de dependência com uma determinada classe c ($DR(m,c)$) e o número de classes que, em alguma versão do sistema, estabeleceram essa relação ($DA(m,c)$), desta forma:

$$DRR(m,c) = \frac{|DR(m,c)|}{|DA(m,c)|} \quad (13)$$

Por fim, a detecção de violações por meio dessa heurística consiste em selecionar classes que possuem relação de dependência com classes que raramente são utilizadas. Além disso, mesmo quando utilizadas, essa dependência é frequentemente removida. A Equação 14 define formalmente esta heurística.

$$DVD(c) = \left\{(x,c) \;\middle|\; \begin{matrix} DRU(m,c,H) \leq k_m \;\wedge\; DRU(l,c,H) \leq k_l \;\wedge \\ DRR(l,c) \geq k_r \wedge\; DA(l,c) \geq k_a \wedge\; depends(x,c,H) \end{matrix}\right\} \quad (14)$$

Nessa equação, as constantes k_m e k_l denotam a utilização de uma classe, em nível de módulo e componente, respectivamente. Adicionalmente, a constante k_r representa um percentual histórico de remoção de uma dependência. Além disso, a constante k_a é utilizada para estabelecer uma condição de suporte para os parâmetros anteriores.

Conforme expresso na Equação 14, esta heurística depende do histórico de versões do sistema, de modo que quanto mais maduro for o código fonte mantido no repositório, maior será a precisão da heurística. Isso acontece porque ela requer que anomalias de implementação relacionadas a violações de modularidade tenham sido identificadas e corrigidas em versões passadas do sistema.

Heurística #2

Esta heurística seleciona classes de um componente l que possuem relações de dependência com classes de um módulo m, quando as seguintes condições são satisfeitas: (a) relações de dependência com classes de m raramente ocorrem no componente l; (b) tais relações ocorrem mais comumente em outros componentes do sistema; (c) de acordo com o histórico de revisões do sistema, essas dependências foram frequentemente removidas.

Essa regra pode ser formalizada conforme descrito a seguir:

1) Calcula-se o percentual de utilização das classes de um módulo m no restante do sistema. Para tanto, deve-se obter inicialmente as classes que dependem de m:

$$DM(m) = \{x \in C \mid \exists c \in C \wedge mod(c) = m \wedge depends(x,c,H)\} \quad (15)$$

Em seguida, selecionam-se as classes de um componente l que dependem de alguma classe de m:

$$DLM(l,m) = \{x \in DM(m) \mid comp(x) = l\} \quad (16)$$

Posteriormente, calcula-se o seguinte percentual de utilização:

$$DLMR(l,m) = \frac{|DLM(l,m)|}{|DM(m)|} \quad (17)$$

2) Calcula-se o índice de exclusão das relações de dependência entre as classes do componente l que dependem de classes do módulo m. Para tanto, seleciona-se as relações de dependência entre l e m estabelecidas em qualquer versão do sistema:

$$DALM(l,m) = \left\{(x,c) \left| \begin{matrix} \exists x,c \in C \ ^\wedge \ \exists v \in V \ ^\wedge \ comp(x) = l \ ^\wedge \\ mod(c) = m \ ^\wedge \ depends(x,c,v) \end{matrix} \right.\right\} \quad (18)$$

Em seguida, são separadas as relações de dependência obtidas pela função $DALM(l,m)$ que foram removidas e não existem na versão atual do sistema:

$$DRLM(l,m) = \{(x,c) \in DALM(l,m) \mid \neg depends(x,c,H)\} \quad (19)$$

Usando-se esse valor, calcula-se o percentual de remoção das relações de dependência das classes do componente l que dependem de classes do módulo m:

$$DLMRR(l,m) = \frac{|DRLM(l,m)|}{|DALM(l,m)|} \quad (20)$$

3) Por fim, os indícios de divergência são detectados quando um percentual baixo de classes de um componente l depende de classes do módulo m. Além disso, tais relações são frequentemente removidas. A Equação 21 especifica formalmente essa regra:

$$DLMV(l,m) = \left\{(x,c) \left| \begin{matrix} \exists x,c \ \in C \^ depends(x,c,H) \^ comp(x) = l \^ mod(c) = m \^ \\ DLMR(l,m) \leq k_{lru} \^ \ DLMRR(l,m) \geq k_{drlm} \end{matrix} \right.\right\} \quad (21)$$

Nessa equação, k_{lru} é uma constante usada para denotar o percentual de classes de l que dependem de classes de m. Já a constante k_{drlm} denota o percentual de remoção no histórico de versões do sistema.

Como exemplo, considere novamente o sistema descrito na Seção 3. Suponha que ao analisar as relações de dependência das classes da camada $Controller$ tenha-se verificado que um percentual pequeno de classes que estabeleceu alguma relação de dependência com classes do pacote `javax.persistence`. Considere também que ao observar o histórico de versões do sistema verificou-se que classes da camada $Controller$ frequentemente excluíram relações de dependência com classes do módulo `javax.persistence`. Além disso, na versão atual do sistema, dentre as classes que possuem relações de dependência com classes do módulo `javax.persistence`, a grande maioria pertence a uma terceira camada (por exemplo, à camada $Model$). Portanto, dadas essas condições, as classes da camada $Controller$ que possuem relações de dependência com classes do módulo `javax.persistence` representam indícios de divergência arquitetural.

4.3. Heurística #3

Via de regra, deve-se evitar a criação de dependências circulares entre os componentes de um sistema. Assim, neste artigo, considera-se que a ocorrência de dependências circulares é uma exceção à regra e representa uma anomalia arquitetural, classificada como uma divergência, pois consiste da utilização inadequada de uma classe.

Para identificação das violações a partir desta heurística, considere uma relação de dependência entre duas classes c_1 e c_2, representada no grafo por uma aresta direcional $e(c_1, c_2)$. A detecção de dependências circulares requer que, inicialmente, identifique-se o sentido de comunicação entre os componentes. Para tanto, define-se como sentido correto aquele que ocorre mais frequentemente dentre as relações de dependência entre dois componentes, conforme descrito a seguir. Primeiro, deve-se obter o percentual de relações de dependência do componente l_1 em relação ao componente l_2 da seguinte forma:

$$LGR(l_1, l_2) = \frac{|e(l_1, l_2)|}{|e(l_1, l_2)| + |e(l_2, l_1)|} \tag{22}$$

onde $|e(l_1, l_2)|$ denota o número de dependências do componentes l_1 em relação ao componente l_2. Em seguida, deve-se selecionar as relações de dependência que representam indícios de violação arquitetural, conforme formalizado a seguir:

$$CDV(l_1, l_2) = \left\{ (c_1, c_2) \middle| \begin{matrix} c_1, c_2 \in C \wedge depends(c_1, c_2, H) \wedge comp(c_1) = l_1 \wedge \\ comp(c_2) = l_2 \wedge LGR(l_1, l_2) < k_{cdr} \end{matrix} \right\} \tag{23}$$

O cenário ideal consiste em não haver dependência circular entre componentes, quando todas as arestas entre dois componentes l_1 e l_2 são arestas de saída em um dos componentes e arestas de entrada no outro — indicando assim que apenas um componente depende do outro. Quando ocorre dependência circular, cada componente possui tanto arestas de entrada quanto arestas de saída em relação ao outro — indicando que ambos os componentes são mutuamente dependentes. Segundo essa heurística, constituem indícios de violação arquitetural as classes do componente l_1 que, na versão atual do sistema, possuem relação de dependência com classes do componente l_2 quando o percentual de arestas de saída de l_1 é menor que um fator k_{cdr}. Como exemplo, considere $LGR(l_1, l_2) = 0.3$ e $k_{cdr} = 0.5$. Nesse caso, 30% das arestas de l_1, em relação a l_2, são arestas de saída e 70% são arestas de entrada. Já a constante k_{cdr} denota o percentual máximo de arestas de saída de um componente. Nesse exemplo, considera-se que caso o número de arestas de saída seja inferior a 50%, então as relações de dependência representadas pelas arestas de saída do componente l_1 indicam violações arquiteturais, pois as arestas de entrada predominam em relação às arestas de saída.

5. Avaliação

A fim de avaliar a abordagem para detecção de violações arquiteturais proposta neste artigo, foi realizado um estudo de caso com um sistema de gestão de uma instituição de ensino superior, os quais chamaremos por questões de confidencialidade, deste ponto em diante, de SGA e IES respectivamente. Esse sistema é especificado e desenvolvido por uma equipe mantida pela própria IES. Atualmente, o sistema SGA automatiza atividades de natureza predominantemente administrativas, como gerenciamento de receitas/despesas, gestão de recursos materiais, recursos humanos, dentre outros. O desenvolvimento do sistema SGA teve início março de 2009 e manteve sua arquitetura, bem como *frameworks* utilizados e plataforma de desenvolvimento, inalterados desde então. A Figura 2 ilustra o modelo arquitetural do sistema SGA.

O sistema SGA é organizado segundo uma arquitetura MVC. A camada $Model$ é organizada em dois módulos principais, um de domínio e um de serviço. O módulo de domínio contém os objetos de negócio (DTOs —*Data Transfer Objects*), como Funcio-

nario, Aluno, Documento etc. Esse módulo contém também os objetos de persistência dos dados junto ao SGBD (DAOs — *Data Access Objects*), os quais valem-se da especificação JPA para realizar operações de persistência. O módulo de serviço encapsula o processo e as regras de negócio (BOs — *Business Objects*), manipulando os estados dos objetos de negócio segundo o *workflow* da IES e interagindo com os objetos DAO para persistência dos objetos manipulados. Cada serviço é provido por uma interface local, que atua como fachada, e um componente que implementa essa interface, os quais seguem as convenções EJB (*Enterprise JavaBeans*).

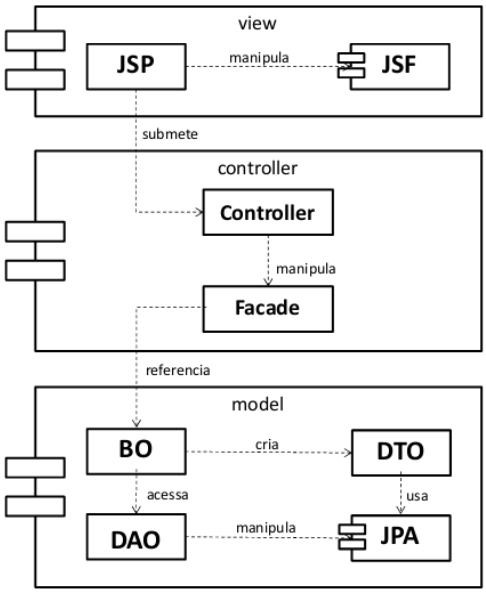

Figura 2: Arquitetura do Sistema SGA

A camada *Controller* fornece uma ponte de ligação da interface com o usuário, monitorando, transferindo e adaptando as entradas fornecidas por ele. Essa camada é constituída, basicamente, por uma interface de fachada e uma implementação que realiza o acesso à camada *Model*, os quais também seguem a especificação EJB.

A camada *View* é implementada em JSP (*Java Server Pages*) e utiliza componentes gráficos e de interação do *framework* JSF (*JavaServer Faces*). Essa camada recebe as solicitações do usuário e as encaminha para a camada *Controller*. Em resumo, o sistema SGA utiliza uma arquitetura baseada em diversos padrões prescritos pela plataforma JEE (*Java Enterprise Edition*), os quais são largamente utilizados atualmente para desenvolvimento de sistemas.

5.1. Coleta dos Dados

A extração das relações de dependência requer o entendimento da arquitetura do sistema, principalmente para inferir a hierarquia entre os componentes. O sistema avaliado possuía, em setembro de 2011, 4923 revisões. A revisão mais recente do sistema possuía nesta época 1852 classes e interfaces, totalizando cerca de 127 KLOC.

Para extração das relações de dependência foram realizadas as seguintes atividades:

1. *Checkout* do sistema no repositório de controle de versões.

2. Execução da ferramenta VerveineJ para extração das relações de dependência entre as classes do sistema. A ferramenta VerveineJ é parte da plataforma Moose para análise e visualização de programas[1]. Basicamente, essa ferramenta gera um meta-modelo de um sistema alvo, o qual inclui informações sobre dependências entre classes.
3. Persistência das informações em um banco de dados.

5.2. Detecção de Ausências

A metodologia descrita na Seção 3 foi usada para detecção de ausências. Particularmente, foram adotados os seguintes parâmetros para essa heurística:

Equação	Parâmetro	Valor
5	k_{mru}	0.9
6	k_{lru}	0.9
10	k_{mhr}	0.6
10	k_{lhr}	0.6

Tabela 1: Parâmetros utilizados na heurística para detecção de ausências.

De forma objetiva, foi considerado indício de violação as classes do módulo m ou componente l que não possuíam relação de dependência com uma determinada classe c quando pelo menos 90% das classes de m ou l dependem de c. Além disso, foram consideradas violações apenas as classes que dependem de c quando, também, pelo menos 60% das classes de m ou l que "nasceram" sem relação de dependência com c adicionaram essa relação em versões seguintes.

5.3. Detecção de Divergências

Descreve-se a seguir como as heurísticas para detecção de divergências, apresentadas na Seção 4, foram aplicadas no Sistema SGA.

Detecção por meio da Heurística #1: Esta heurística seleciona classes que possuem relação de dependência com uma classe c que raramente é utilizada (menos de 40% das classes de um módulo e menos de 5% das classes de um componente). Além disso, das ocasiões em que as classes de l adicionaram alguma relação de dependência com c, em pelo menos 60% a dependência foi removida. Por fim, para estabelecer uma condição de suporte para os parâmetros anteriores, considera-se como violação quando pelo menos cinco classes de l, em algum momento, adicionaram relação de dependência com a classe c.

Estes valores foram usados como parâmetro na Equação 14, conforme detalhado na Tabela 2.

Detecção por meio da Heurística #2: Esta heurística seleciona as classes de um componente l que estabeleceram relação de dependência com uma classe qualquer de um determinado módulo m quando menos de 5% das classes de l dependem de classes desse módulo. Além disso, em 60% das ocasiões em que classes de l adicionaram tal relação, ela posteriormente foi removida. Para tanto, a Equação 21 utiliza os seguintes fatores: $k_{lru} = 0.05$ e $k_{drlm} = 0.6$.

[1] http://www.moosetechnology.org/

Parâmetro	Valor
k_m	0.4
k_l	0.05
k_r	0.6
k_a	5

Tabela 2: Parâmetros utilizados na Heurística #1 para detecção de divergências.

5.4. Análise dos Resultados

Após a aplicação das heurísticas descritas nas Seções 3 e 4, utilizando os parâmetros definidos nas Seções 5.2 e 5.3, todos os indícios de violação selecionados foram documentados e contabilizados. Em seguida, com o auxílio de um arquiteto pertencente à equipe de desenvolvimento do sistema SGA, inspecionou-se cada indício de violação de forma a inferir se o mesmo consistia em uma violação real (verdadeiro-positivo) ou não (falso-positivo).

Em relação aos indícios de ausência foram detectados 108 indícios de violação. Desses, após inspeção cuidadosa, foram confirmados 43 violações reais, representando 39,8% de precisão em relação aos indícios detectados.

A Tabela 3 apresenta as divergências detectadas. Conforme pode ser observado, a Heurística #1 detectou 52 indícios de violação, dos quais 16 foram confirmados. Tanto a Heurística #2 quanto a Heurística #3 apresentaram precisão de 100% na detecção de violações. A última coluna agrupa os resultados pois uma mesma relação de dependência pode ser detectada por mais de uma heurística. Após o agrupamento, pode-se verificar uma precisão de 66,97% na detecção de divergências.

	H1	H2	H3	H1 ∪ H2 ∪ H3
Indícios (I)	52	9	50	109
Verdadeiro-positivos (VP)	16	9	50	73
Falso-positivos (FP)	36	0	0	36
Precisão (VP/I)	30,77%	100,00%	100,00%	66,97%

Tabela 3: Divergências detectadas no Sistema SGA.

Para concluir a avaliação dos resultados e com base no julgamento realizado pelo arquiteto do sistema SGA, foram manualmente inspecionadas 10291 relações de dependência da versão mais recente do sistema no repositório de forma a identificar violações não detectadas pela metodologia (isto é, para avaliar o *recall* ou cobertura da solução proposta). Durante essa inspeção, que demandou cerca de 12 horas, inicialmente filtrou-se as dependências recorrentes em cada componente e que não violam a arquitetura, de modo a manter as relações que, efetivamente, representam violações. Após essa inspeção, verificou-se a existência de 79 divergências, das quais 73 foram detectadas pela abordagem proposta, como apresentado na Tabela 3. Portanto, a metodologia apresentada neste artigo foi capaz de detectar 92,41% das divergências existentes no sistema avaliado.

6. Trabalhos Relacionados

Sarkar et al. [4] conduziram um estudo com o objetivo de descobrir o modelo de organização em camadas de sistemas de software. O modelo arquitetural gerado foi utilizado para detectar violações arquiteturais por meio das relações de dependência entre módu-

los, de acordo com as suas respectivas camadas. A abordagem proposta por eles apenas detectou chamadas que violam a estrutura hierárquica das camadas. Por outro lado, no presente artigo, apresentou-se também um conjunto de heurísticas para: (a) detectar relações ausentes entre duas camadas; (b) detectar relações divergentes entre componentes, os quais não necessariamente precisam seguir uma estrutura hierárquica.

Terra e Valente propuseram uma linguagem declarativa para restrição de dependência que verifica estaticamente a arquitetura de um software em relação às restrições estabelecidas por um arquiteto [5]. A abordagem utilizada requer que um arquiteto defina as restrições, sendo que uma ferramenta incluída na solução verifica apenas o que foi prescrito pelo arquiteto. De maneira geral, a abordagem proposta neste artigo tem por objetivo detectar ausências e divergências, de modo semelhante ao trabalho de Terra e Valente. No entanto, na abordagem proposta o arquiteto não precisa especificar manualmente as restrições arquiteturais, o que invariavelmente tende a se revelar uma tarefa tediosa e sujeita a erros.

Zhenmin e Zhou apresentam uma ferramenta, chamada PR-Miner — baseada na técnica de mineração de dados chamada *frequent itemset mining* — para extração automática de regras de codificação [7]. A abordagem proposta utiliza um formalismo de extração de relações de dependência entre funções que é fortemente dependente de linguagens procedurais. A estratégia proposta para detecção de violações considera apenas os fluxos das chamadas de funções, independentemente do contexto modular e/ou arquitetural onde essas chamadas ocorreram. Por outro lado, a abordagem apresentada no presente artigo é centrada na detecção de violações arquiteturais. É interessante notar, no entanto, que os valores de precisão apresentados pelo PR-Miner foram, de modo geral, inferiores àqueles reportados na Seção 5. Por exemplo, considerando apenas os 60 *warnings* de maior prioridade levantados durante a análise do sistema Linux, apenas 16 se revelaram de fato erros de programação (*bugs*).

Por fim, Mileva et al. conduziram uma análise nos padrões de evolução entre duas versões de um sistema de modo a detectar modificações pendentes no código [8]. A partir das modificações pendentes, a abordagem proposta recomenda, com base nos padrões de uso identificados, modos de correção das pendências. De modo semelhante ao trabalho realizado por Zhenmin e Zhou, a abordagem utilizada no trabalho de Mileva e colegas é fortemente vinculada a conceitos de linguagens procedurais, desconsiderando relações de uso típicos de linguagens orientadas por objetos, como por exemplo relações de herança. Por outro lado, no presente artigo, considera-se, além dos formalismos presentes em linguagens orientadas por objetos, todo o histórico de modificações realizadas em um sistema. Apesar de tais diferenças de foco e linguagens, é importante destacar que a abordagem descrita neste artigo, de modo geral, revelou-se capaz de descobrir um maior número de violações. Por exemplo, a abordagem de Mileva e Zeller foi capaz de descobrir apenas seis violações em um estudo de caso envolvendo as plataformas Eclipse 1.0 e 2.0.

7. Conclusões

Durante o ciclo de desenvolvimento de um software, é comum que a equipe de desenvolvimento adote estratégias de codificação que não são consoantes com o modelo arquitetural planejado para o sistema. Neste artigo, procurou-se detectar violações que

frequentemente ocorrem durante atividades de codificação de um sistema. Dentre essas violações, procurou-se identificar desvios de implementação no código-fonte, com base nos padrões de uso, por meio das relações de dependência entre classes e com base no histórico de manutenções realizadas no sistema. Foram consideradas tanto relações prescritas no modelo arquitetural, mas ausentes no código-fonte, quanto relações presentes no código mas que não estão previstas no modelo arquitetural.

A abordagem proposta foi avaliada em um sistema real de grande porte, tendo sido capaz de detectar 39,81% das violações por ausência e 66,97% das violações por divergência. Além disso, a abordagem identificou 92,41% das divergências presentes no código fonte.

Como trabalho futuro, pretende-se ampliar o estudo realizado, possivelmente acrescentando novas heurísticas de detecção de violações utilizando técnicas de mineração de dados e análise formal de conceitos de modo a diminuir o número de falsos positivos. Por fim, pretende-se realizar novos estudos de caso, envolvendo sistemas que utilizem padrões arquiteturais diferentes daqueles do sistema SGA.

Referências

[1] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor Mendonca. *Static architecture-conformance checking: An illustrative overview. IEEE Software*, 27:82–89, 2010.

[2] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17:40–52, 1992.

[3] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. *In 6th Working IEEE/IFIP Conference on Software Architecture*, 07, pages 12–, 2007.

[4] Santonu Sarkar, Girish Maskeri, and Shubha Ramachandran. Discovery of architectural layers and measurement of layering violations in source code. *Journal of Systems and Software*, 82:1891–1905, 2009.

[5] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software Practice and Experience*, 39:1073–1094, 2009.

[6] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27:364–380, 2001.

[7] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *In 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 306–315, 2005.

[8] Yana Momchilova Mileva, Andrzej Wasylkowski, and Andreas Zeller. Mining evolution of object usage. *In 25th European Conference on Object-oriented Programming*, pages 105–129, 2011.