Verificação Estática de Arquiteturas de Software utilizando Restrições de Dependência

Ricardo Terra, Marco Túlio Valente

Instituto de Informática, PUC Minas
Anel Rodoviário Km 23,5 - São Gabriel
31.980-110 – Belo Horizonte – MG – Brasil

rterrabh@gmail.com, mtov@pucminas.br

Resumo. *Este artigo descreve um sistema para verificação estática de arquiteturas de software que permite a arquitetos restringir o espectro de dependências possíveis em um dado sistema. O objetivo principal é permitir a definição de dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema. Uma vez definidas, tais restrições são automaticamente verificadas por uma ferramenta, evitando assim erosões silenciosas na arquitetura. O artigo também apresenta resultados da aplicação da linguagem de restrição de dependência proposta em um sistema de gerenciamento de recursos humanos, com cerca de 240 KLOC. Como resultado, a abordagem proposta foi capaz de detectar diversas violações na arquitetura desse sistema.*

Abstract. *This paper proposes a static software architecture verification approach based on a dependency constraint language that allows software architects to restrict the spectrum of dependencies that can be present in a given software system. The ultimate goal is to allow the definition of acceptable and unacceptable dependencies according to the planned system architecture. Once defined, such restrictions are automatically enforced by a tool, thus avoiding silent erosions in the system architecture. The paper also presents results of applying the proposed dependency constraint language in a 240 KLOC human resource management system. The proposed approach was able to detect several violations in the planned architecture of this system.*

1 Introdução

Arquitetura de software é geralmente definida como um conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução de sistemas de software. Isto inclui como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir [4]. Apesar de sua inquestionável importância, a arquitetura documentada de um sistema – se disponível – geralmente não reflete a sua implementação atual. Na prática, desvios em relação à arquitetura planejada são comuns, devido a desconhecimento por parte dos desenvolvedores, requisitos conflitantes, dificuldades técnicas etc [6]. Ainda mais importante, tais desvios geralmente não são capturados e resolvidos, levando aos fenômenos conhecidos como erosão e desvio arquitetural [12].

Este artigo é centrado na observação de que dependências inter-modulares impróprias são uma fonte importante de violações arquiteturais. Por exemplo, suponha um sistema organizado estritamente nas camadas $M_p, M_{p-1}, \ldots, M_0$ (onde M_0 representa

o módulo de mais baixo nível a hierarquia). Assim, nesse sistema, M_i somente pode utilizar serviços providos pelo módulo M_{i-1}, $i > 0$. Qualquer modificação no sistema que viole essa regra está, de fato, violando sua arquitetura. Um outro exemplo seria um sistema *web* que inclui um módulo de controle C e um módulo P que encapsula serviços de persistência. Assim, nesse sistema, C é o único módulo que pode manipular requisições e respostas HTTP (usando *servlets* ou outra tecnologia similar). Do mesmo modo, P é o único módulo que pode utilizar os serviços providos por um *framework* de persistência.

Linguagens de programação normalmente suportam ocultamento de informação por meio de modificadores de visibilidade (tais como `public`, `private` e `protected`). Entretanto, elas não provêm meios para restringir dependências intermodulares. Na prática, qualquer serviço público provido por um módulo (ou classe) M pode ser utilizado por qualquer outro módulo do sistema. Para tratar esse problema, propõe-se neste artigo uma linguagem de restrição de dependência que permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidadas em um dado sistema. Essa linguagem permite que arquitetos definam dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema. Pode-se, por exemplo, definir que serviços de persistência somente podem ser chamados a partir de classes integrantes da camada de modelo de um sistema. Uma vez definidas, tais restrições são automaticamente verificadas por uma ferramenta integrada ao ambiente de programação Eclipse. Assim, o objetivo central da ferramenta proposta é prover conformação arquitetural por construção, por meio de uma linguagem de restrição de dependência declarativa, que pode ser validada empregando-se apenas técnicas de análise estática de programas.

A fim de validar a abordagem de verificação arquitetural proposta, descreve-se também no artigo sua aplicação em um sistema real de grande porte, chamado SGP (Sistema de Gestão de Pessoas), utilizado atualmente pelo Serviço Federal de Processamento de Dados (SERPRO) para gestão de recursos humanos. Restrições de dependência foram especificadas para este sistema e então aplicadas em três versões do mesmo, com o auxílio da ferramenta proposta no trabalho. Como resultado, foi observado, por exemplo, que de um subconjunto de 18 restrições de dependência definidas para a terceira versão analisada do sistema – a qual possui cerca de 240 mil linhas de código e mais de 2.300 classes – 12 restrições foram violadas em pelo menos um ponto do sistema. Ao todo, 379 violações das restrições de dependência desse subconjunto foram detectadas nesta terceira versão analisada do sistema. Ou seja, a abordagem proposta foi efetivamemente capaz de detectar desvios na arquitetura de um sistema de grande porte, os quais não eram de conhecimento de seus arquitetos. Essa detecção foi realizada de modo estático e não-invasivo, sem impactar a versão em produção do sistema.

O restante deste artigo está organizado conforme descrito a seguir. A Seção 2 apresenta uma visão geral da solução proposta. A Seção 3 apresenta a linguagem de restrição de dependência, a qual constitui o componente central da abordagem de verificação arquitetural proposta no trabalho. A Seção 4 ilustra a aplicação dessa linguagem em um estudo de caso envolvendo diferentes versões do sistema SGP. A Seção 5 discute os principais benefícios e limitações da abordagem de verificação arquitetural proposta no trabalho. A Seção 6 descreve trabalhos relacionados e a Seção 7 conclui.

2 Abordagem Proposta

A abordagem de verificação arquitetural proposta utiliza técnicas de análise estática para detectar dependências inter-modulares impróprias, as quais constituem uma fonte importante de violações arquiteturais. Conforme ilustrado na Figura 1, inicialmente o arquiteto de software deve definir as restrições de dependência válidas no sistema, utilizando para isso uma linguagem de restrição de dependência, descrita na Seção 3. Para definição dessas restrições, o arquiteto deve se basear no modelo arquitetural (isto é, em um modelo que represente a arquitetura esperada ou planejada de um sistema) e em um modelo do código fonte, o qual pode incluir, por exemplo, diagramas de classe, de pacotes e de interação. A solução proposta inclui também uma ferramenta, chamada VARD (Verificação Arquitetural por Restrições de Dependência), que automaticamente detecta pontos do código fonte que violam as restrições de dependência especificadas.

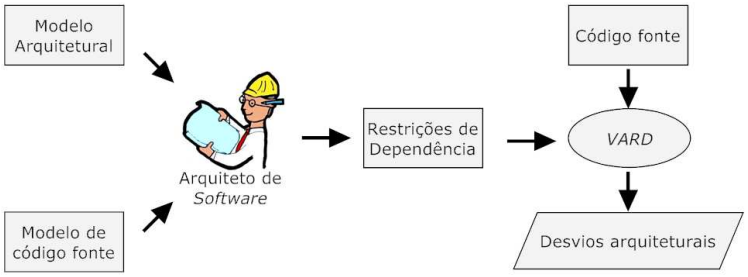

Figura 1. Visão da abordagem proposta

Como resultado da aplicação das restrições de dependência sobre o código fonte de um sistema, três diferentes situações podem ocorrer [11, 6]:

- Convergência: quando uma dependência existente no código fonte está de acordo com o modelo arquitetural (e com as restrições de dependência especificadas).

- Divergência: quando uma dependência observada no código fonte não está de acordo com o modelo arquitetural do sistema. Por exemplo, quando se tenta instanciar de um objeto do tipo `A` (via operador `new`) em um módulo que não foi especificado como sendo uma fábrica do tipo `A`.

- Ausência: dependência inexistente no código fonte, mas que é obrigatória de acordo com o modelo arquitetural. Por exemplo quando as classes do módulo `M` não implementam uma determinada interface `I`, quando isso deveria ser obrigatório.

A ferramenta VARD foi implementada como um *plug-in* do ambiente Eclipse. A ferramenta apresenta divergências e ausências na mesma janela onde são exibidos os erros e os alertas de compilação, porém rotulados como *desvios arquiteturais*. Normalmente, um desvio arquitetural deve ser analisado e corrigido pelo próprio desenvolvedor que o cometeu. No entanto, em certos casos, pode ser necessário que o arquiteto de software tenha que ajustar o modelo arquitetural e/ou as restrições de dependência especificadas. Ou seja, a ferramenta proposta suporta conformação arquitetural por construção.

3 Linguagem de Restrição de Dependência

O propósito da linguagem para verificação arquitetural utilizada neste trabalho é permitir a definição de restrições de dependência entre módulos[1]. Na notação utilizada, um módulo é um conjunto de classes. Suponha, por exemplo, as seguintes definições de módulos:

```
module A: org.foo.persistence.*
module B: org.foo.**
module C: org.foo.view.*, org.foo.model.Employee
```

Neste exemplo, o módulo `A` inclui todas as classes públicas do pacote `org.foo.persistence`. O módulo `B` inclui todas as classes públicas declaradas na hierarquia do pacote `org.foo`, isto é, inclui o pacote e todos os seus subpacotes. O módulo `C` inclui todas as classes públicas do pacote `org.foo.view` e a classe `Employee` do pacote `org.foo.model`.

A linguagem proposta permite a definição das seguintes restrições:

- *Somente classes do módulo A podem depender de tipos definidos no módulo B*, onde as possíveis dependências são as seguintes:

  - `only A can-access B`: somente classes declaradas no módulo `A` podem acessar membros não-privados de classes declaradas no módulo `B`. Acessar neste caso significa chamar métodos ou ler ou escrever em atributos.
  - `only A can-declare B`: somente classes declaradas no módulo `A` podem declarar variáveis de tipos declarados no módulo `B`.
  - `only A can-handle B`: somente classes declaradas no módulo `A` podem acessar e declarar variáveis de tipos declarados no módulo `B`. Em outras palavras, esta restrição é uma abreviação para `only A can-access, can-declare B`[2].
  - `only A can-create B`: somente classes declaradas no módulo `A` podem criar objetos de classes declaradas no módulo `B`.
  - `only A can-extend B`: somente classes declaradas no módulo `A` podem estender classes declaradas no módulo `B`.
  - `only A can-implement B`: somente classes declaradas no módulo `A` podem implementar interfaces declaradas no módulo `B`.
  - `only A can-derive B`: somente classes declaradas no módulo `A` podem derivar tipos declarados no módulo `B`. Portanto, esta restrição é uma abreviação para `only A can-extend, can-implement B`.
  - `only A can-throw B`: somente métodos de classes declaradas no módulo `A` podem lançar exceções declaradas no módulo `B`.

[1] Uma versão preliminar dessa linguagem é descrita em [14].

[2] A princípio, pode parecer que as restrições `access` e `declare` estão sempre vinculadas à restrição `handle`. Porém, existem situações nas quais se acessa determinada classe sem a necessidade de declará-la como ocorre, por exemplo, em acessos a atributos ou métodos estáticos. A situação inversa – nas quais se declara uma variável de um tipo `A`, mas não são acessados atributos ou métodos de `A` – ocorre, por exemplo, em métodos de classes que representam Adaptadores ou Fachadas, nos quais os parâmetros formais são usados apenas na chamada de outros métodos, o que não demanda, portanto, acesso aos mesmos.

  - `only A can-annotated B`: somente classes e seus membros declarados no módulo `A` podem receber anotações declaradas no módulo `B`.

- *Classes declaradas no módulo A não podem depender de tipos definidos no módulo B*, onde as dependências que não são permitidas são as seguintes:

  - `A cannot-access B`: nenhuma classe declarada no módulo `A` pode acessar membros não-privados de classes declaradas no módulo `B`.
  - `A cannot-declare B`: nenhuma classe declarada no módulo `A` pode declarar variáveis de tipos declarados no módulo `B`.
  - `A cannot-handle B`: nenhuma classe declarada no módulo `A` pode acessar ou declarar variáveis de tipos declarados no módulo `B`.
  - `A cannot-create B`: nenhuma classe declarada no módulo `A` pode criar objetos de classes declaradas no módulo `B`.
  - `A cannot-extend B`: nenhuma classe declarada no módulo `A` pode estender classes declaradas no módulo `B`.
  - `A cannot-implement B`: nenhuma classe declarada no módulo `A` pode implementar interfaces declaradas no módulo `B`.
  - `A cannot-derive B`: nenhuma classe declarada no módulo `A` pode derivar de tipos declarados no módulo `B`.
  - `A cannot-throw B`: nenhum método das classes declaradas no módulo `A` pode lançar exceções declaradas no módulo `B`.
  - `A cannot-annotated B`: nenhuma classe nem seus respectivos membros declarados no módulo `A` podem receber anotações do módulo `B`.

- *Classes declaradas no módulo A devem depender de tipos definidos no módulo B*, onde as dependências que são obrigatórias são as seguintes:

  - `A must-extend B`: todas as classes declaradas no módulo `A` devem estender uma das classes declaradas no módulo `B`.
  - `A must-implement B`: todas as classes declaradas no módulo `A` devem implementar pelo menos uma interface declarada no módulo `B`.
  - `A must-derive B`: todas as classes declaradas no módulo `A` devem derivar de pelo menos um tipo declarado no módulo `B`.

É importante mencionar que na verificação das restrições *only can* e *cannot* somente é possível que detectar divergências, pois se declara o que é permitido (no caso de *only can*) ou o que é proibido (no caso de *cannot*). Por outro lado, nas restrições *must* só é possível que ocorram ausências, pois se declaram restrições que são mandatórias.

4 Estudo de Caso

A fim de validar a abordagem de verificação arquitetural proposta neste artigo, foi realizado um estudo de caso com o sistema SGP (Sistema de Gestão de Pessoas). Esse sistema foi especificado e desenvolvido pelo SERPRO para automatizar a gestão dos seus funcionários. Atualmente, o sistema SGP automatiza atividades como gerenciamento de empregados, folha de pagamento, benefícios (plano de cargo, plano de saúde, pensão,

aposentadoria etc), atendimento às leis trabalhistas (férias, licenças etc) de aproximadamente 12 mil empregados.

O projeto do sistema SGP iniciou-se em dezembro de 2005 e sofreu uma importante reestruturação arquitetural em maio de 2006. Assim, pode-se considerar que a primeira versão estável do sistema foi disponibilizada em junho de 2006. Essa será a primeira versão analisada neste estudo de caso.

A arquitetura do sistema SGP segue o padrão arquitetural *Model View Controller* (MVC) [3], conforme ilustrado na Figura 2. A camada de Modelo contém Objetos de Negócio (*Business Objects* ou BOs), Objetos de Transferência de Dados (*Data Transfer Objects* ou DTOs) e Objetos de Acesso a Dados (*Data Access Objects* ou DAOs). BOs encapsulam regras de negócio e comportamentos. DTOs representam entidades do domínio, tais como empregados, planos de cargo, departamentos etc. DAOs provêm uma interface para acesso ao *framework* de persistência subjacente. Particularmente, na implementação do sistema SGP utiliza-se o *framework* Hibernate para persistência objeto/relacional.

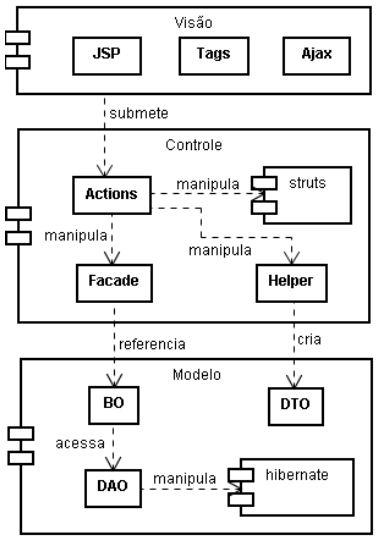

Figura 2. Arquitetura do Sistema SGP

A camada de Controle contém componentes que monitoram e adaptam entradas do usuário, manipulam o Modelo e atualizam a Visão. No sistema SGP, o *framework* Struts é utilizado pela camada de Controle para manipular requisições HTTP. Tais requisições são adaptadas por um componente *Helper* [2]. Em seguida, as mesmas são encaminhadas para um componente de fachada, que provê um ponto de acesso único à camada de Modelo. Finalmente, a camada de Visão é composta por *Java Server Pages* (JSP) e *Tags*. Em resumo, a arquitetura do sistema se baseia em padrões (*MVC*, *Factory*, *Helper*, *Facade*, *Business and Data Access Objects* etc) e em tecnologias (JSP, Hibernate, Struts, JAX-WS etc) que atualmente são largamente utilizados no desenvolvimento de sistemas *web*.

4.1 Metodologia

Após uma fase inicial de estudo e entendimento da arquitetura do sistema, foram selecionadas três versões em diferentes estágios do seu desenvolvimento, conforme descrito

	Primeira versão	Segunda versão	Terceira versão
Data	junho 2006	junho 2007	abril 2008
Linhas de código (LOC)	18.062	181.306	239.589
Pacotes	26	49	83
Classes/Interfaces	308	1.923	2.329
Bibliotecas externas (JARs)	32	60	68
Tecnologias	Java EE, JSP, Struts, Hibernate, Display-tag, Log4J e JSTL	Anteriores mais JAX-WS, Quartz, DWR e Hibernate Annotations	Anteriores mais JasperReports

Tabela 1. Versões analisadas no estudo de caso

na Tabela 1. A primeira versão analisada foi disponibilizada em junho de 2006, correspondendo à primeira versão estável do sistema; a terceira refere-se à versão do sistema em abril de 2008, quando o estudo aqui descrito foi realizado. Já a segunda versão analisada situa-se temporalmente entre a primeira e a terceira versões, isto é, escolheu-se uma versão do sistema disponibilizada em junho de 2007. É importante observar que houve um aumento significativo do tamanho do sistema entre essas versões. Enquanto a primeira versão escolhida possui pouco mais de 18 mil linhas de código, a última versão já possui quase 240 mil linhas de código.

A fim de validar a solução proposta, as seguintes atividades foram realizadas sobre cada uma das versões escolhidas:

1. Com apoio do arquiteto de software responsável pelo sistema SGP foram definidas restrições de dependência (RDs), incluindo a definição de módulos e das restrições *only can*, *cannot* e *must*. Esta atividade será descrita com mais detalhes na Seção 4.2.

2. A ferramenta VARD foi executada, de acordo com as restrições definidas no passo anterior.

3. Foram contabilizadas e analisadas as divergências e as ausências encontradas pela ferramenta, conforme será descrito na Seção 4.3.

4.2 Definição de Restrições de Dependência

Em conjunto com o arquiteto de software responsável pelo sistema SGP, foram definidas RDs para cada uma das três versões escolhidas para realização deste estudo de caso. Informações sobre o número e o tipo das restrições definidas são apresentadas na Tabela 2. Particularmente, ocorreram entre as versões, principalmente entre a primeira e a segunda versão, diversas evoluções arquiteturais visando aprimorar a qualidade do projeto do sistema. Com isso, RDs foram acrescentadas ou alteradas na lista de RDs definidas para a primeira versão do estudo de caso. Basicamente, isso ocorreu devido à incorporação de novas tecnologias e *frameworks* e à reestruturação de módulos do sistema (envolvendo divisão de pacotes, por exemplo).

Observa-se também na Tabela 2 que as restrições *only can* são mais comuns que as restrições do tipo *cannot*. Isto se justifica pelo fato de as restrições *only can* gerarem, implicitamente, diversas restrições *cannot*. Por exemplo, suponha um sistema com

três módulos: `A`, `B` e `C`. Quando se define uma restrição da forma `only A can-x B`, implica automaticamente na definição das seguintes restrições: `B cannot-x B` e `C cannot-x B`, onde `x = access, create, handle` etc. Assim, o número de restrições *cannot* é normalmente inferior ao de restrições *only can*.

	1ª versão	2ª versão	3ª versão
Número de módulos	19	32	37
Número de restrições *only can*	14	17	21
Número de restrições *cannot*	03	01	01
Número de restrições *must*	15	20	23
Total de restrições	32	38	45

Tabela 2. Número de RDs definidas para as versões do sistema SGP

A Figura 3 apresenta um subconjunto das RDs comuns às três versões escolhidas para desenvolvimento do estudo de caso[3]. Inicialmente, uma seqüência de definições de módulos é utilizada para agrupar classes relacionadas (linhas 1-19). Pode-se observar que os módulos definidos se assemelham àqueles existentes na visão arquitetural do sistema apresentada na Figura 2. Isto evidencia que a linguagem proposta permite expressar dependências entre entidades que são normalmente utilizadas por arquitetos de software para descrever seus sistemas.

Nas linhas 20-30, uma sequência de restrições *only can* é definida. Essencialmente, tais restrições são fundamentais para garantir que a arquitetura MVC seja preservada durante a evolução do sistema. Por exemplo, uma das restrições define que somente classes da camada de Controle podem manipular (isto é, acessar e declarar) tipos do módulo `IFacade` e do *framework* Struts (linha 21). Isto impede que a camada de Visão ultrapasse a camade de Controle e acesse diretamente o Modelo. Assim, basicamente, as restrições das linhas 20-30 têm como objetivo assegurar uma propriedade chave sobre a direção das dependências no padrão MVC: módulos da camada Controle devem depender de módulos da camada de Modelo, mas o contrário não é verdadeiro. Na realidade, módulos da camada de Modelo dependem somente do *framework* de persistência Hibernate (linha 26).

Usando a linguagem de restrição de dependência (LRD) proposta é possível tornar explícita a diferença entre fábricas e clientes de um certo tipo. Assim, há uma restrição que define que `BOs` somente possam ser criados na classe `BOFactory` (linha 23). Além do mais, uma outra restrição indica que somente `BOs` podem manipular interfaces de `DAO` (linha 28), porém não podem manipular diretamente as implementações dos `DAOs` (linha 32). É possível também restringir os métodos do programa que podem lançar exceções de determinado tipo. Como exemplo, exceções definidas no módulo `ControllerExcp` somente podem ser lançadas por métodos dos módulos `AllAction`, `Helper` ou `IFacade` (linha 29).

Nas linhas 33-40, define-se uma sequência de restrições *must*. Tais restrições são utilizadas para garantir que todas as classes que compõem um certo módulo estendem ou implementam um certo tipo. Geralmente, esse tipo é definido em um outro módulo do sistema ou é implementado por algum *framework* externo. Como um exemplo do primeiro

[3]Por questões de legibilidade, o nome do pacote `br.gov.serpro.sgp` foi abreviado para `sgp`.

```
 1: %Módulos
 2: module Controller:            sgp.controller.**
 3: module AllAction:             sgp.controller.action.**
 4: module Helper:                sgp.controller.helper.*
 5: module IFacade:               sgp.facade.*
 6: module BO:                    sgp.model.bo.*
 7: module IDAO:                  sgp.model.dao.*
 8: module HibernateDAO:          sgp.model.dao.hibernate.*
 9: module BasePojo:              sgp.pojo.ParentPersistent
10: module Pojo:                  sgp.pojo.**
11: module ControllerExcp:        sgp.controller.exception.*
12: module ModelExcp:             sgp.model.exception.*
13: module AgendadorQuartz:       sgp.controller.action.admin.AgendadorTarefaAction,
                                  sgp.controller.action.admin.QuartzAction
14: module Tags:                  sgp.taglib.**
15: module QuartzJob:             sgp.quartz.job.*
16: module QuartzPkgs:            sgp.quartz.**
17: module Quartz:                org.quartz.**
18: module Hibernate:             org.hibernate.**
19: module HibernateAnnotations:  org.hibernate.annotations.*

20: %Restrições only can
21: only Controller can-handle Struts, IFacade
22: only sgp.model.dao.hibernate.HibernateDAOFactory can-create HibernateDAO
23: only sgp.model.bo.DefaultBusinessObjectFactory can-create BO
24: only sgp.controller.service.HelperLocator can-create Helper
25: only Pojo can-annotated HibernateAnnotations
26: only BasePojo, HibernateDAO can-handle Hibernate
27: only AgendadorQuartz, QuartzPkgs can-handle Quartz
28: only BO can-handle IDAO
29: only AllAction, Helper, IFacade can-throw ControllerExcp
30: only BO, IDAO, HibernateDAO can-throw ModelExcp

31: %Restrições cannot
32: BO cannot-handle HibernateDAO

33: %Restrições must
34: AllAction must-extend sgp.controller.action.BaseAction
35: Pojo must-derive BasePojo, java.io.Serializable
36: QuartzJob must-extend sgp.quartz.job.SGPJob
37: IDAO must-implement sgp.model.dao.DataAccessObject
38: HibernateDAO must-implement IDAO
39: HibernateDAO must-extend sgp.model.dao.hibernate.HibernateImplDAO
40: Tags must-implement javax.servlet.jsp.tagext.JspTag
```

Figura 3. Subconjunto das RDs do sistema SGP

caso, todo `Action` deve estender uma classe interna do sistema chamada `BaseAction` (linha 34). Como exemplo do segundo caso, todas as classes no módulo `Tags` devem implementar a interface `javax.servlet.jsp.tagext.JspTag` (linha 40). Tais restrições são importantes para garantir que o sistema utilize corretamente os serviços providos por outras classes ou *frameworks*. Do mesmo modo, eles contribuem para orientar os desenvolvedores a utilizar *frameworks* corretamente, conforme prescrito pela arquitetura do sistema.

4.3 Resultados

Seguindo a metodologia descrita na Seção 4.1, a ferramenta VARD foi utilizada para verificar as RDs definidas para cada uma das três versões do sistema SGP. Em seguida, foram documentadas e contabilizadas as divergências e ausências encontradas conforme descrito na Tabela 3.

Algumas das restrições definidas apresentaram um considerável número de des-

Restrições de Dependência (RDs)	Divergências e ausências		
	1ª versão	2ª versão	3ª versão
Restrições *only can*			
21: **only** Controller **can-handle** Struts, IFacade	-	3	7
22: **only** sgp.model.dao.hibernate.HibernateDAOFactory **can-create** HibernateDAO	-	18	24
23: **only** sgp.model.bo.DefaultBusinessObjectFactory **can-create** BO	-	2	3
24: **only** sgp.controller.service.HelperLocator **can-create** Helper	-	2	2
25: **only** Pojo **can-annotated** HibernateAnnotations	ND	-	-
26: **only** BasePojo, HibernateDAO **can-handle** Hibernate	3	20	25
27: **only** AgendadorQuartz, QuartzPkgs **can-handle** Quartz	ND	1	2
28: **only** BO **can-handle** IDAO	-	65	72
29: **only** AllAction, Helper, IFacade **can-throw** ControllerExcp	-	17	25
30: **only** BO, IDAO, HibernateDAO **can-throw** ModelExcp	-	20	187
Restrições *cannot*			
32: BO **cannot-handle** HibernateDAO	-	25	30
Restrições *must*			
34: AllAction **must-extend** sgp.controller.action.BaseAction	-	-	-
35: Pojo **must-derive** BasePojo, java.io.Serializable	-	-	-
36: QuartzJob **must-extend** sgp.quartz.job.SGPJob	-	-	-
37: IDAO **must-implement** sgp.model.dao.DataAccessObject	-	-	-
38: HibernateDAO **must-implement** IDAO	-	1	1
39: HibernateDAO **must-extend** sgp.model.dao.hibernate.HibernateImplDAO	-	1	1
40: Tags **must-implement** javax.servlet.jsp.tagext.JspTag	-	-	-
Total	3	175	379

ND: Não Disponível.

Tabela 3. Violações das RDs definidas para o Sistema SGP

vios. Como exemplo, a arquitetura do sistema SGP define um serviço de criação de DAOs (restrição 22, da Tabela 3). Contudo, foram encontrados 18 classes na segunda versão e 24 classes na terceira versão nas quais DAOs são criados diretamente, sem a utilização desse serviço de fábrica. Como um outro exemplo, BOs somente podem manipular interfaces de DAOs e nunca a implementações dos mesmos (restrição 32). Essa restrição é importante para tornar as regras de negócio independentes do serviço de persistência utilizado. Porém, foram encontrados 25 BOs na segunda versão e 30 BOs na terceira versão que acessam diretamente implementações de DAOs. Além disto, foi constatado que diversas classes acessam interfaces de DAOs, sendo que somente BOs deveriam acessar tais interfaces (restrição 28). Por fim, restrições simples como as que definem que somente módulos da camada de Controle e de Modelo podem retornar exceções específicas dessas camadas (restrições 29 e 30) apresentaram várias divergências, indicando que existem módulos de uma camada levantando exceções de uma outra camada.

Por outro lado, dentre as restrições avaliadas, as restrições *must* foram aquelas que obtiveram o maior grau de convergência, ocorrendo uma ausência em uma única classe DAO que não seguiu duas RDs (restrições 38 e 39).

5 Avaliação

Assegurar que a implementação de um sistema esteja sempre de acordo com sua arquitetura é um dos problemas mais relevantes em desenvolvimento de software. Para tratar esse problema, foi proposto neste artigo uma abordagem de verificação arquitetural com

as seguintes características:

- Baseada em restrições de dependências entre os módulos de um sistema orientado por objetos. Como observado no estudo de caso da Seção 4, diversas violações na arquitetura de um sistema se manifestam no código fonte por meio do estabelecimento de relações inter-modulares impróprias. Por exemplo, na terceira versão analisada do sistema SGP, foram detectadas 379 restrições inter-modulares inválidas, no sentido de que as mesmas comprovadamente constituíram violações em relação à arquitetura original desse sistema. As violações detectadas foram reportadas aos arquitetos do sistema SGP, que então decidiram abrir ordens de manutenção para correção das mesmas.

- Baseada em técnicas de análise estática. Ou seja, a verificação da arquitetura de um sistema é realizada de forma totalmente não-invasiva e sem requerer sua execução. Essa característica foi decisiva para viabilizar a realização do estudo de caso com o sistema SGP. Como se trata de um sistema de grande porte cujos dados são altamente confidenciais, os arquitetos e gerentes do sistema SGP somente concordaram em disponibilizá-lo para realização do estudo de caso quando tiveram a garantia de que isso poderia ser feito em um ambiente totalmente independente do ambiente de desenvolvimento e execução do sistema. Acredita-se que essa seja uma preocupação comum a gerentes de grandes sistemas de software.

- Baseada em um mapeamento entre elementos arquiteturais e estruturas do código fonte de um sistema, por meio do qual são definidas as RDs. Ou seja, o objetivo da abordagem proposta não é (re)construir o modelo arquitetural de um sistema, mas sim verificar se o mesmo é seguido nas fases de implementação e manutenção. Portanto, a solução proposta requer um passo manual inicial, que consiste na definição das RDs. No entanto, a experiência com o sistema SGP, mostrou que a notação utilizada para definição dessas restrições é bastante simples e intuitiva, tendo sido rapidamente dominada pelo arquiteto responsável por esse sistema. Apesar de simples, é importante mencionar que para a definição dessas restrições, o arquiteto deve ter conhecimento da estrutura de classes, interfaces e pacotes do código fonte, bem como dos principais *frameworks* e bibliotecas externas utilizados. Por outro lado, a linguagem proposta permite definir módulos de mais alto nível, constituídos por um conjunto de classes e/ou pacotes. No estudo de caso realizado, tais módulos corresponderam em grande medida aos componentes originalmente usados pelo arquiteto do sistema SGP para descrever a organização desse sistema. Por fim, evoluções na arquitetura de um sistema podem demandar uma atualização nas RDs definidas, como ocorreu no estudo de caso ao se migrar de uma versão para outra do sistema.

O estudo de caso realizado demonstrou também o poder de expressão da LRD proposta no trabalho. Por exemplo, foram detectadas violações envolvendo todas as formas de dependência que podem ser expressas pela linguagem proposta (`declare`, `access`, `create`, `extend`, `annotated` etc). No entanto, a ferramenta VARD não é capaz de detectar dependências geradas por meio dos recursos de reflexão computacional de Java. Por exemplo, caso o código fonte inclua chamadas de métodos por meio de reflexão, a

linguagem proposta não será capaz de verificar se essas chamadas estão de acordo com as regras de dependência definidas.

Como é baseada em técnicas de análise estática, a solução proposta é indicada para detectar violações em propriedades estruturais de um sistema, como aquelas que tratam dos relacionamentos entre seus módulos. Por outro lado, essas técnicas não são adequadas para detectar propriedades comportamentais (ou dinâmicas) de um sistema. Por exemplo, não é possível expressar que toda execução de um determinado método `m1` deve ser seguida pela execução de um outro método `m2` ou mesmo que um determinado método `m1` deve ser sempre chamado ao se executar um método `m2`.

6 Trabalhos Relacionados

Pelo menos as seguintes técnicas já foram propostas para lidar com os problemas de erosão e desvio arquitetural:

Linguagens de restrição: Sangal et al. propuseram o emprego de *Dependency Structure Matrixes* (DSM) para revelar dependências e padrões arquiteturais subjacentes a um sistema de software [13]. Eles também propuseram o uso de regras de projeto para destacar entradas das DSMs que violam a arquitetura planejada de um sistema. A LRD proposta neste artigo é inspirada nesta linguagem de regras de projeto. Entretanto, a linguagem de Sangal et al. suporta a definição de somente duas formas de relações entre módulos: `can-use` e `cannot-use`. Por outro lado, nossa linguagem permite a definição de um conjunto mais rico de dependências, contemplando também operações de criação de objetos, derivação de tipos, ativação de exceções, uso de anotações etc.

Técnicas de outros paradigmas de programação também já foram propostas para definição de restrições arquiteturais, incluindo programação em lógica [8] e programação orientada por aspectos [9]. Contudo, objetivando uma solução simples, de fácil entendimento e eficiente, optamos por adotar na abordagem proposta uma linguagem específica de domínio, projetada exclusivamente para definição de RDs.

Ferramentas de recuperação arquitetural: *Frameworks* de recuperação arquitetural utilizam tecnologias de reengenharia de software para extrair modelos arquiteturais de alto nível de sistemas existentes. Como exemplo, podemos citar os sistemas Discotect [15] e Rigi [10]. O principal desafio de tais *frameworks* é recuperar modelos que sejam similares àqueles projetados pelos desenvolvedores dos sistemas alvo, em relação não apenas a correção do modelo, mas também ao nível de abstração dos elementos arquiteturais recuperados. *Reflexion models* (RM) tratam tal problema exigindo que desenvolvedores informem um modelo de alto nível da arquitetura planejada de um sistema e um mapeamento entre este modelo e o código fonte [11]. Ferramentas baseadas em RM (como, por exemplo, o *plug-in* SAVE para Eclipse [5, 6]) destacam convergências, divergências e ausências de relações entre esse modelo de alto nível e o código fonte. Entretanto, a abordagem proposta neste artigo disponibiliza um conjunto mais rico de relações entre módulos do que as linguagens utilizadas em RMs. Além disto, a linguagem proposta foi projetada para promover conformação arquitetural por construção, isto é, modificações que violam a arquitetura planejada de um sistema alvo são detectadas logo após serem implementadas no código fonte.

Linguagens de descrição arquitetural (ADL): ADLs representam outra alternativa para conformação arquitetural por construção [7]. Tais linguagens permitem aos desenvolvedores expressar o comportamento arquitetural e a estrutura de um sistema de software em uma linguagem declarativa e abstrata. Ferramentas de geração de código podem ser então utilizadas para mapear descrições arquiteturais para código fonte de uma dada linguagem de programação. Entretanto, tais abordagens normalmente requerem o uso de compiladores e ferramentas de desenvolvimento baseadas em ADLs para manter o código gerado sincronizado com a especificação arquitetural. Uma variante desta abordagem defende a extensão de linguagens de programação de propósito geral com construções para modelagem arquitetural, o que na prática exige que desenvolvedores dominem uma linguagem de programação nova [1]. A abordagem proposta neste artigo trata esse problema propondo uma linguagem declarativa simples e de fácil entendimento para definir dependências entre módulos.

7 Conclusões

O problema de garantir que a arquitetura planejada de um sistema é efetivamente seguida durante sua implementação e evolução possui inquestionável importância. Assim, neste artigo foi apresentada uma abordagem para verificação estática de arquiteturas de software que centra-se na observação de que dependências inter-modulares impróprias são uma fonte importante de violações arquiteturais. A abordagem proposta utiliza uma LRD para detectar várias dessas violações e inclui uma ferramenta integrada ao ambiente de desenvolvimento Eclipse, viabilizando assim conformação arquitetural por construção. A abordagem proposta foi validada por meio de um estudo de caso de grande porte (com quase 240 KLOC), tendo sido capaz de detectar quase quatro centenas de violações arquiteturais no código fonte desse sistema, das quais seus arquitetos não possuíam conhecimento.

Como trabalho futuro, pretende-se inicialmente investigar e delimitar melhor o espectro de violações arquiteturais que podem ser detectadas por meio da LRD proposta no trabalho. O estudo de caso realizado demonstrou a capacidade da solução proposta em detectar violações arquiteturais, não tendo ocorrido no mesmo nenhum caso de falso positivo. Isto é, as violações apontadas efetivamente correponderam a desvios em relação à arquitetura planejada do sistema. No entanto, não foi realizada uma análise de possíveis outras violações arquiteturais existentes no sistema analisado e que não foram detectadas pela solução proposta. A comprovação de falsos negativos pode motivar uma extensão da LRD proposta no trabalho e, eventualmente, justificar a incorporação de outras técnicas de análise de programa, incluindo técnicas de análise dinâmica. Por fim, pretende-se realizar um segundo estudo de caso com a solução proposta, envolvendo um sistema de grande porte que utilize padrões arquiteturais diferentes daqueles implementados no sistema SGP.

Agradecimentos: O desenvolvimento deste trabalho foi apoiado pela CAPES (por meio de uma bolsa de mestrado PROSUP) e pela FAPEMIG (processo PPM-CEX-APQ 4543-5). Gostaríamos de agradecer ao SERPRO (Unidade Belo Horizonte) e, principalmente, aos analistas Ricardo Valério Martelleto e Lucas Martins do Amaral, o apoio na verificação arquitetural do sistema SGP.

Referências

[1] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *22nd International Conference on Software Engineering (ICSE)*, pages 187–197, 2002.

[2] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2nd edition, 2003.

[3] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[4] David Garlan and Mary Shaw. *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[5] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 279–294, 2006.

[6] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *6th Working Conference on Software Architecture (WICSA)*, pages 1–12, 2007.

[7] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[8] Kim Mens, Roel Wuyts, and Theo D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *29th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 33–45, 1999.

[9] Paulo Merson. Using aspect-oriented programming to enforce architecture. Technical report, Software Engineering Institute, September 2007.

[10] Hausi A. Muller and K. Klashinsky. Rigi a system for programming-in-the-large. In *10th International Conference on Software Engineering (ICSE)*, pages 80–87, 1988.

[11] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Symposium on the Foundations of Software Engineering (FSE)*, pages 18–28, 1995.

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

[13] 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.

[14] Ricardo Terra and Marco Tulio de Oliveira Valente. Towards a dependency constraint language to manage software architectures. In *2nd European Conference on Software Architecture (ECSA)*, pages 1–8, 2008.

[15] Hong Yan, David Garlan, Bradley R. Schmerl, Jonathan Aldrich, and Rick Kazman. DiscoTect: A system for discovering architectures from running systems. In *26th International Conference on Software Engineering (ICSE)*, pages 470–479, 2004.