Processo de Conformidade Arquitetural em Integração Contínua

Arthur F. Pinto, Ricardo Terra

Departamento de Ciência da Computação,
Universidade Federal de Lavras (UFLA), Brasil

arthurfp@sistemas.ufla.br, terra@dcc.ufla.br

Abstract. *As software evolves, developers usually introduce deviations from the planned architecture, due to unawareness, conflicting requirements, technical difficulties, deadlines, etc. Although architectural compliance processes identify architectural violations, (i) these tools are underused and (ii) detected violations are rarely corrected. To address these shortcomings, this paper proposes a solution of architectural compliance into continuous integration. Thus, the architectural compliance process is triggered by every code integration, and when no violations are detected, the code is integrated into the repository. In addition, this paper presents the ArchCI tool—that implements the proposed solution using DCL as underlying conformance technique and Jenkins as the CI server—and a controlled evaluation that demonstrates the applicability of the solution.*

Resumo. *No decorrer de um projeto de software, desenvolvedores normalmente introduzem desvios em relação à arquitetura planejada, seja por desconhecimento, requisitos conflitantes, dificuldades técnicas, prazos curtos, etc. Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, este artigo propõe uma solução de conformidade arquitetural em integração contínua. Isso implica que o processo de conformidade arquitetural é ativado a cada integração de código e, quando violações não forem detectadas, o código poderá ser integrado ao repositório. Além disso, este artigo apresenta a ferramenta ArchCI – que implementa a solução proposta usando DCL como técnica de conformidade e Jenkins como servidor de CI – e uma avaliação controlada que demonstra a aplicabilidade da solução.*

1. Introdução

No decorrer de um projeto de software, desenvolvedores normalmente introduzem desvios em relação à arquitetura planejada, seja por desconhecimento, requisitos conflitantes, dificuldades técnicas, prazos curtos, etc. [14, 13, 19]. Isso se agrava em projetos com vários desenvolvedores uma vez que o acúmulo dos possíveis desvios arquiteturais que podem ocorrer durante sua implementação, são potencializados pelo aumento do número de desenvolvedores em um projeto, levando ao fenômeno conhecido como erosão arquitetural [11, 6]. Mais importante, esses desvios arquiteturais impactam negativamente o projeto, podendo anular características essenciais de um sistema, como manutenibilidade, reusabilidade, escalabilidade, portabilidade, etc. [13, 20].

Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, este artigo propõe uma solução de conformidade arquitetural em integração contínua. Isso implica que o processo de conformidade arquitetural é ativado a cada integração de código e, quando violações não forem detectadas, o código poderá ser integrado ao repositório, o que soluciona os problemas *(i)* e *(ii)*. Além disso, este artigo apresenta a ferramenta ArchCI – que implementa a solução proposta usando DCL (*Dependency Constraint Language*) como técnica de conformidade [19] e Jenkins como servidor de CI [16] – e uma avaliação controlada que demonstra a aplicabilidade da solução.

O restante deste artigo está organizado como a seguir. A Seção 2 introduz conceitos fundamentais ao estudo. A Seção 3 descreve a solução proposta que evita os problemas decorrentes de um processo de erosão arquitetural. A Seção 4 detalha a implementação da ferramenta ArchCI. A Seção 5 avalia a aplicabilidade da solução proposta. Por fim, a Seção 6 apresenta as considerações finais e trabalhos futuros.

2. Background

2.1. Controle de Versão

Um sistema de controle de versão (*Version Control System*, VCS) é um software com a finalidade de gerenciar diferentes versões no desenvolvimento de artefatos de um projeto [17, 18]. Como principal contribuição, oferece rastreabilidade das alterações, como o responsável pelas mudanças, hora e data, diferenças das versões, etc.

Os sistemas podem ser centralizados ou distribuídos [9]. VCSs centralizados apresentam repositórios de códigos, onde o acesso e a escrita de dados estão restritos a um grupo de desenvolvedores [2]. VCSs distribuídos, por outro lado, trabalham com a arquitetura *peer-to-peer*, de forma que cada cópia de um projeto contém todo o histórico e os metadados do projeto, garantindo aos desenvolvedores a capacidade de compartilhar as mudanças da forma que mais se adeque às suas necessidades [12]. Dentre as principais ferramentas de controle de versão – CVS, SVN, Git e Mercurial – escolheu-se, para o desenvolvimento deste projeto, o Git[1] por oferecer a possibilidade de se desenvolver de maneira centralizada e distribuída, além de ser um dos mais utilizados atualmente [12].

Neste artigo, é importante a contextualização com os seguintes conceitos [17, 18]: (i) *tag*, nome simbólico atribuído à uma versão específica; (ii) *branch*, um conjunto de versões de arquivos fontes que é identificado por uma *tag*; (iii) *commit*, comando que integra as alterações de um desenvolvedor a um *branch* do repositório local; e (iv) *push*, comando que integra uma série de *commits* de um desenvolvedor a um *branch* do repositório remoto.

2.2. Integração Contínua

Integração Contínua (*Continuous Integration*, CI) trata-se da prática de desenvolvimento de software, onde membros de uma equipe incorporam certas mudanças ao software, aplicando processos de compilação e testes que asseguram a integridade do projeto [8]. Essa prática facilita na detecção de erros e problemas nas fases anteriores à conclusão do software, visando um menor custo de reparo [3]. A solução proposta neste artigo objetiva

[1] http://git-scm.com/

complementar esse processo de integração, provendo meios de verificar a arquitetura do sistema de software.

Servidores de CI podem ser configurados para verificarem sempre que mudanças são realizadas em um repositório [7]. Assim, recupera as versões mais recentes das classes, compila o código, e, em seguida, executa os testes para integração, exibindo os resultados aos desenvolvedores [4]. Dentre os servidores de CI mais relevantes – Jenkins, TeamCity e CruiseControl – o Jenkins[2] conseguiu um alcance maior na comunidade *open-source*, tendo assim, certa vantagem para a identificação e correção de *bugs*, bem como certas melhorias, se tornando o servidor mais recomendado para este projeto [16].

2.3. Conformidade Arquitetural

Conforme um projeto de software é desenvolvido, sua arquitetura está sempre evoluindo à medida que seu sistema também evolui. Portanto, são necessários meios de rastrear essas evoluções e outros aspectos implícitos do sistema de software. Esse processo é chamado de *architectural monitoring* [11]. Torna-se, assim, imprescindível para um sistema de software garantir a conformidade entre a arquitetura planejada e sua implementação atual. Contudo, é comum o acúmulo de violações arquiteturais ao longo do tempo, levando ao fenômeno conhecido como erosão arquitetural [14].

Define-se como erosão arquitetural o fenômeno que ocorre quando a arquitetura implementada de um sistema de software diverge de sua arquitetura planejada [6]. Existem diversas técnicas para evitar a erosão arquitetural, bem como para se realizar o processo de *architectural monitoring*. Dentre as principais técnicas, pode-se citar: Modelos de Reflexão [10], Matrizes de Dependências Estruturais [15], Source Code Query Languages [21], ArchJava [1], Testes de Desenho [5], e Linguagens de Restrição Arquiteturais [19]. Dessas técnicas, a que será utilizada neste projeto será a linguagem DCL, devido ao seu fácil uso e ao fato da mesma apresentar uma alta expressividade na forma de se tratar o problema de erosão arquitetural.

DCL é uma linguagem declarativa de domínio específico, que apoia a definição de restrições estruturais entre módulos em sistemas orientados a objetos, tendo como objetivo principal, restringir a organização modular de um sistema de software, em vez de seu comportamento [19]. Através da definição de restrições estruturais por meio do DCL, torna-se possível capturar dois tipos de violações arquiteturais: *divergências* (quando uma dependência observada no código fonte não está de acordo com o modelo arquitetural do sistema) e *ausências* (dependência inexistente no código fonte, mas que é obrigatória de acordo com o modelo arquitetural). Essencialmente, esse modelo abrange qualquer forma de relação entre classes que podem ser verificadas estaticamente. Através da combinação de uma linguagem simples e autoexplicativa com uma ferramenta de suporte publicamente disponível, acredita-se que DCL possa auxiliar na prevenção da erosão arquitetural.

3. Solução Proposta

Embora processos de conformidade arquitetural identifiquem violações arquiteturais, (i) essas ferramentas são subutilizadas e (ii) violações detectadas são raramente corrigidas. Diante disso, esta seção descreve uma solução de conformidade arquitetural em integração contínua, conforme ilustrada na Figura 1.

[2]http://jenkins-ci.org/

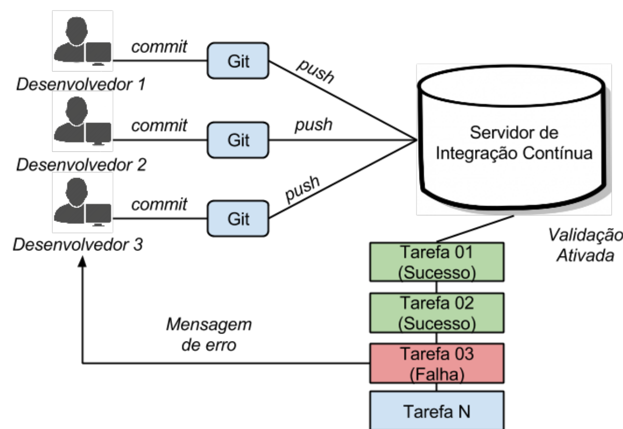

Figura 1. Funcionamento do ArchCI

O processo de conformidade arquitetural é ativado a cada integração de código e, quando violações não forem detectadas, o código poderá ser integrado ao repositório. Isso visa a integridade da arquitetura do software, uma vez que mantém o código-fonte sempre convergente com a arquitetura planejada. Assim, a solução proposta garante que:

- O processo de verificação de conformidade arquitetural seja realizado em toda integração de código sem a necessidade de instalações em máquinas de desenvolvedores, apenas no servidor de CI. Isso visa solucionar o problema *(i)* de subutilização de ferramentas de conformidade arquitetural.
- Como o processo de conformidade arquitetural é ativado a cada integração de código, é possível permitir a integração de código ao repositório apenas quando violações arquiteturais não forem detectadas. Isso visa solucionar o problema *(ii)* de violações detectadas serem raramente corrigidas.

É importante observar que a integração da solução proposta em processos reais de desenvolvimento de software contribuirá diretamente com a qualidade arquitetural do sistema de software, uma vez que a arquitetura implementada (como implementada no código fonte) estará sempre em conformidade com a arquitetura planejada.

3.1. Linguagem DCL

A solução proposta requer uma técnica de conformidade arquitetural subjacente. Para demonstrar a aplicabilidade da solução, utiliza-se DCL para definir as restrições arquiteturais de um projeto. Nessa técnica, define-se *módulos* que são conjunto de classes e, em seguida, *restrições arquiteturais* entre os módulos definidos, conforme Figura 2.

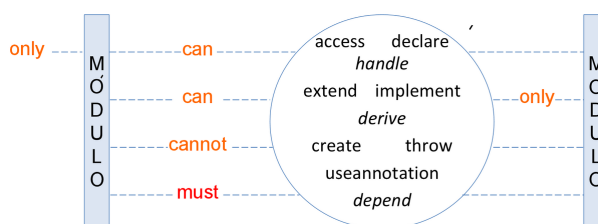

Figura 2. Sintaxe DCL

O exemplo a seguir demonstra a definição e o funcionamento de tais restrições estruturais entre módulos:

```
1: only Factory can-create Product
2: Util can-depend-only $java, Util
3: View cannot-access Model
4: Product must-implement Serializable
```

A restrição da linha 1 especifica que somente classes do módulo `Factory` podem criar objetos de classes no módulo `Product`. A restrição da linha 2 especifica que que as classes do módulo `Util` podem estabelecer dependências somente com o próprio módulo `Util` e a biblioteca padrão da linguagem Java. Já a restrição da linha 3 especifica que as classes do módulo `View` não podem acessar as classes do módulo `Model`. Por último, a restrição da linha 4 especifica que todas as classes no módulo `Product` devem implementar a interface `Serializable`.

Como pode ser observado, é de suma importância a definição de restrições arquiteturais. DCL provê quatro tipos de restrições: *cannot*, *can only*, *only can* e *must*. Assuma os módulos M_A e M_B de um sistema. Assuma também que A e B representem duas classes aleatórias do sistema e que $\overline{M_A}$ representa o complemento de M_A, assim como $\overline{M_B}$ representa o complemento de M_B. Por fim, assuma que `dep` corresponde às possíveis dependências que podem ser especificadas por meio do DCL, como *create*, *access*, *declare*, *handle*, etc. Dessa forma, é possível estipular a seguinte semântica vinculada ao tipo de restrição *cannot*:

$$\exists \mathtt{A}\, \exists \mathtt{B}\, [\, \mathtt{A} \in \mathtt{M_A} \;\wedge\; \mathtt{B} \in \mathtt{M_B} \;\wedge\; \mathtt{dep}(\mathtt{A},\mathtt{B})\,]$$

Assim, para os tipos de restrição *can only* e *only can*, as semânticas podem ser estipuladas em função da restrição *cannot*:

$$\mathtt{only}\ \mathtt{M_A}\ \mathtt{can\text{-}dep}\ \mathtt{M_B} \Longrightarrow \overline{\mathtt{M_A}}\ \mathtt{cannot\text{-}dep}\ \mathtt{M_B}$$

$$\mathtt{M_A}\ \mathtt{can\text{-}only\text{-}dep}\ \mathtt{M_B} \Longrightarrow \mathtt{M_A}\ \mathtt{cannot\text{-}dep}\ \overline{\mathtt{M_B}}$$

Por fim, a semântica vinculada ao tipo de restrição *must*:

$$\exists \mathtt{A}\, \neg\exists \mathtt{B}\, [\, \mathtt{A} \in \mathtt{M_A} \;\wedge\; \mathtt{B} \in \mathtt{M_B} \;\wedge\; \mathtt{dep}(\mathtt{A},\mathtt{B})\,]$$

3.2. Jenkins

A solução proposta requer um servidor de CI subjacente. Para demostrar a aplicabilidade da solução, utiliza-se o servidor Jenkins para programação das tarefas que garantam a conformidade arquitetural das integrações realizadas pelos desenvolvedores. Cada tarefa inclusa no Jenkins refere-se a um projeto de software específico, ou mesmo suas ramificações (seus diferentes *branches*). Sendo assim, cada integração (*push*) realizada ao repositório, dispara um gatilho no servidor, que dará início à tarefa específica do referente projeto, onde esta, por sua vez, validará somente as classes alteradas na determinada integração.

Caso o processo de conformidade arquitetural não detecte violações, as alterações serão integradas ao repositório com sucesso. No entanto, caso violações sejam detectadas, a tentativa de integração será negada, informando ao desenvolvedor que acionou a tarefa, as violações encontradas. Desse modo, a ferramenta garante a propriedade de atomicidade, assegurando que somente as integrações que estejam em total acordo com as restrições de dependência estabelecidas sejam aceitas pelo servidor, rejeitando, assim, alterações que estejam em desacordo ou parcial acordo, mesmo que as mesmas sejam uma série de integrações realizadas ao servidor local (*commits*) antes da requisição de integração ao servidor remoto (*push*).

4. Ferramenta ArchCI

A ferramenta ArchCI implementa a solução proposta, tendo sua concepção voltada para o uso da linguagem DCL como técnica subjacente de conformidade arquitetural e o Jenkins como servidor de CI. Primeiramente, foi necessário criar uma implementação *standalone* de DCL que dependesse apenas de uma biblioteca de manipulação de AST (Eclipse JDT, *Java Development Tools*). Dessa forma, tornou-se possível sua integração ao Jenkins, o que implica que a ferramenta não requeira qualquer instalação em máquina cliente. Além disso, a verificação de conformidade arquitetural das integrações no servidor remoto é realizada de forma individual, verificando cada arquivo separadamente e, mais importante, apenas aqueles que sofreram alguma modificação.

Conforme ilustrado na Figura 3, a implementação de ArchCI segue uma arquitetura com cinco módulos principais:

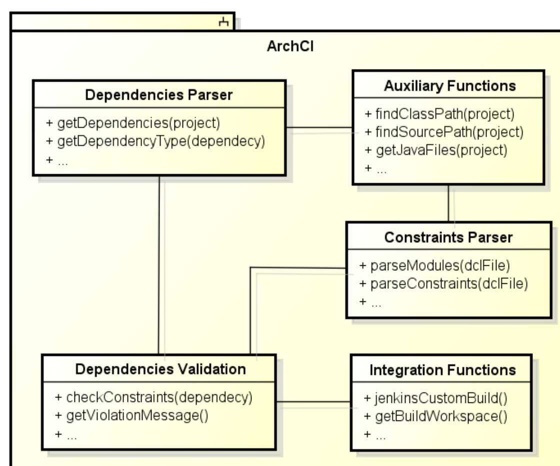

Figura 3. Arquitetura do ArchCI

- *Dependencies Parser*: Módulo responsável pela obtenção das dependências de um projeto, assim como a manipulação das mesmas. Apresenta funções que analisam cada elemento das classes a serem validadas, analisando o tipo de dependência ao qual o determinado elemento se refere.
- *Constraints Parser*: Módulo encarregado da análise e decomposição do arquivo contendo os módulos do projeto e as restrições de dependência estabelecidas para a arquitetura do sistema.

- *Dependencies Validation*: Módulo envolvendo funções para garantir a conformidade arquitetural do projeto por meio da verificação e validação de desvios arquiteturais com base nas restrições de dependência previamente estabelecidas.

- *Auxiliary Functions*: Módulo responsável por fornecer funções que auxiliem as tarefas do ArchCI de modo geral, tais como localizar o caminho das bibliotecas e dos arquivos necessários para a resolução das dependências, identificar o tipo de projeto, etc.

- *Integration Functions*: Módulo contendo as funções relacionadas às práticas de CI, assim como funções necessárias para integrar o código ao servidor Jenkins. Esse, por sua vez, engloba funções para a customização do *build*, obtenção do *workspace* com o código a ser integrado, identificação das classes a serem validadas, etc.

Por fim, após a conformidade arquitetural realizada durante a integração contínua, o ArchCI fornece como retorno uma mensagem de erro juntamente com as violações encontradas nas classes alteradas da integração, caso as mesmas existam. A interface da mensagem e sua representação é demonstrada na Figura 4(c), tendo como base o exemplo de restrição de dependência da Figura 4(a) e a violação da Figura 4(b).

```
module Main: project.main.*

Main cannot-depend java.lang.Math
```

(a) Exemplo de Restrição de Dependência

```
package project.main;

public class Main {
    public static void main(String[] args) {
        System.out.println(Math.pow(2, 5));
    }
}
```

(b) Exemplo de Violação

```
$ git add .
$ git commit -m "Integration Changes"
[dev a57ceb7] Integration Changes
 1 file changed, 4 insertions(+), 4 deletions(-)
$ git push

FAILURE
VIOLATION 1: 'project.main.Main' contains the method 'main' that statically invo
kes the method 'pow' of an object of 'java.lang.Math'
```

(c) Relatório após a tentativa de integração de código

Figura 4. Interface ArchCI

5. Avaliação

Para demonstrar a aplicabilidade da solução proposta, foi conduzida uma avaliação controlada envolvendo a aplicação `myAppointments` [13], um sistema de gerenciamento de informação pessoal simples implementado para ilustrar técnicas de conformidade arquitetural. O sistema possui funcionalidades primárias aos usuários, como criar, recuperar, atualizar e excluir contatos pessoais. Apesar de seu tamanho e sua complexidade serem simplificados, seu conjunto de restrições de dependência são provavelmente utilizados em muitos casos de conformidade arquitetural de projetos reais.

A arquitetura do `myAppointments` segue um padrão bastante conhecido chamado *Model-View-Controller* (MVC), fornecendo uma nítida divisão entre seus componentes. O componente *Model* encapsula o estado da aplicação, enquanto o componente *View* está associado à objetos da interface. O componente *Controller*, por sua vez, faz a mediação de todas as interações entre o *Model* e o *View*. Internamente ao componente *Model*, estão contidos *Domain Objects*, que representam entidades de domínio, e *Data Access Objects* (DAOs), que encapsulam o *framework* de persistência da aplicação. Tal arquitetura é ilustrada pelo diagrama da Figura 5:

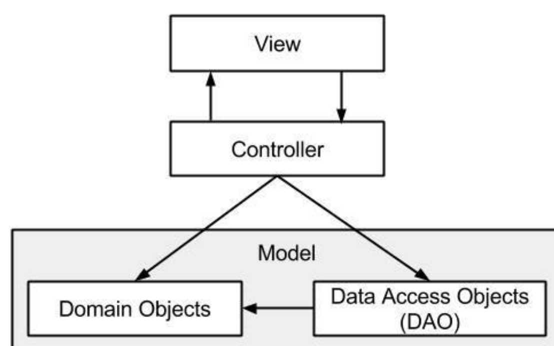

Figura 5. Arquitetura MVC do MyAppointments

Embora simplificado, o sistema do `myAppointments` trabalha com as principais restrições de dependência envolvendo o modelo MVC. Sua implementação usa as seguintes restrições arquiteturais (RA):

(RA1) Somente a camada *View* pode depender dos componentes providos pelo *AWT/Swing*.

(RA2) Somente os DAOs da camada *Model* podem depender dos serviços de banco de dados. Uma exceção é concedida para a classe `model.DB`, responsável por controlar as conexões do banco de dados.

(RA3) A camada *View* pode depender apenas dos serviços providos por ela mesma, pela camada *Controller* e pelo pacote *Util* (por exemplo, para dissociar a apresentação dos dados do acesso aos dados, componentes do *View* não podem acessar componentes do *Model* diretamente).

(RA4) *Domain Objects* não devem depender dos módulos DAO, *Controller* e *View*.

(RA5) Classes DAO podem depender somente de *Domain Objects*, das classes *Model* autorizadas a utilizar os serviços de banco de dados (como o `model.DB`), quanto do pacote *Util*.

(RA6) O pacote *Util* não pode depender de nenhuma classe específica do código fonte do sistema.

Para utilização da solução proposta, a definição das restrições arquiteturais em DCL é demonstrado na Figura 6.

```
%Módulos
module Controller:    myappointments.controller.*
module View:          myappointments.view.*
module Model:         myappointments.model.**
module Domain:        myappointments.model.domain.*
module Util:          myappointments.util.*
module DB:            myappointments.model.DB
module DAO:           "myappointments.model.[a-zA-Z0-9/.]*DAO"
module JavaAwtSwing:  java.awt.**, javax.swing.**
module JavaSql:       java.sql.**

%Restrições
only View can-depend JavaAwtSwing
only DAO, DB can-depend JavaSql
View cannot-depend Model
Domain can-depend-only $java
DAO can-depend-only Domain, Util, javaSql
Util cannot-depend $system
```

Figura 6. Restrições Arquiteturais DCL do MyAppointments

Para avaliar o funcionamento da solução proposta, foram intencionalmente criadas seis violações arquiteturais, uma para cada restrição de dependência do `myAppointments`, conforme ilustrado na Figura 7.

```
package myappointments.model.domain;

import java.util.Date; [...]

public class Appointment  {

    public Appointment() throws Exception{

        javax.swing.JOptionPane dialog;

        Date date = new java.sql.Date(2015, 04, 19);

        myappointments.controller.AgendaController ac;

    }
    [...]
```

(a) Violações RA1, RA2 e RA4

```
package myappointments.view;

import myappointments.model.AgendaDAO; [...]

public class AppointmentView extends
        AbstractAppointmentView {
    public AppointmentView
            (AppointmentController controller)
                throws Exception {
        this.controller = controller ;
        this.appForm = new AppointmentForm() ;

        Object aDAO = AgendaDAO.getInstance();

        initComponents() ;
        [...]
```

(b) Violação RA3

```
package myappointments.model;

import myappointments.view.AppointmentView; [...]

public class AgendaDAO extends AbstractAgendaDAO  {

    private static AgendaDAO agendaDAO
        = new AgendaDAO();

    private static AppointmentView av;
    [...]
```

(c) Violação RA5

```
package myappointments.util;

import myappointments.view.AppointmentView; [...]

public class DateUtils {

    public static final String HOUR_FMT = "HH:mm";
    public static final String SHORT_DATE_FMT =
        "MM/dd/yyyy";
    public static final String LONG_DATE_FMT =
        "MM/dd/yyyy HH:mm";

    private static AppointmentView av;
    [..]
```

(d) Violação RA6

Figura 7. Violações introduzidas no MyAppointments

(RA1) Um variável do tipo `javax.swing.JOptionPane` foi declarada dentro da classe `Appointment` que pertence a camada *Domain*. Isso representa uma violação na restrição (RA1) que indica que *somente a camada View pode depender dos componentes providos pelo* `AWT/Swing` (vide Figura 7(a)).

(RA2) Foi instanciado um objeto do tipo `java.sql.Date` na classe `Appointment` da camada *Domain*, violando a restrição (RA2) de que *somente os DAOs da camada Model podem depender dos serviços de banco de dados* (vide Figura 7(a)).

(RA3) O método `getInstance()` do objeto `AgendaDAO`, pertencente à camada DAO foi invocado pela classe `AppointmentView` da camada *View*, violando a restrição (RA3) de que *a camada View pode depender apenas dos serviços providos por ela mesma, pela camada Controller e pelo pacote Util* (vide Figura 7(b)).

(RA4) A classe `Appointment` da camada *Domain* instancia a variável `ac` do tipo `AgendaController` (pertencente à camada *Controller*), violando a restrição (RA4) de que *Domain Objects não devem depender dos módulos DAO, Controller e View* (vide Figura 7(a)).

(RA5) A classe `AgendaDAO` presente na camada DAO contém o campo `av` do tipo `AppointmentView` (pertencente à camada *View*), violando a restrição (RA5) de que *classes DAO podem depender somente de Domain Objects, das classes Model autorizadas a utilizar os serviços de banco de dados, quanto do pacote Util* (vide Figura 7(c)).

(RA6) Assim como na violação anterior, a classe `DateUtils` presente na camada *Util* contém o campo `av` do tipo `AppointmentView` (pertencente à camada *View*), violando a restrição (R6) de que *o pacote Util não pode depender de nenhuma classe específica do código fonte do sistema* (vide Figura 7(d)).

Dessa forma, foi realizada a avaliação do sistema para verificar se a ferramenta proposta é capaz de realizar o processo de conformidade arquitetural corretamente, detectando todas as violações criadas. O resultado da aplicação da ferramenta é demonstrado na Figura 8. Conforme observado, o ArchCI foi capaz de encontrar todas as violações criadas para esta avaliação com sucesso, cancelando o processo de integração (*push*) e informando as referentes violações ao desenvolvedor, cumprindo assim, seu propósito.

```
$ git add .
$ git commit -m "MyAppointments Changes"
[dev 8976d37] MyAppointments Changes
 23 files changed, 2189 insertions(+), 39 deletions(-)
$ git push

FAILURE
VIOLATION 1: 'myappointments.model.domain.Appointment' contains the local variable 'dialog' i
n method 'Appointment' whose type is 'javax.swing.JOptionPane'
VIOLATION 2: 'myappointments.model.domain.Appointment' contains the method 'Appointment' that
 creates an object of 'java.sql.Date'
VIOLATION 3: 'myappointments.model.domain.Appointment' contains the local variable 'ac' in me
thod 'Appointment' whose type is 'myappointments.controller.AgendaController'
VIOLATION 4: 'myappointments.view.AppointmentView' contains the method 'AppointmentView' that
 statically invokes the method 'getInstance' of an object of 'myappointments.model.AgendaDAO'
VIOLATION 5: 'myappointments.model.AgendaDAO' contains the field 'av' whose type is 'myappoin
tments.view.AppointmentView'
VIOLATION 6: 'myappointments.util.DateUtils' contains the field 'av' whose type is 'myappoint
ments.view.AppointmentView'
```

Figura 8. Violações detectadas pelo ArchCI no MyAppointments

Limitações: A avaliação foi realizada em um ambiente controlado – um sistema de pequeno porte, um único desenvolvedor, poucas integrações de código e um pequeno conjunto de violações. Entretanto, o objetivo da avaliação de se verificar a aplicabilidade da solução proposta foi atingido ao se demonstrar que é sim possível integrar um processo de conformidade arquitetural em Integração Contínua.

6. Conclusão

É de suma importância para a engenharia de software garantir a conformidade arquitetural de um sistema, principalmente no desenvolvimento de software em conjunto, onde problemas como a erosão arquitetural tornam-se mais comuns, causando a anulação de características como manutenibilidade, reusabilidade, escalabilidade, portabilidade, etc.

Este artigo apresenta uma solução para a verificação da conformidade arquitetural de um projeto de software – com base em restrições arquiteturais entre módulos – incorporadas em um servidor de Integração Contínua. Como principal contribuição, a solução proposta evita os problemas decorrentes de um processo de erosão arquitetural através de um processo de conformidade arquitetural mais rígido, e.g., integrações de código só ocorrem quando não foram detectadas violações arquiteturais.

Como trabalho futuro, pretende-se: (i) aplicar a solução proposta em cenários reais de desenvolvimento a fim de avaliar sua expressividade, aplicabilidade e desempenho; (ii) avaliar a usabilidade da ferramenta, e.g., melhor forma de se realizar a verificação, melhor forma de se apresentar as violações, além das características mais importantes para aceitação dos desenvolvedores, considerando distintas abordagens para o reporte de violações, e.g., exibição de *warnings* ou envio de e-mails, ao invés de bloquear a integração de código; (iii) analisar como fatores humanos influenciam as violações para propor novas funcionalidades; e (iv) realizar melhorias na implementação da ferramenta.

Agradecimentos

Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting software architecture to implementation. In *24th International Conference on Software Engineering (ICSE)*, pages 187–197, 2002.

[2] Brian De Alwis and Jonathan Sillito. Why are software projects moving from centralized to decentralized version control systems? In *2nd Cooperative and Human Aspects on Software Engineering (CHASE)*, pages 36–39, 2009.

[3] Alan Berg. *Jenkins Continuous Integration Cookbook*. Packt Publishing, Birmingham, 2012.

[4] Jon Bowyer and Janet Hughes. Assessing undergraduate experience of continuous integration and test-driven development. In *28th International Conference on Software Engineering (ICSE)*, pages 691–694, 2006.

[5] João Brunet, Dalton Serey, and Jorge Figueiredo. Structural conformance checking with design tests: An evaluation of usability and scalability. In *27th International Conference on Software Maintenance (ICSM)*, pages 143–152, 2011.

[6] Lakshitha de Silva and Dharini Balasubramaniam. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[7] Paul M. Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education, Boston, 2007.

[8] Martin Fowler and Matthew Foemmel. Continuous integration. Technical report, Thought-Works, 2006.

[9] Konrad Hinsen, Konstantin Läufer, and George K. Thiruvathukal. Essential tools: Version control systems. *Computing in Science & Engineering*, 11(6):84–91, 2009.

[10] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering (FSE)*, pages 18–28, 1995.

[11] Oscar Nierstrasz and Mircea Lungu. Agile software assessment. In *20th International Conference on Program Comprehension (ICPC)*, pages 3–10, 2012.

[12] Bryan O'Sullivan. Making sense of revision-control systems. *Queue*, 7(7):30–40, 2009.

[13] Leonardo Passos, Ricardo Terra, Renato Diniz, Marco Tulio Valente, and Nabor Mendonça. Static architecture conformance checking: An illustrative overview. *IEEE Software*, 27(5):132–151, 2010.

[14] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[15] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 167–176, 2005.

[16] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, Inc, Sebastopol, 2011.

[17] Diomidis Spinellis. Version control, part 1. *IEEE Software*, 22(5):107–107, 2005.

[18] Diomidis Spinellis. Version control, part 2. *IEEE Software*, 22(6):c3–c3, 2005.

[19] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094, 2009.

[20] Ricardo Terra and Marco Tulio Valente. Definição de padrões arquiteturais e seu impacto em atividades de manutenção de software. In *VII Workshop de Manutencao de Software Moderna (WMSWM)*, pages 1–8, 2010.

[21] Mathieu Verbaere, Michael W. Godfrey, and Tudor Gîrba. Query technologies and applications for program comprehension. In *16th IEEE International Conference on Program Comprehension*, pages 285–288, 2008.