Ricardo Terra Nunes Bueno Villela

Conformação Arquitetural utilizando Restrições de Dependência entre Módulos

Dissertação apresentada ao Programa de Pós-Graduação em Informática da Pontifícia Universidade Católica de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Informática.

Belo Horizonte

Março de 2009

FICHA CATALOGRÁFICA

Elaborada pela Biblioteca da Pontifícia Universidade Católica de Minas Gerais

V735c

Villela, Ricardo Terra Nunes Bueno

Conformação arquitetural utilizando restrições de dependência entre módulos. / Ricardo Terra Nunes Bueno Villela. – Belo Horizonte, 2009.

74f. : il.

Orientador: Marco Túlio de Oliveira Valente.

Dissertação (Mestrado) – Pontifícia Universidade Católica de Minas Gerais, Programa de Pós-graduação em Informática.

Bibliografia.

1.Arquitetura de software – Teses. 2. Programação orientada a objetos. I. Valente, Marco Túlio de Oliveira. II. Pontifícia Universidade Católica de Minas Gerais. III. Título

CDU: 681.3.03

Bibliotecário: Fernando A. Dias – CRB6/1084

PUC Minas
Programa de Pós-graduação em Informática

ATA DE DEFESA DE DISSERTAÇÃO DO ALUNO
RICARDO TERRA NUNES BUENO VILLELA

Realizou-se, no dia dezenove de março de dois mil e nove, às 14:00 horas, na sala de Multimeios 30 - Bloco I da PUC Minas, Unidade São Gabriel, a 24ª defesa de dissertação do Programa de Pós-graduação em Informática, com título *"Conformação Arquitetural utilizando Restrições de Dependência entre Módulos"*, apresentada por RICARDO TERRA NUNES BUENO VILLELA. A banca examinadora foi composta pelos seguintes professores:

Prof: Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Profa: Claudia Maria Lima Werner - (UFRJ)
Prof: Roberto da Silva Bigonha - (UFMG)
Profa: Maria Augusta Vieira Nelson - (PUC Minas)

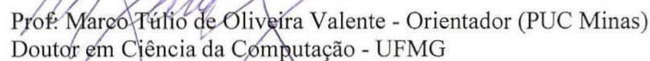
A banca examinadora considerou a dissertação:

(X) Aprovada (o candidato terá até dez dias para entregar o texto final da dissertação).
( ) Aprovada de forma condicional (o candidato terá até quarenta e cinco dias para entregar o texto final da dissertação).
( ) Reprovada.

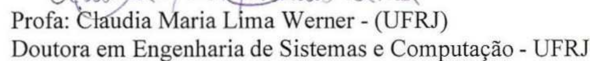
Finalizados os trabalhos, lavrei a presente ata que, lida e aprovada, vai assinada por mim e pelos membros da comissão.

Belo Horizonte, 19 de março de 2009.

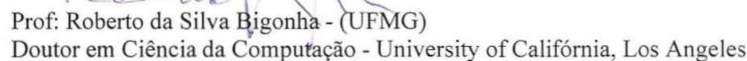
Giovana Cássia da Silva – Secretária



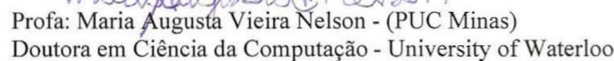
Prof: Marco Túlio de Oliveira Valente - Orientador (PUC Minas)
Doutor em Ciência da Computação - UFMG



Profa: Cláudia Maria Lima Werner - (UFRJ)
Doutora em Engenharia de Sistemas e Computação - UFRJ



Prof: Roberto da Silva Bigonha - (UFMG)
Doutor em Ciência da Computação - University of Califórnia, Los Angeles



Profa: Maria Augusta Vieira Nelson - (PUC Minas)
Doutora em Ciência da Computação - University of Waterloo

Anel Rodoviário – KM 23,5, Rua Walter Ianni, 255 – São Gabriel / Belo Horizonte – MG / CEP: 31980-110
Fone: (31) 3439-5204 – Fax: (31) 3439-5204 – e-mail: mestradoinfsec@pucminas.br

Resumo

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. Isso inclui como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir. 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 ao desconhecimento por parte dos desenvolvedores, requisitos conflitantes, dificuldades técnicas, etc. Ainda mais importante, tais desvios geralmente não são capturados e resolvidos, levando aos fenômenos conhecidos como erosão e desvio arquitetural.

Esta dissertação é centrada na observação de que dependências inter-modulares impróprias são uma fonte importante de violações arquiteturais, as quais contribuem para o processo de erosão arquitetural. Diante disso, é proposta uma linguagem de restrição de dependência, denominada DCL (*Dependency Constraint Language)*, que permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidas em sistemas orientados a objetos. O objetivo central é prover aos arquitetos meios para definir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada dos seus sistemas. Além disso, foi implementado um protótipo de uma ferramenta, chamada `dclcheck`, que verifica se o código fonte (isto é, a arquitetura concreta de um sistema) respeita restrições de dependência definidas em DCL.

A fim de validar a aplicabilidade da solução proposta em sistemas reais, descreve-se uma experiência de aplicação da linguagem DCL e da ferramenta `dclcheck` 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 seus empregados. Para isso, foram especificadas restrições de dependência em DCL para três diferentes versões do sistema, no sentido de garantir que sua implementação segue a arquitetura planejada. Em seguida, foi aplicada a ferramenta `dclcheck` em cada uma dessas versões. Como resultado, foi observado, por exemplo, que 19 das 28 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 – foram violadas em pelo menos um ponto do sistema. Ao todo, foram detectadas 245 classes com violações. Ou seja, a abordagem proposta foi efetivamente capaz de detectar desvios na arquitetura de um sistema de grande porte, os quais não eram do conhecimento de seus arquitetos.

Abstract

Software architecture is usually defined as a set of design decisions that are critical in each aspect of software development and evolution. This includes how systems are structured into components and constraints on how these components must interact. Despite its unquestionable importance, the documented architecture of a system – if available at all – usually does not reflect its actual implementation. In practice, deviations from the planned architecture are common, due to unawareness by developers, conflicting requirements, technical difficulties etc. More important, such deviations are usually not captured and solved, leading to the phenomena known as architectural drift and architectural erosion.

This dissertation is centered on the observation that improper inter-module dependencies are relevant sources of architectural violations, which contribute to the process of architectural erosion. Therefore, a dependency constraint language called DCL (*Dependency Constraint Language)* was proposed. DCL allows software architects to restrict the spectrum of structural dependencies that can be established in object-oriented systems. The ultimate goal is to provide architects with means to define acceptable and unacceptable dependencies according to the planned architecture of their systems. Moreover, a supporting tool called `dclcheck` was implemented. It verifies whether the source code (i.e. the concrete architecture of a system) honors dependency constraints defined in DCL.

In order to evaluate the applicability of the proposed solution, the DCL language and the `dclcheck` tool were applied to a large size human resource management system, called SGP, which is currently used by the Brazilian Federal Data Processing Service (SERPRO) to manage its employees. Dependency constraints have been defined in DCL for three different versions of this system, to ensure that its implementation follows the planned architecture. Next, the `dclcheck` tool has been applied to each of these versions. As result, it could be observed for example that 19 from 28 dependency constraints defined for the third version of the system – which has around 240 KLOC and more than 2,300 classes – have been violated in at least one point of the source code. More specifically, 245 classes with violations were detected. Therefore, the proposed approach was effective to detect deviations in the architecture of a large real-world system. It is also important to mention that SGP architects were not aware of such deviations.

Dedico esta dissertação aos meus pais por sempre acreditarem em mim e à minha avó Zita Nunes Bueno Vilela por, além de sempre me apoiar, ser uma pessoa especial na minha vida.

Agradecimentos

Em primeiro lugar, gostaria de agradecer aos meus pais todo apoio e carinho que me foram dados em todas as fases da minha vida e à minha família, especialmente minhas irmãs, por sempre estarem presentes nos momentos mais importantes da minha vida.

Agradeço ao professor Marco Túlio de Oliveira Valente, meu pai do mestrado, por, além de ser um orientador perfeito, ser um exemplo de comprometimento, dedicação, sabedoria e humildade. Devo muito a ele.

Agradeço à Giovana Silva, minha mãe do mestrado, por ser uma das pessoas mais especiais que já conheci. Dentre suas várias qualidades, sua atenção, prestatividade, apoio e carinho são imensuráveis. Devo muito a ela.

Agradeço aos professores do mestrado em Informática da PUC Minas, principalmente à Lucila Ishitani e ao Mark Alan Junho Song, todo o conhecimento e apoio transmitidos.

Agradeço aos meus colegas de mestrado, principalmente ao Alexander Flávio, Geovália Coelho e Virgílio Borges, o incentivo, o apoio e o carinho em todos os momentos.

Agradeço aos professores que compuseram a banca, Roberto Bigonha, Claudia Werner e Maria Augusta Vieira, a leitura cuidadosa deste texto e as sugestões construtivas.

Agradeço ao Lucas Martins Amaral e Ricardo Valério Martelleto a ajuda indispensável durante a realização do estudo de caso do sistema SGP.

Agradeço a minha avó, Zita Nunes Bueno Vilela, por me hospedar em sua casa por uma grande parte do desenvolvimento desta dissertação.

Finalmente, agradeço a Deus por sempre estar ao meu lado e por ter me dado todas as ferramentas e oportunidades para que eu chegasse onde estou.

Conteúdo

Lista de Figuras **vi**

Lista de Tabelas **vii**

1 Introdução **1**

1.1 Visão Geral do Problema 1
1.2 Objetivos 2
1.3 Visão Geral da Solução Proposta 3
1.4 Estrutura da Dissertação 4

2 Conformação Arquitetural **6**

2.1 Sistema Motivador 8
2.2 LDM 10
2.2.1 Utilizando LDM 11
2.3 .QL 14
2.3.1 Utilizando .QL 14
2.4 SAVE 18
2.4.1 Utilizando SAVE 19
2.5 Outros Trabalhos 22
2.6 Considerações Finais 24

3 A Linguagem DCL **29**

3.1 Visão Geral 29
3.2 Linguagem DCL 31
3.3 Exemplos 38
3.3.1 Tipos de Dependência 38

3.3.2 Sistema `myAppointments` . . . 41
3.4 `dclcheck` . . . 42
3.5 Considerações Finais . . . 45

4 Estudo de Caso **47**
4.1 Sistema de Gestão de Pessoas (SGP) . . . 47
4.2 Metodologia . . . 49
4.3 Restrições de Dependência . . . 50
4.4 Resultados . . . 54
4.5 Análise Crítica . . . 59
4.6 Considerações Finais . . . 62

5 Conclusão **63**
5.1 Visão Geral da Solução Proposta . . . 63
5.2 Comparação com Trabalhos Relacionados . . . 65
5.3 Contribuições . . . 66
5.4 Trabalhos Futuros . . . 67

A Gramática da Linguagem DCL **69**

Bibliografia **70**

Lista de Figuras

2.1 Tela principal do `myAppointments` 8

2.2 Arquitetura do `myAppointments` 9

2.3 Exemplo de uma DSM 11

2.4 DSM do `myAppointments` 12

2.5 Grafo de dependências gerado pela ferramenta .QL 16

2.6 Janela exibindo violação da restrição RA1 17

2.7 Modelo de reflexão para o sistema motivador 20

2.8 Modelo de código fonte do sistema motivador 21

2.9 Restrição LogEn especificando um padrão de projeto *Factory* 23

2.10 Modelagem e implementação de um componente Compilador em ArchJava 25

3.1 Solução proposta para conformação arquitetural 30

3.2 Sintaxe para declaração de restrições de dependência na linguagem DCL . 37

3.3 Restrições de dependência do sistema `myAppointments` 42

3.4 Funcionamento da ferramenta `dclcheck` 44

3.5 Tela do `dclcheck` com violações detectadas no sistema `myAppointments` . 45

4.1 Arquitetura do Sistema SGP 48

4.2 Restrições de dependência do sistema SGP 53

4.3 Trecho de um BO contendo a mesma violação em diversos pontos 55

4.4 Tela do `dclcheck` com violações detectadas no sistema SGP 56

Lista de Tabelas

2.1 Comparativo de DCL com outras técnicas 28

4.1 Versões analisadas no estudo de caso 49
4.2 Informações sobre as restrições de dependência definidas para o sistema SGP 50
4.3 Alterações nas restrições definidas (quando comparadas as três versões analisadas no sistema SGP) 51
4.4 Violações arquiteturais detectadas no sistema SGP 58
4.5 Tipos de Violação (somente com base na terceira versão do sistema SGP) . 59
4.6 Desempenho da ferramenta `dclcheck` (em segundos) 60

5.1 Comparativo de DCL com outras técnicas 65

Lista de Siglas

ADL *Architecture Description Language*

API *Application Programming Interface*

AWT *Abstract Window Toolkit*

BNF *Backus-Nahur Form*

BO *Business Object*

DAO *Data Access Object*

DCL *Dependency Constraint Language*

DSM *Dependency Structure Matrixes*

DTO *Data Transfer Object*

DWR *Direct Web Remoting*

FCL *Framework Constraint Language*

GLC Gramática Livre de Contexto

GUI *Graphical User Interface*

HTTP *Hypertext Transfer Protocol*

IDE *Integrated Development Environment*

JAR *Java ARchive*

JAX-WS *Java API for XML Web Services*

JDBC *Java Database Connectivity*

JPA *Java Persistence API*

JSP *JavaServer Pages*

JSTL *JavaServer Pages Standard Tag Library*

JVM *Java Virtual Machine*

LDM *Lattix Dependency Manager*

LOC *Lines Of Code*

MVC *Model-View-Controller*

POJO *Plain and Old Java Object*

RA Restrição Arquitetural

RAM *Random Access Memory*

REQ Requisito

SAVE *Software Architecture Visualization and Evaluation*

SCL *Structural Constraint Language*

SERPRO Serviço Federal de Processamento de Dados

SGP Sistema de Gestão de Pessoas

SQL *Structured Query Language*

Capítulo 1

Introdução

Este capítulo está organizado como descrito a seguir. A Seção 1.1 apresenta uma visão geral do problema tratado na dissertação. A Seção 1.3 expõe brevemente a solução proposta e, por fim, a Seção 1.4 apresenta a estrutura desta dissertação.

1.1 Visão Geral do Problema

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. Isso inclui como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir [GS96, Fow02]. Apesar de sua inquestionável importância, a arquitetura documentada de um sistema – se disponível – geralmente não reflete a sua implementação atual [KC99, KMNL06, SAG+06, MNS01, KP07]. Na prática, desvios em relação à arquitetura planejada são comuns, devido ao desconhecimento por parte dos desenvolvedores, requisitos conflitantes, dificuldades técnicas, etc [KP07]. Ainda mais importante, tais desvios geralmente não são capturados e resolvidos, levando aos fenômenos conhecidos como erosão e desvio arquitetural [PW92].

Apesar das pesquisas constantes na área de arquitetura de software e das diversas propostas de soluções no sentido de evitar o fenômeno de erosão arquitetural, ainda é pouco comum que artefatos arquiteturais sejam mantidos em sincronia com os requisitos do sistema e sua implementação [GB02, MEG03]. O processo de erosão arquitetural faz com que os benefícios proporcionados por um projeto arquitetural (manutenibilidade, reusabilidade, escalabilidade, portabilidade, etc) sejam anulados. Assim, verificação de conformação arquitetural é uma importante técnica para evitar o processo de erosão [BCK03, GZ05]. Define-se conformação arquitetural como uma medida do grau de aderência da arquitetura implementada em código fonte de um sistema em relação à sua

arquitetura planejada [KP07].

A princípio, soluções para verificação de conformação arquitetural podem ser classificadas em duas principais linhas de atuação: as que se baseiam em técnicas de análise estática e as que se baseiam em técnicas de análise dinâmica [Bal99, HLL04, YGS+04, JR97]. Cada uma delas tem suas vantagens e desvantagens. Por exemplo, as que se baseiam em técnicas de análise estática são não-invasivas, não requerem execução do sistema nem acesso à base de dados e cobrem propriedades estruturais (isto é, baseadas em código fonte). Por outro lado, as que se baseiam em técnicas de análise dinâmica requerem a execução do sistema, acesso à base de dados e cobrem somente os caminhos acessados nas execuções do sistema. Contudo, são capazes de lidar com sistemas cuja arquitetura se altera em tempo de execução e não impõem qualquer tipo de restrição na implementação do sistema.

1.2 Objetivos

Esta dissertação tem como objetivo principal prover uma solução para verificação de conformação arquitetural baseada em técnicas de análise estática. Dentre as soluções existentes nessa categoria, podem ser mencionadas as ferramentas LDM [SJSJ05], .QL [MVH+07] e SAVE [KMNL06], cada uma delas baseada em uma técnica de conformação diferente. LDM é uma ferramenta baseada no conceito de matrizes de dependência estrutural (*Dependency Structure Matrixes* ou DSM) [SGCH01]. Contudo, seu modelo de definição de restrições de dependência possui um nível de granularidade considerado insuficiente. Já .QL é uma ferramenta baseada em uma linguagem de consulta em código fonte. Possui alto poder de expressão, contudo sua sintaxe é relativamente complexa. Por fim, SAVE (*Software Architecture Visualization and Evaluation*) é uma ferramenta baseada em modelos de reflexão (*reflexion models*) [MNS95, MNS01]. No entanto, seu modelo de granularidade não permite restringir todos os tipos de dependência que podem ser estabelecidos em sistemas orientados a objetos. Além disso, essa ferramenta não provê conformação arquitetural por construção, ou seja, violações na arquitetura planejada não são detectadas logo que são implementadas no código fonte.

Além dessas ferramentas, existem outras linhas de pesquisa na área de conformação arquitetural baseadas, por exemplo, em linguagens de restrições lógicas [HH06, HHR04, EKKM08, MKPW06, MK06] e linguagens de descrição arquitetural (ADLs) [MT00, AG97, MDEK95, LKA+95, LV95, GMW97, ACN02]. Linguagens de restrições lógicas são baseadas em notações sobrecarregadas que tornam sua escrita e posterior manutenção uma tarefa complexa. Além disso, elas possuem ferramentas de suporte que são menos maduras e menos estáveis. Por outro lado, ADLs permitem expressar o comportamento arquite-

tural e a estrutura de um sistema de software em uma linguagem declarativa e abstrata. Contudo, ADLs requerem a utilização de compiladores específicos e podem requerer a extensão de linguagens de programação.

Pretende-se também levantar alguns requisitos considerados indispensáveis a uma nova solução para verificação de conformação arquitetural. Dentre alguns dos possíveis requisitos podem ser citados: processo bem definido, linguagem simples com alto poder de expressão, compatibilidade com linguagens atuais, detecção imediata de violações, entre outros.

Assim, o objetivo principal desta dissertação de mestrado é investigar uma solução para conformação arquitetural centrada na observação de que dependências inter-modulares impróprias são uma fonte importante de violações arquiteturais e, portanto, contribuem para o processo de erosão arquitetural [SJSJ05]. 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 na hierarquia), de tal forma que M_i somente pode utilizar serviços providos pelo módulo M_{i-1}, $i > 0$. Assim, o estabelecimento de qualquer dependência nesse sistema que viole essa regra está, de fato, violando sua arquitetura.

1.3 Visão Geral da Solução Proposta

Linguagens de programação normalmente suportam ocultamento de informação por meio de modificadores de visibilidade (tais como `public`, `private` e `protected`). Entretanto, elas não proveem meios para restringir dependências inter-modulares. Na prática, qualquer serviço público provido por um módulo (ou classe) M pode ser utilizado por qualquer outro módulo do sistema. Diante disso, propõe-se nesta dissertação uma linguagem de restrição de dependência que permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidas em um dado sistema, isto é, uma linguagem que permite definir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema.

Mais especificamente, foi projetada e implementada uma linguagem de domínio específico e estaticamente verificável denominada DCL (*Dependency Constraint Language)* [TV08a, TV08b, TV10]. Seu projeto, além de ter sido guiado por uma série de requisitos relevantes, perseguiu atributos como simplicidade e objetividade para fornecer uma sintaxe clara e auto-explicativa na definição de restrições estruturais entre módulos. DCL utiliza um modelo de granularidade fina para especificação de dependências estruturais comuns em sistemas orientados a objetos. Mais especificamente, esse modelo permite a definição de dependências originadas a partir do acesso a atributos e métodos, declaração

de variáveis, criação de objetos, extensão de classes, implementação de interfaces, ativação de exceções e uso de anotações. Pode-se definir, por exemplo, que serviços de persistência somente podem ser chamados a partir de classes da camada de modelo, que classes da camada de visão não podem depender de classes da camada de modelo, etc.

Com o intuito de validar a aplicabilidade da solução proposta, foi implementado um protótipo de uma ferramenta, chamada `dclcheck`, que verifica se o código fonte (isto é, a arquitetura concreta de um sistema) respeita restrições de dependência definidas em DCL. Essa ferramenta é baseada em técnicas de análise estática, isto é, ela não causa qualquer *overhead* durante a execução dos sistemas. A ferramenta também é não-invasiva, uma vez que não depende do código fonte para efetuar a verificação das restrições de dependência (que é realizada diretamente sobre os *bytecodes*[1]) nem realiza a instrumentação de código intermediário. Além disso, `dclcheck` provê conformação arquitetural por construção.

A fim de demonstrar a aplicabilidade da solução proposta em sistemas reais, descreve-se uma experiência de aplicação da linguagem DCL e da ferramenta `dclcheck` 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 mais de 12 mil empregados. Para isso, foram especificadas restrições de dependência em DCL para três diferentes versões do sistema, no sentido de garantir que sua implementação segue a sua arquitetura planejada. Em seguida, foi aplicada a ferramenta `dclcheck` em cada uma dessas versões. Como resultado, foi observado, por exemplo, que de um subconjunto de 28 restrições de dependência definidas para a terceira versão analisada do sistema – que possui cerca de 240 mil linhas de código e mais de 2.300 classes – 19 restrições foram violadas em pelo menos um ponto do sistema. Ao todo, 245 classes com violações das restrições de dependência desse subconjunto foram detectadas nessa terceira versão. 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.

1.4 Estrutura da Dissertação

O restante desta dissertação está organizado da seguinte maneira:

- No Capítulo 2, utilizando como base um sistema motivador chamado `myAppointments`, são comparadas e avaliadas algumas das principais ferramentas que podem

[1] *Bytecode* é o código intermediário gerado pelo compilador Java a ser interpretado pela JVM.

ser utilizadas para verificação de conformação arquitetural. Além disso, é realizada uma descrição de algumas outras linhas de pesquisa, como linguagens de restrições lógicas e ADLs. Por fim, esse capítulo levanta requisitos relevantes em soluções para verificação de conformação arquitetural;

- No Capítulo 3, propõe-se uma nova solução para verificação de conformação arquitetural que visa atender aos requisitos levantados no Capítulo 2. Assim, é apresentada uma visão geral da solução proposta, a linguagem DCL, alguns exemplos práticos de sua utilização (inclusive no sistema `myAppointments`) e, por fim, apresenta-se a ferramenta `dclcheck`;

- No Capítulo 4, descreve-se a aplicação da linguagem DCL e da ferramenta `dclcheck` em um sistema real de grande porte, chamado SGP, que é utilizado pelo Serviço Federal de Processamento de Dados (SERPRO) para gestão de seus empregados. Além de uma breve descrição do sistema, é apresentada a metodologia seguida no estudo de caso, o levantamento das restrições de dependência, os resultados obtidos e uma análise crítica desses resultados;

- Por fim, no Capítulo 5, são apresentadas as considerações finais deste estudo, ressaltando a solução proposta, suas melhorias em relação a outras abordagens, as contribuições do trabalho e propostas de trabalhos futuros.

Capítulo 2

Conformação Arquitetural

Um problema recorrente enfrentado por arquitetos de software é o de garantir que um sistema seja implementado de acordo com a sua arquitetura planejada [MK88, KMNL06, KP07]. Durante a implementação e evolução de um sistema, é comum observar desvios em relação à arquitetura inicialmente definida, devido ao desconhecimento por parte dos desenvolvedores, requisitos conflitantes, dificuldades técnicas, prazos de entrega reduzidos, etc. Mais importante, tais desvios tendem a acumular com o tempo, levando aos fenômenos conhecidos como desvio e erosão arquitetural [PW92].

Neste capítulo são comparadas três ferramentas que podem ser utilizadas para garantir que a implementação de um sistema atende à sua arquitetura planejada. Tais ferramentas proveem meios para detectar desvios em relação à arquitetura planejada de um sistema e, portanto, são recursos importantes para previnir erosão arquitetural. Mais especificamente, foram comparadas as seguintes ferramentas:

- LDM (*Lattix Dependency Manager*), uma ferramenta para conformação e gerenciamento arquitetural baseada no conceito de matrizes de dependência estrutural (*Dependency Structure Matrixes* ou DSM) [SJSJ05]. DSM são matrizes de adjacência utilizadas para representar dependências entre módulos de um sistema. LDM também suporta o conceito de regras de projeto (*design rules*), que podem ser utilizadas para definir dependências que violam a arquitetura planejada de um sistema.

- SemmleCode .QL (*Query Language*), uma linguagem de consulta em código fonte inspirada na sintaxe da linguagem SQL [MVH+07]. .QL suporta várias tarefas de análise em código fonte, como procura por erros, cálculo de métricas de engenharia de software, verificação de convenções de código, busca e navegação no código fonte, etc. Neste capítulo, será abordado o uso de .QL como uma ferramenta de

conformação arquitetural.

- SAVE (*Software Architecture Visualization and Evaluation*), uma ferramenta para avaliação estática de arquiteturas de software [KMNL06]. Baseada nos princípios de modelos de reflexão (*reflexion models*) [MNS95], SAVE compara um modelo arquitetural (isto é, a arquitetura planejada de um sistema) com o modelo de código fonte (isto é, a arquitetura implementada desse sistema). Como resultado dessa comparação, a ferramenta destaca relações convergentes, divergentes e ausentes entre os dois modelos.

Os seguintes critérios conduziram a seleção das ferramentas mencionadas:

1. As três ferramentas proveem suporte a técnicas representativas para prevenir erosão arquitetural, incluindo matrizes de dependência estrutural (DSM), linguagens de consulta e modelos de reflexão;

2. As três ferramentas proveem verificação de conformação arquitetural para sistemas implementados em Java, uma linguagem de programação largamente utilizada atualmente na construção dos mais diversos tipos de sistemas computacionais;

3. As três ferramentas representam sistemas maduros e estáveis, que possuem *plug-ins* para a IDE Eclipse.

Para comparar e avaliar as ferramentas consideradas neste capítulo, será utilizado um sistema motivador chamado `myAppointments`, que implementa um sistema simples de gerenciamento de informações pessoais com funções para agendar, atualizar e remover compromissos. `myAppointments` foi implementado em Java seguindo o padrão arquitetural MVC (*Model-View-Controller*) [Fow02, KP88]. Além disso, seis restrições arquiteturais foram identificadas como mandatórias, de acordo com a arquitetura planejada desse sistema. As ferramentas LDM, .QL e SAVE foram então utilizadas para garantir que tais restrições sejam sempre seguidas pela implementação do sistema.

O restante deste capítulo está organizado como a seguir. A Seção 2.1 descreve o sistema motivador que guiará a avaliação das ferramentas de conformação arquitetural. As Seções 2.2, 2.3 e 2.4 descrevem, respectivamente, como as ferramentas LDM, .QL e SAVE podem ser utilizadas para verificar se a arquitetura planejada do sistema motivador é mesmo seguida pela sua implementação. A Seção 2.5 aborda outras linhas de pesquisa sobre conformação de arquiteturas de software, como linguagens de descrição arquitetural (*Architecture Description Languages* ou ADLs) e linguagens de restrições lógicas. E, por fim, a Seção 2.6 apresenta algumas considerações finais.

2.1 Sistema Motivador

myAppointments é um sistema simples de gerenciamento de informações pessoais implementado exclusivamente para avaliação das ferramentas de conformação arquitetural consideradas neste capítulo. A implementação do sistema myAppointments foi realizada de forma independente e autônoma por um pesquisador, com vasta experiência em orientação por objetos, vinculado ao Grupo de Engenharia de Software da PUC Minas. Basicamente, os requisitos do sistema foram informados a esse pesquisador, bem como a arquitetura de software que ele deveria seguir (fundamentalmente, baseada no padrão MVC). Essa estratégia teve como objetivo principal evitar que a implementação do sistema pudesse adotar padrões e estruturas que favorecessem uma determinada ferramenta em detrimento de outra. A versão atual do sistema possui 1.215 LOC, 16 classes e três interfaces. A Figura 2.1 ilustra a tela principal do sistema. Basicamente, myAppointments permite aos usuários criar, pesquisar, atualizar e remover compromissos.

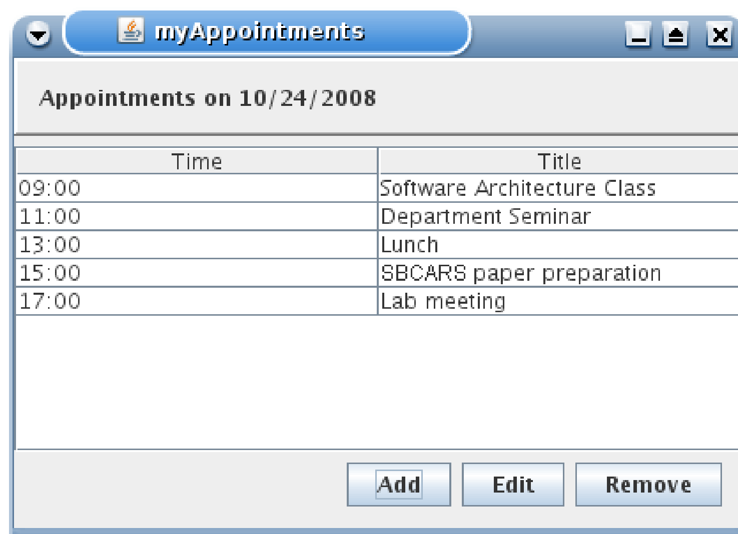


Figura 2.1: Tela principal do myAppointments

Como pode ser observado na Figura 2.2, a arquitetura do sistema segue o padrão MVC, largamente utilizado na construção de aplicações com interfaces gráficas [Fow02, KP88]. O padrão MVC promove uma divisão clara entre os objetos da Visão e do Modelo, sendo que os primeiros são usualmente associados a componentes GUI, tais como *Frames*, *Buttons*, *TextFields*, etc. Em sistemas que utilizam o padrão MVC, objetos de modelo são completamente independentes de qualquer *framework* para contrução de interfaces gráficas. De fato, as interações entre o Modelo e a Visão são mediadas por objetos da camada de Controle. Particularmente, na implementação do myAppointments, o Modelo inclui Obje-

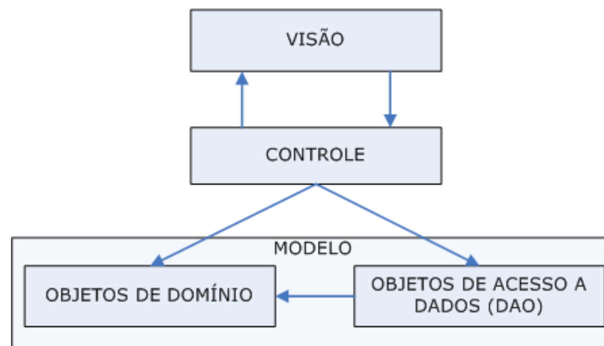


Figura 2.2: Arquitetura do `myAppointments`

tos de Domínio (*Domain Objects*), que representam entidades tais como Compromissos e Objetos de Acesso a Dados (*Data Access Objects* ou DAOs), que encapsulam o *framework* de persistência subjacente.

Restrições Arquiteturais: O pesquisador, além de implementar o sistema motivador, definiu as seguintes restrições arquiteturais:

- **(RA1)** Para implementar os componentes da interface gráfica, `myAppointments` deve utilizar AWT e Swing. Além disso, somente a camada de Visão pode depender dos componentes e serviços providos por essas duas APIs.

- **(RA2)** Persistência deve ser implementada de modo que somente objetos DAO da camada de Modelo dependam de serviços SQL.

- **(RA3)** A camada de Visão somente pode depender dos serviços providos por ela mesma, pelas APIs AWT e Swing, pela camada de Controle e pelo pacote `util`, que inclui classes utilitárias para manipular datas e *strings*. Em resumo, componentes da Visão não podem acessar diretamente o Modelo.

- **(RA4)** Objetos de Domínio não devem depender de DAOs nem de qualquer objeto das camadas de Visão ou de Controle, isto é, Objetos de Domínio devem representar *Plain and Old Java Objects* (POJOs).

- **(RA5)** Classes DAO somente devem depender de Objetos de Domínio, de classes utilitárias e de serviços SQL.

- **(RA6)** Classes do pacote `util` não devem depender de nenhuma classe específica do sistema `myAppointments` (isto é, elas não devem depender de classes do Modelo, do Controle e da Visão).

Adicionalmente, as seguintes convenções de nomes e subtipos devem ser seguidas pela implementação do sistema motivador:

- Classes DAO devem possuir o sufixo `DAO`;
- Classes da Visão devem estender a classe abstrata `View`;
- Classes de Controle devem implementar a interface `IController`.

Nas restrições acima, o conceito de dependência inclui qualquer tipo de relacionamento que pode ser estabelecido entre classes na linguagem Java. Mais especificamente, considera-se que a classe `A` depende da classe (ou interface) `B` se pelo menos uma das seguintes condições ocorrer:

- `A` declara atributos, parâmetros formais ou variáveis locais do tipo `B`;
- `A` invoca métodos definidos em `B`;
- `A` cria objetos do tipo `B`;
- `A` estende (ou implementa) o tipo `B`;
- métodos de `A` ativam exceções do tipo `B`.

2.2 LDM

Matrizes de dependência estrutural (DSMs) foram inicialmente propostas por Baldwin e Clark para demonstrar a importância de princípios de projeto modular na indústria de hardware [BC99]. Após isso, Sullivan *et al.* demostraram que o conceito de DSMs também pode ser utilizado no projeto de software [SGCH01].

Neste estudo, serão utilizadas DSMs geradas pela ferramenta LDM[1] (*Lattix Dependency Manager*), versão 4.8 [SJSJ05]. LDM é uma ferramenta de conformação e visualização arquitetural que utiliza DSMs para representar e gerenciar dependências inter-classes em sistemas orientados a objetos. Conforme implementado na ferramenta LDM, uma DSM é uma matriz quadrada cujas linhas e colunas são classes de um sistema orientado a objetos. Conforme ilustrado na Figura 2.3, um `x` na linha referente à classe `A` e na coluna referente à classe `B` denota que a classe `B` depende da classe `A`, isto é, existem referências explícitas em `B` para elementos sintáticos de `A`. Uma outra possibilidade é representar na célula `(A,B)` o número de referências que `B` contém para `A`.

[1]Ferramenta disponível em: `http://www.lattix.com`.

		1	2	3
Class A	1	.	X	
Class B	2		.	
Class C	3			.

Figura 2.3: Exemplo de uma DSM

O objetivo principal da ferramenta LDM é oferecer aos arquitetos uma ferramenta gráfica que permita revelar padrões arquiteturais e detectar dependências que possam indicar violações arquiteturais ou mesmo um projeto de software deficiente. Para esse propósito, LDM automaticamente extrai a DSM do código fonte de sistemas existentes utilizando técnicas de análise estática.

Para auxiliar os arquitetos a descobrir e raciocionar sobre estilos arquiteturais, LDM implementa um algoritmo de reordenação (ou particionamento). Basicamente, esse algoritmo decide a ordem de apresentação das linhas em uma DSM, iniciando pelos pacotes que proveem menos serviços e finalizando com os pacotes que são mais utilizados pelos outros pacotes. Esse algoritmo também agrupa pacotes que são mutualmente dependentes.

Para prover conformação arquitetural, LDM inclui uma linguagem simples para declarar regras de projeto que devem ser seguidas pela implementação do sistema alvo. Basicamente, regras de projeto possuem duas formas: `A can-use B` e `A cannot-use B`, indicando que classes do conjunto `A` podem (ou não podem) depender das classes do conjunto `B`. Violações em regras de projeto são visualmente exibidas na própria DSM extraída, com o objetivo de alertar sobre possíveis erosões arquiteturais.

2.2.1 Utilizando LDM

A Figura 2.4 ilustra a DSM – como gerada pela ferramenta LDM – do sistema motivador. Como pode-se observar nessa figura, a DSM apresentada claramente revela o padrão arquitetural subjacente ao `myAppointments`. Por exemplo, observando a primeira coluna, pode-se verificar que a camada de Visão somente utiliza serviços providos pelo Controle (célula [2,1] da matriz) e pelo pacote `util` (célula [7,1]), como prescrito pela restrição arquitetural RA3 (definida na Seção 2.1). Ainda nessa figura, o pacote `model` foi expandido. Desse modo, é possível notar que o objeto de domínio `Appointment` (coluna 6) não depende de DAOs, nem de classes das camadas de Controle e Visão, como prescrito pela RA4.

$root		1 view	2 controller	3 AbstractAgendaDAO	4 AgendaDAO	5 DAOCommand	6 Appointment	7 util
myappointments	view 1	.	11					
	controller 2	4	.					
model	AbstractAgendaDAO 3		4	.	1			
	AgendaDAO 4		2		.			
	DAOCommand 5		1		1	.		
	Appointment 6		5	1	1		.	
	util 7	1	4		2		1	.

Figura 2.4: DSM do `myAppointments`

Conformação Arquitetural: Para prover meios para detectar erosões na arquitetura planejada do sistema motivador, foram definidas regras de projeto para cada uma das restrições arquiteturais descritas na Seção 2.1. Por exemplo, a seguinte regra de projeto especifica que somente a Visão pode utilizar serviços providos pela API AWT e Swing (como prescrito pela RA1):

```
1: $root CANNOT-USE java.awt
2: $root CANNOT-USE javax.swing
3: view CAN-USE java.awt
4: view CAN-USE javax.swing
```

Primeiramente, essa regra especifica que `$root`, que denota todas as classes e interfaces do sistema, não pode utilizar serviços providos pelos pacotes `awt` e `swing` da API de Java (linhas 1-2). Em seguida, são abertas exceções para as restrições anteriores, especificando que classes do pacote `view` podem utilizar serviços providos pelos pacotes `awt` e `swing` (linhas 3-4).

Como um outro exemplo, a seguinte regra de projeto implementa a RA5, que requer que as classes do pacote `util` não dependam de qualquer classe específica do sistema motivador:

```
1: util CANNOT-USE $root
2: util CAN-USE util
```

Primeiramente nessa regra, classes do pacote `util` são proibidas de utilizar qualquer outra classe do projeto (linha 1). Em seguida, sua utilização é permitida somente dentro do seu próprio pacote (linha 2).

O poder de expressão de regras de projeto não foi suficiente para expressar adequadamente as restrições arquiteturais RA3 e RA5. Por exemplo, RA3 prescreve que a Visão somente pode depender dos serviços providos pelo Controle. Quando mapeada para a implementação atual do sistema motivador, essa regra requer que a Visão somente possa depender de classes que implementam a interface `IController`. Entretanto, a linguagem de regras de projeto da ferramenta LDM não permite selecionar classes que implementam uma interface específica. Por essa razão, foi preciso manualmente encontrar as classes que implementam `IController` e criar regras específicas garantindo a essas classes acesso à Visão. As regras criadas para suportar RA3 são as seguintes:

```
1: view CANNOT-USE $root
2: view CAN-USE AppointmentController
3: view CAN-USE AgendaController
4: view CAN-USE util
```

Primeiramente, essa regra proíbe que o pacote `view` utilize qualquer classe do projeto (linha 1). Depois, duas exceções para essa regra geral são especificadas: permite-se ao pacote `view` utilizar os serviços providos pelas classes `AppointmentController` e `AgendaController` (linhas 2-3), que implementam a interface `IController`, como prescrito pela RA3. Além de requerer que os desenvolvedores manualmente encontrem as classes que implementam `IController`, essa regra de projeto é particularmente frágil em relação a evoluções no sistema motivador, pois necessitará de atualizações toda vez em que um novo subtipo de `IController` for criado.

Um problema similar ocorreu na RA5, que restringe dependências entre DAOs e outros tipos. Na implementação do `myAppointments`, classes DAO sempre possuem o sufixo `DAO`. Entretanto, LDM não suporta a seleção de classes utilizando expressões regulares e, por essa razão, foi necessário definir uma regra de projeto para cada DAO existente.

Uma outra recomendação importante ao se utilizar a ferramenta LDM é definir de forma cuidadosa a estrutura hierárquica dos pacotes do sistema. Isso se deve ao fato de a ferramenta utilizar essa estrutura para automaticamente extrair as matrizes de dependência do código fonte, utilizando técnicas convencionais de análise estática. Em outras palavras, é recomendado que a hierarquia de pacotes seja compatível com o nome dos componentes da arquitetura conceitual do sistema. Por exemplo, no sistema motivador, existem pacotes chamados `view`, `model`, etc.

2.3 .QL

.QL é uma linguagem de consulta em código fonte que provê suporte a uma ampla gama de tarefas de desenvolvimento de software, tais como verificação de convenções de código, procura por erros, cálculo de métricas de engenharia de software, detecção de oportunidades de refatoração, etc [MVH+07]. Apesar de .QL automatizar várias tarefas de desenvolvimento, este estudo concentra-se na utilização da linguagem para detectar violações na arquitetura planejada do sistema motivador. Em outras palavras, nosso objetivo é avaliar a adequação do uso de consultas em .QL para detectar possíveis violações das restrições arquiteturais descritas na Seção 2.1.

.QL é inspirada na linguagem SQL, o que torna sua sintaxe familiar para a maioria dos desenvolvedores. Entretanto, .QL inclui várias outras características que aumentam o seu poder de expressão para a consulta em código fonte. Por exemplo, a implementação interna da linguagem utiliza um *engine* Datalog – uma linguagem lógica mais restrita que o Prolog – para definir consultas recursivas ao longo de uma hierarquia de herança ou de um grafo de chamadas de métodos em sistemas orientados a objetos. Além disso, conceitos de orientação a objetos – tais como classes e herança – podem ser usados para estender a linguagem com novos predicados e construir bibliotecas de consultas. Finalmente, para melhorar seu desempenho e escalabilidade, a implementação de .QL utiliza um sistema de banco de dados relacional para armazenar relações entre elementos do código fonte. Por essa razão, consultas .QL são primeiramente traduzidas para Datalog a fim de serem otimizadas e, em seguida, são traduzidas para SQL.

SemmleCode .QL[2] é um *plug-in* para a IDE Eclipse que permite a execução de consultas .QL sobre sistemas em Java. Nesta dissertação, foi utilizada o Semmlecode Professional Edition, versão 1.0. Basicamente, a ferramenta inclui um editor de consultas cuja apresentação de seus resultados pode ser vista em forma de árvores, tabelas, gráficos, grafos e *warnings* reportados pelo ambiente de desenvolvimento. Além disso, conta com a definição de repositórios de consultas para armazenar, por exemplo, consultas realizadas de forma usual.

2.3.1 Utilizando .QL

Para ilustrar a utilização de .QL na definição de consultas de conformação arquitetural para o sistema motivador, primeiramente será demonstrada uma consulta que retorna as dependências entre os pacotes do sistema `myAppointments` e os pacotes `AWT`, `Swing` e `SQL` da API de Java:

[2] Ferramenta disponível em: `http://semmle.com`.

```
1: from RefType r1, RefType r2
2: where
3:   r1.fromSource() and depends(r1,r2) and
4:   (r2.fromSource() or isSwingApi(r2) or isSqlApi(r2))
5: select r1.getPackage(), r2.getPackage()
```

Como pode ser observado na Figura 2.5, essa consulta retorna todos os pares de pacotes (`r1`, `r2`) (linha 5) de `RefType` × `RefType` (linha 1) em que `r1` é um elemento de código fonte (isto é, um elemento interno do projeto), `r1` depende de `r2` (linha 3) e `r2` é um elemento de código fonte ou um componente dos pacotes `AWT`, `Swing` ou `SQL` (linha 4). Nessa consulta, `depends` é um predicado já existente que suporta exatamente a definição de dependência entre classes descrita na Seção 2.1. Além disso, `RefType` é uma classe .QL que representa referências em sistemas Java. Ela contém métodos (por exemplo, `getPackage()`) e predicados (por exemplo, `fromSource()`). Os predicados `isSwingApi` e `isSqlApi` foram definidos como:

```
 1: predicate isSwingApi(RefType r) {
 2:   r.getPackage().getName().matches("java.awt")    or
 3:   r.getPackage().getName().matches("java.awt.%")  or
 4:   r.getPackage().getName().matches("javax.swing") or
 5:   r.getPackage().getName().matches("javax.swing.%")
 6: }
 7: predicate isSqlApi(RefType r) {
 8:   r.getPackage().getName().matches("java.sql") or
 9:   r.getPackage().getName().matches("java.sql.%")
10: }
```

Quando compara-se nomes, o caracter `%` funciona da mesma maneira que o operador `like` em cláusulas `where` na linguagem SQL. Basicamente, permite-se que esse operador possa ser substituído por qualquer *string* de qualquer tamanho. Por exemplo, as classes `java.sql.Connection`, `java.sql.Statement` e qualquer outra classe cujo nome qualificado[3] inicie-se por `java.sql.` casam com o padrão `java.sql.%` (linha 9).

Convém mecionar que o grafo exibido na Figura 2.5 representa o padrão arquitetural subjacente ao `myAppointments`. Por exemplo, pode-se observar que somente o pacote `myappointments.view` utiliza as classes dos pacotes das APIs AWT e Swing (`javax.swing`, `javax.swing.text`, `javax.swing.border`, `javax.swing.table`, `java.awt`

[3]O nome qualificado de uma classe é o nome do pacote em que ela está localizada mais o seu nome. Por exemplo, o nome qualificado da classe `Serializable` do pacote `java.io` é `java.io.Serializable`.

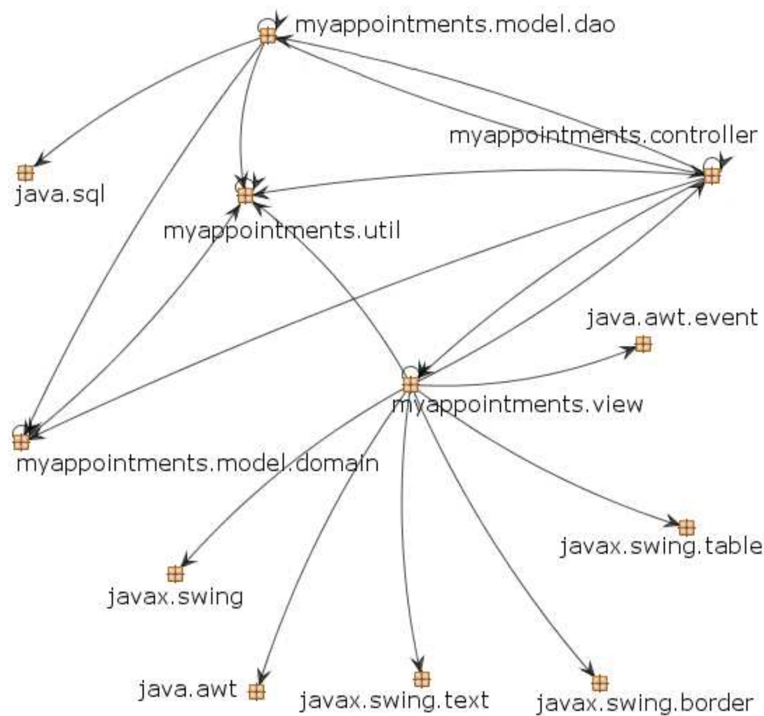


Figura 2.5: Grafo de dependências gerado pela ferramenta .QL

e `java.awt.event`), como prescrito pela restrição arquitetural RA1 (definida na Seção 2.1). Um outro exemplo seria o pacote `myappointments.util` que não depende de nenhuma classe específica do sistema, como prescrito pela RA6.

Conformação Arquitetural: Utilizando classes, métodos e predicados de .QL, foram definidas consultas para detectar violações das restrições arquiteturais definidas para o sistema `myAppointments`. Por exemplo, a seguinte consulta verifica se a restrição arquitetural RA1 é seguida:

```
1: from RefType r1, RefType r2
2: where
3:   r1.fromSource()
4:   and not(r1.getPackage().getName().matches("myappointments.view"))
5:   and depends(r1,r2) and isSwingApi(r2)
6: select r1 as Tipo,
7:   "RA1 violada: " + r1.getQualifiedName()
8:                   + " utilizando " + r2.getQualifiedName() as Violacao
```

Essa consulta verifica se existe um tipo `r1` que não seja parte do pacote `myappointments.view` (linhas 3-4) e que dependa de um outro tipo `r2` que seja parte do pacote

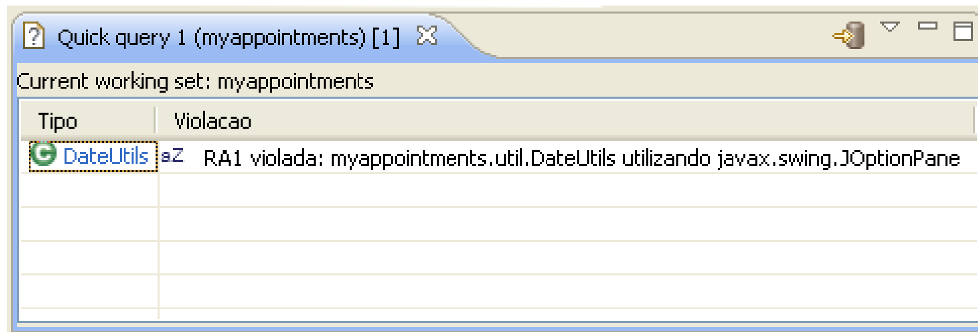


Figura 2.6: Janela exibindo violação da restrição RA1

AWT/Swing (linha 5). Caso exista alguma violação, a consulta retorna `r1` e uma *string* descrevendo a violação arquitetural (linhas 6-8). Para simular uma violação, foi manualmente inserida uma chamada a um método estático da classe `javax.swing.JOptionPane` a partir de um método da classe `DateUtils` (classe do pacote `util`). Como pode ser observado na Figura 2.6, .QL foi capaz de detectar essa violação. Convém ainda salientar que, se desejado, é possível acrescentar na mensagem de erro o nome do método da classe de origem, o nome do método invocado ou atributo acessado, a linha em que a violação ocorreu, etc.

De maneira similar, a restrição RA3 é definida como a seguir:

```
 1: from RefType view, RefType ref
 2: where
 3:   view.getPackage().getName().matches("myappointments.view")
 4:   and ref.fromSource()
 5:   and not (ref.getPackage().getName().matches("myappointments.view"))
 6:   and not isController(ref)
 7:   and not isUtil(ref)
 8:   and depends(view, ref)
 9: select view as Tipo_Visao,
10:   "RA3 violada: " + view.getQualifiedName()
11:                   +  " utilizando " + ref.getQualifiedName() as Violacao
```

Nessa consulta são utilizados os predicados `isController` e `isUtil`, definidos como a seguir:

```
12: predicate isController(RefType ref) {
13:   ref.getASupertype*().hasQualifiedName
14:    ("myappointments.controller","IController")
15: }
```

```
16: predicate isUtil(RefType ref) {
17:   ref.getPackage().getName().matches("myappointments.util")
18: }
```

Inicialmente, essa consulta verifica se elementos do pacote `myappointments.view` (linha 3) dependem de tipos que não pertençam a `view`, `util` e `controller` (linhas 4-8). O predicado `isController` utiliza o método `getASupertype()` definido na classe `RefType`. Esse método pode ser seguido pelos caracteres $*$ (nenhuma ou mais vezes) ou $+$ (uma ou mais vezes). Assim, as linhas 13-14 verificam se `ref` é um subtipo direto ou indireto de `IController`, como prescrito pela RA3.

A definição das outras restrições (RA4, RA5 e RA6) segue as mesmas idéias utilizadas nas consultas anteriores e, por isso, não são apresentadas neste estudo.

2.4 SAVE

Desenvolvida pelo Instituto Fraunhofer IESE, SAVE[4] (*Software Architecture Visualization and Evaluation*) [KMNL06] é uma ferramenta de conformação arquitetural centrada no conceito de modelo de reflexão de software proposto por Murphy *et al.* [MNS95]. Nesta dissertação, foi utilizada a versão acadêmica do sistema SAVE, versão 1.4.3. Segundo essa abordagem, arquitetos de software devem primeiramente definir um modelo de alto nível que represente a arquitetura planejada ou desejada de um sistema. Basicamente, esse modelo inclui os principais componentes do sistema e as relações entre eles (invocações, instanciações, herança, etc). Além disso, arquitetos devem também definir um mapeamento entre o modelo de código fonte[5] (isto é, a arquitetura implementada do sistema) e o modelo de alto nível proposto. Assim, uma ferramenta baseada em modelo de reflexão, como SAVE, pode ser utilizada para classificar relações entre componentes como:

- Convergente: quando uma relação prescrita no modelo de alto nível é seguida pelo código fonte;
- Divergente: quando uma relação não prescrita no modelo de alto nível existe no código fonte;
- Ausente: quando uma relação prescrita no modelo de alto nível não existe no código fonte.

[4]Neste trabalho, foi utilizada a versão acadêmica do SAVE. Essa versão não está publicamente disponível para *download*, tendo sido obtida por meio de uma solicitação encaminhada ao Fraunhofer IESE.

[5]Nesta dissertação, foi adotada a nomenclatura *modelo de código fonte* proposta por Knodel *et al.* para denotar a arquitetura implementada de um sistema [KMNL06]. Essa arquitetura, por sua vez, também é chamada de *modelo de implementação*.

2.4.1 Utilizando SAVE

SAVE inclui um editor gráfico que permite a arquitetos construir modelos de alto nível. Isso é uma característica diferencial de ferramentas baseadas em modelos de reflexão, uma vez que elas explicitamente delegam aos arquitetos a construção do modelo idealizado de seus sistemas, ao invés de automaticamente inferir tal modelo a partir do código fonte (como ocorre, por exemplo, com ferramentas baseadas em DSM). A vantagem nesse caso é que os arquitetos podem construir um modelo compatível com as suas visões do sistema. Ou seja, podem eliminar da verificação de conformação detalhes que não possuem relevância arquitetural.

Ao construir o modelo de alto nível, SAVE requer que os arquitetos definam as dependências aceitáveis entre os componentes arquiteturais propostos. Para esse propósito, eles devem incluir uma seta do componente de origem para o de destino. Além disso, três importantes características podem ser aplicadas ao se criar dependências: (i) múltiplas dependências entre os mesmos componentes podem ser fundidas em uma única seta; (ii) arquitetos podem discriminar o tipo da dependência (invocação, instanciação, herança, etc) ou simplemente especificar uma dependência genérica; (iii) setas podem ser ocultadas para não "poluir" o modelo.

Conformação Arquitetural: Para avaliar a arquitetura do sistema `myAppointments` utilizando a versão acadêmica da ferramenta SAVE, as seguintes tarefas foram realizadas:

1. Definição dos componentes do modelo de alto nível: Primeiramente, foi criado o modelo de alto nível descrevendo a arquitetura planejada do sistema, como ilustrado na Figura 2.7. Esse modelo foi dividido em duas partes: a parte da esquerda inclui pacotes da API de Java e a parte da direita inclui os pacotes do sistema `myAppointments`. Somente os pacotes `java.awt`, `javax.swing` e `java.sql` foram incluídos no modelo proposto, uma vez que os outros pacotes da API de Java não eram relevantes do ponto de vista arquitetural. Por outro lado, os componentes do `myAppointments` incluem os pacotes `view`, `controller`, `model` e `util`.

2. Definição de restrições arquiteturais: Para capturar as restrições arquiteturais propostas, setas representando as dependências aceitáveis foram criadas. Por exemplo, para capturar a RA1, setas foram criadas da `view` para `javax.swing` e `java.awt`. Para capturar a RA2, setas foram criadas do `dao` e `BD` para `java.sql`. Para capturar RA3, uma seta da `view` para `IController` foi criada. `IController` é um subcomponente de `Controller` contendo somente classes que implementam a interface `IController`, conforme prescrito pela RA3. Além disso, uma outra seta foi

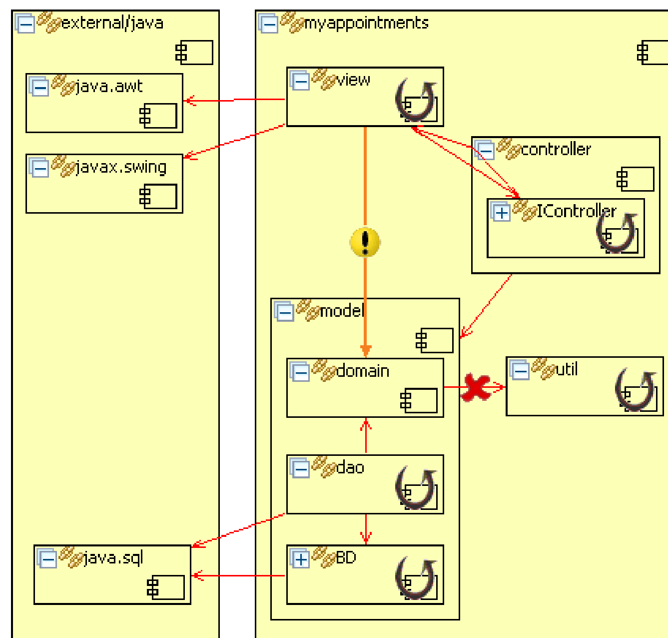


Figura 2.7: Modelo de reflexão para o sistema motivador

inserida da `view` para `util`, contudo essa seta foi ocultada para não "poluir" o modelo. Para capturar a RA5, setas do `dao` para `domain`, `BD` e `util` foram criadas (essa última foi também ocultada). Como as restrições RA4 e RA6 somente prescrevem dependências *must not* (aquelas que não devem ocorrer), elas não necessitam ser representadas no modelo de alto nível. Em outras palavras, os arquitetos somente definem as relações aceitáveis no modelo; as relações não definidas explicitamente são automaticamente interpretadas como inaceitáveis.

3. Geração do modelo de código fonte: esse modelo inclui todos os componentes implementados no código fonte e todas as dependências detectadas entre eles (isto é, esse modelo pode incluir dependências que não são relevantes do ponto de vista arquitetural). O modelo de código fonte é automaticamente extraído pela ferramenta SAVE, utilizando técnicas de análise estática. A Figura 2.8 exibe o modelo de código fonte do sistema `myAppointments`, organizado de modo a prover uma melhor visualização dos módulos mais significativos do ponto de vista arquitetural.

4. Mapeamento: Após a geração do modelo de código fonte, os arquitetos devem mapear os componentes extraídos para os componentes do modelo de alto nível. Para realizar esse mapeamento, uma lista de componentes de ambos os modelos é apresentada aos arquitetos. Então, eles devem manualmente associar cada componente de alto nível aos seus componentes correspondentes no modelo de código fonte.

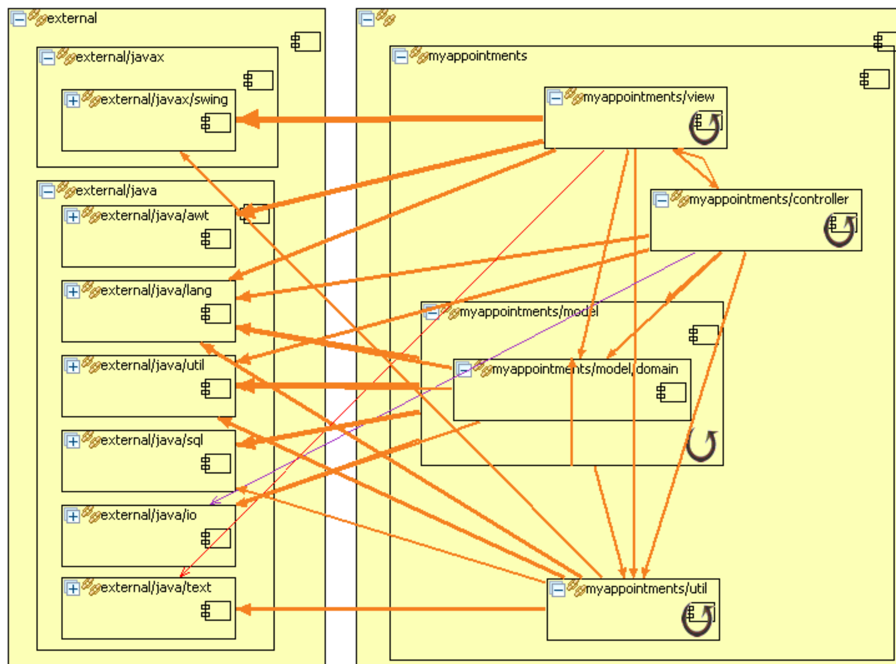


Figura 2.8: Modelo de código fonte do sistema motivador

5. Modelo de reflexão: Nesse último passo, arquitetos requisitam à ferramenta para computar o modelo de reflexão com o intuito de destacar as relações divergentes e ausentes. Tais relações são automaticamente detectadas pelo SAVE a partir da comparação do modelo de alto nível com o modelo de código fonte.

Para simular divergências e ausências no modelo de reflexão computado, foram inseridos dois desvios arquiteturais na implementação do sistema motivador. Primeiramente, foram removidos todos os acessos do `domain` para serviços providos pelo pacote `util`. Como pode ser observado na Figura 2.7, essa remoção resultou em uma ausência no modelo de reflexão, indicada por um `x`. Depois, foi inserido no código fonte da `view` um acesso direto ao `model`. Isso resultou em uma relação divergente, indicada por um ponto de exclamação no modelo de reflexão computado.

A ferramenta SAVE reconstrói o modelo de código fonte sempre que uma nova verificação de conformação arquitetural é requisitada (isto é, sempre que os arquitetos requisitam a computação de um novo modelo de reflexão). Como esse modelo é gerado automaticamente, esse requisito não representa um problema para os arquitetos. Entretanto, eles são responsáveis por manter o mapeamento entre os modelos de código fonte e de alto nível atualizado. Por exemplo, toda vez que classes que impactam o modelo de alto nível são inseridas ou removidas do código fonte, os arquitetos devem atualizar o mapeamento de acordo. Para auxiliá-los nessa tarefa, SAVE suporta a seleção de classes

utilizando expressões regulares. Entretanto, não é possível mapear um componente de alto nível para subtipos de um dado tipo (por exemplo, para indicar que `IController` deve conter as classes do código fonte que implementam essa interface).

2.5 Outros Trabalhos

Na primeira parte deste capítulo, foram abordadas três ferramentas estáveis e que utilizam técnicas representativas de conformação arquitetural. Entretanto, existem outras técnicas e linguagens que também podem ser utilizadas para verificação de conformação arquitetural. A seguir, algumas dessas outras técnicas são descritas.

Linguagens de restrições lógicas: SCL (*Structural Constraint Language*) [HH06] – e sua predecessora FCL (*Framework Constraint Language*) [HHR04] – são linguagens de restrições lógicas de primeira ordem que permitem aos desenvolvedores expressar intensões arquiteturais e de projeto em termos de restrições sobre a estrutura estática de sistemas orientados a objetos. Basicamente, especificações SCL consistem em uma sequência de declarações e fórmulas lógicas. Fórmulas possuem uma semântica baseada em lógica de primeira ordem e podem fazer uso de um vasto conjunto de funções para obter informações sobre a estrutura sintática de sistemas orientados a objetos. SCL também inclui uma ferramenta de conformação que pode ser utilizada para verificar se o código fonte segue as restrições definidas.

LogEn é uma linguagem lógica de domínio específico para expressar dependências estruturais entre grupos lógicos de elementos do código fonte, chamados *ensembles* [EKKM08]. Restrições definidas em LogEn são estaticamente verificadas de forma contínua por uma ferramenta integrada ao processo de compilação incremental da IDE Eclipse. Como ela é uma linguagem lógica de domínio específico, os autores de LogEn afirmam que a linguagem possui uma sintaxe mais amigável que SCL. Por exemplo, a Figura 2.9 ilustra uma restrição LogEn que define que somente classes `Factory` podem criar objetos da classe `Product` e de suas subclasses. Como pode ser observado, mesmo sendo uma linguagem de domínio específico, a sintaxe não é tão simples.

Linguagens de restrições lógicas não foram abordadas em maior profundidade neste capítulo por duas razões: (i) elas são baseadas em notações complexas, que usualmente tornam a escrita e manutenção de restrições arquiteturais uma tarefa árdua para desenvolvedores com experiência somente em linguagens imperativas e orientadas a objetos; (ii) elas possuem ferramentas de suporte que são menos maduras e menos estáveis, o que pode prejudicar o emprego de tais linguagens em ambientes reais de desenvolvimento de

```
% Factory é um ensemble que inclui toda a classe org.foo.Factory
part_of(E, "Factory"):-
   type(ClassId, 'org.foo.Factory'),
   E= ClassId|method(E, _, ClassId, _, _)|field(E, _, ClassId, _).

% ProductCreation é um ensemble que inclui todos os construtores
% de todas as subclasses de org.foo.Product
part_of(E, "ProductCreation"):-
   type(ClassId,'org.foo.Product'),
   inherits(SubClassId, ClassId),
   method(E, SubclassId, '<init>', _, _).

% FactoryViolation é uma restrição que especifica que somente ao ensemble
% Factory é permitido criar instâncias de Product e de suas subclasses
violation(S, T, "FactoryViolation"):-
   part_of(T, "ProductCreation"),
   tnot(part_of(S, "Factory")).
```

Figura 2.9: Restrição LogEn especificando um padrão de projeto *Factory*

software.

Visão Intensional (*Intensional View*) é uma outra técnica baseada em restrições lógicas para descrever o modelo conceitual da estrutura de um sistema e verificar a consistência de tal modelo com o respectivo código fonte [MKPW06, MK06]. Uma visão intensional é um conjunto de elementos (classes ou métodos) estruturalmente similares. Os elementos de uma visão são definidos por meio de uma intensão, utilizando a meta-linguagem de programação Soul [MMW02]. A conformação de visões intensionais em relação ao código fonte é verificada utilizando intensões alternativas ou relações intensionais. Essa abordagem requer que intensões alternativas (isto é, múltiplas definições para uma mesma visão) representem exatamente o mesmo conjunto de entidades, mesmo após alterações no código fonte. Por outro lado, relações intensionais definem relações lógicas de primeira ordem entre visões intensionais que devem ser seguidas pelo código fonte. Visões intensionais não foram incluídas na comparação realizada neste capítulo porque sua implementação somente suporta sistemas desenvolvidos em Smalltalk. Além disso, como é uma linguagem baseada no paradigma lógico, Visão Intensional também utiliza-se de uma sintaxe e semântica mais complexas.

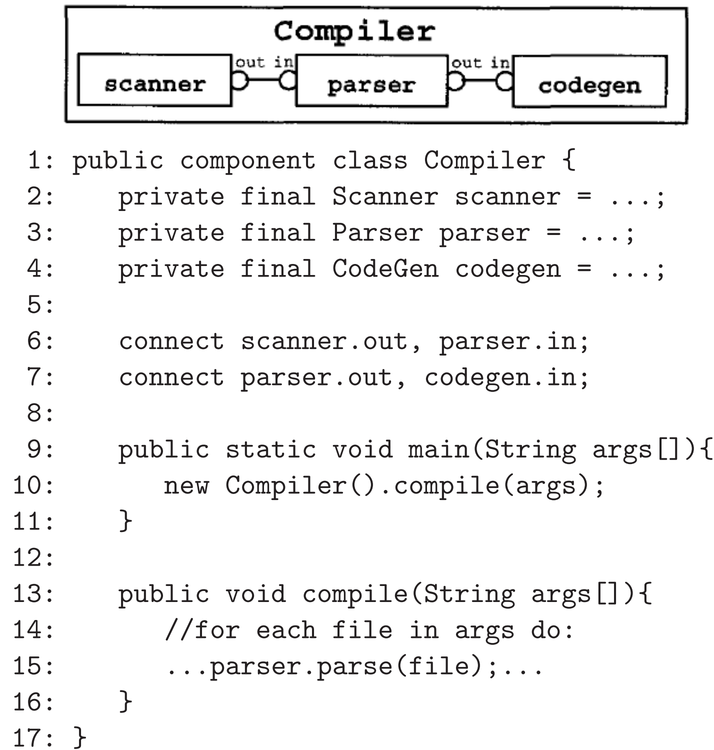
Linguagens de Descrição Arquitetural (ADLs): ADLs, como Darwin [MDEK95], Rapide [LKA+95, LV95], Wright [AG97] e ACME [GMW97], representam uma outra alternativa para prover conformação arquitetural por construção [MT00]. Tais linguagens permitem expressar o comportamento arquitetural e a estrutura de um sistema de software em uma linguagem declarativa e abstrata. Ferramentas de geração de código podem então ser utilizadas para mapear descrições arquiteturais para uma dada lingua-

gem de programação. Entretanto, tais abordagens normalmente requerem a utilização de compiladores baseados em ADLs para manter o código gerado sincronizado com a especificação arquitetural. Uma variante dessa abordagem defende a extensão de linguagens de programação com construções para modelagem arquitetural [ACN02]. Por exemplo, ArchJava estende a linguagem de programação Java para incorporar características arquiteturais e forçar integridade de comunicação, garantindo assim um código fonte sempre em conformidade com a arquitetura planejada. Para isso, ArchJava adiciona em Java construções para representar componentes, conexões e portas. Um componente é um tipo especial de objeto que se comunica com outros componentes através de portas previamente declaradas. Um exemplo de sua utilização é exibido na Figura 2.10, a qual representa a arquitetura de um compilador [GS93]. Nessa figura, o componente `Compiler` é composto pelos subcomponentes `Scanner`, `Parser` e `CodeGen`. Além disso, a porta `out` de um componente está conectada à porta `in` do próximo componente. Linguagens como ArchJava, na prática, requerem que desenvolvedores dominem uma linguagem de programação complementamente nova. Por exemplo, o componente `Compiler` se assemelha a uma classe Java, contudo é um `component class` (linha 1) que especifica as conexões permitidas entre seus subcomponentes (linha 6 e 7). Assim, ADLs não foram consideradas neste estudo devido ao seu foco em soluções compatíveis com linguagens de programação atuais.

2.6 Considerações Finais

Neste capítulo, foram comparadas algumas soluções para verificação de conformação arquitetural. Dentre essas soluções, as ferramentas LDM, .QL e SAVE receberam maior destaque, pois cada uma delas é baseada em uma técnica de conformação diferente. Além disso, elas representam ferramentas estáveis, de origem acadêmica e que atualmente estão inseridas no ambiente profissional. Para guiar a comparação, foi utilizado o sistema `myAppointments`, que segue um conhecido padrão arquitetural, o que consequentemente contribuiu para que as restrições arquiteturais consideradas representassem restrições tipicamente utilizadas em cenários reais de conformação arquitetural. A realização do estudo descrito neste capítulo teve como objetivo capturar os pontos fortes e as limitações de cada uma dessas soluções, visando o desenvolvimento de uma nova solução para verificação de conformação arquitetural. Os principais pontos fortes e limitações das soluções estudadas são resumidos a seguir:

LDM: De acordo com a avaliação realizada, matrizes de dependência representam um instrumento simples e, ao mesmo tempo poderoso, para visualizar e raciocionar sobre




```
 1: public component class Compiler {
 2:    private final Scanner scanner = ...;
 3:    private final Parser parser = ...;
 4:    private final CodeGen codegen = ...;
 5:
 6:    connect scanner.out, parser.in;
 7:    connect parser.out, codegen.in;
 8:
 9:    public static void main(String args[]){
10:       new Compiler().compile(args);
11:    }
12:
13:    public void compile(String args[]){
14:       //for each file in args do:
15:       ...parser.parse(file);...
16:    }
17: }
```

Figura 2.10: Modelagem e implementação de um componente Compilador em ArchJava

arquiteturas de software. Isso se justifica por duas razões. Primeiramente, DSM são estruturas inerentemente hierárquicas, permitindo aos arquitetos rapidamente expandir e contrair a estrutura interna de pacotes de seus sistemas, o que é um requisito crucial para manipular sistemas de grande porte. Portanto, DSMs possuem melhor escalabilidade que representações gráficas, como aquelas utilizadas nas ferramentas .QL e SAVE. Em segundo lugar, DSMs incluem algoritmos de reordenação que ajudam os desenvolvedores a descobrir e raciocionar sobre o padrão arquitetural subjacente a um sistema. Por outro lado, a linguagem de regras de projeto disponibilizada pela ferramenta LDM revelou-se insuficiente para expressar até mesmo restrições arquiteturais simples. Como mencionado na Seção 2.2.1, regras de projeto não permitem a especificação de restrições arquiteturais utilizando expressões regulares ou relacionamentos de subtipos. Por essa razão, não foi possível utilizar a ferramenta LDM de maneira conveniente para expressar as restrições RA2, RA3 e RA5 do sistema motivador. Além disso, regras de projeto são limitadas em relação aos tipos de dependência (`can-use` e `cannot-use`), isto é, não permitem restringir somente determinados tipos de dependência como, por exemplo, instanciação, invocação, etc.

.QL: De acordo com a avaliação realizada, .QL representa uma poderosa e relativamente simples linguagem de consulta em código fonte. O poder de expressão da linguagem deve-se a sua origem na linguagem Datalog, enquanto a sua relativa simplicidade se deve à sintaxe inspirada em SQL. Como resultado, foi possível utilizar .QL para definir todas as restrições arquiteturais propostas para o sistema `myAppointments`. Por outro lado, foram encontradas dificuldades para visualizar, navegar e raciocinar sobre representações arquiteturais geradas a partir de consultas em código fonte, quando comparadas, por exemplo, com DSM. Isso se deve ao fato da ausência de algoritmos de reordenação e de recursos para rapidamente expandir ou contrair elementos da arquitetura extraída.

SAVE: Das ferramentas avaliadas, SAVE é aquela que suporta o melhor e mais completo processo para verificação de conformação arquitetural [KMHM08]. Inicialmente, esse processo requer a criação de um modelo de alto nível da arquitetura planejada. Desse modo, arquitetos têm controle sobre a granularidade e o nível de abstração dos componentes que serão considerados. Por outro lado, arquitetos devem manualmente manter o mapeamento entre o modelo de alto nível e o modelo de código fonte atualizado. Considerando que isso pode ser uma tarefa que exige tempo, é provável que arquitetos somente apliquem a ferramenta quando tiverem versões estáveis de seus sistemas. Em outras palavras, SAVE não foi projetado para ser continuamente aplicado, por exemplo, sobre uma versão diária ou mesmo antes de submeter cada nova versão do sistema a um sistema de controle de versões. É importante destacar, no entanto, que para simplificar a tarefa de manter os modelos atualizados, SAVE inclui suporte a expressões regulares.

Linguagens de restrições lógicas: Linguagens de restrições lógicas, tais como SCL, FCL, LogEn e Visões Intensionais, são baseadas em notações sobrecarregadas que tornam sua escrita e posterior manutenção uma tarefa árdua, ainda mais para aqueles desenvolvedores que somente têm experiência com linguagens imperativas. Além do mais, possuem ferramentas que são menos maduras e menos estáveis, o que pode impedir a sua aplicação em projetos reais de desenvolvimento de software.

ADLs: ADLs permitem expressar o comportamento arquitetural e a estrutura de um sistema de software em uma linguagem declarativa e abstrata. Contudo, tais soluções requerem a utilização de compiladores baseados em ADLs para manter o código gerado sincronizado com a especificação arquitetural, o que consequentemente dificulta a sua adoção em ambientes industriais. Por outro lado, abordagens como o ArchJava, que estendem uma linguagem de programação com construções para modelagem arquitetural,

garantem um código fonte sempre em conformidade com a arquitetura planejada. Contudo, requerem que os desenvolvedores lidem com a variedade e complexidade dos novos elementos da linguagem.

Com base na experiência adquirida com o estudo descrito neste capítulo, conclui-se que uma nova abordagem de conformação arquitetural deve atender aos seguintes requisitos:

- (REQ1) Deve se basear em um processo bem definido para realização da verificação de conformação arquitetural. Em outras palavras, deve possuir uma metodologia com tarefas bem definidas para a sua utilização, de preferência mais simples do que a adotada pela ferramenta SAVE;

- (REQ2) Deve permitir a especificação de restrições arquiteturais de forma semelhante às regras de projeto utilizadas pela ferramenta LDM, contudo com um maior poder de expressão, possibilitando a utilização de expressões regulares (como ocorre no SAVE) e relacionamentos de subtipos (como ocorre no .QL);

- (REQ3) Deve contar com uma linguagem simples e de fácil compreensão. Assim, ao contrário das linguagens de restrições lógicas, a nova abordagem deve incluir uma linguagem de domínio especifíco simples e auto-explicativa, voltada principalmente a desenvolvedores e arquitetos acostumados com linguagens imperativas e orientadas a objetos;

- (REQ4) Não deve estender a linguagem de programação utilizada. Desse modo, desenvolvedores não terão que dominar uma nova linguagem de programação e lidar com a variedade e complexidade de novos elementos (como ocorre no ArchJava);

- (REQ5) Não deve necessitar de compiladores específicos, como ocorre com a maioria das ADLs. Em outras palavras, a nova abordagem deve ser completamente compatível com os compiladores normalmente utilizados pelos desenvolvedores e arquitetos;

- (REQ6) Deve possuir uma ferramenta para verificação automática de conformação arquitetural. Além disso, essa ferramenta deve ser compatível com linguagens atuais orientadas a objetos, ao contrário das ADLs;

- (REQ7) Deve prover conformação arquitetural por construção. Em outras palavras, modificações que violam a arquitetura planejada de um sistema devem ser detectadas logo que implementadas no código fonte. Desse modo, considera-se fundamental a

existência de uma ferramenta que possa ser facilmente aplicada sem, por exemplo, a necessidade de geração de um novo modelo (como ocorre no SAVE). Além disso, a ferramenta também deve ser continuamente aplicável, de preferência sempre após o processo de compilação do sistema.

A Tabela 2.1 apresenta um quadro comparativo das soluções descritas neste capítulo em relação ao atendimento dos requisitos mencionados.

		LDM	**SAVE**	**.QL**	**LCL**	**ADLs**
REQ1	Processo bem definido	√	√	X	√	√
REQ2	Poder de expressão	√*	√*	√	√	√
REQ3	Simplicidade	√	√	√*	X	X
REQ4	Não estende a linguagem alvo	√	√	√	√	X
REQ5	Não requer compiladores específicos	√	√	√	√	X
REQ6	Ferramenta compatível com linguagens atuais	√	√	√	√*	X
REQ7	Conformação arquitetural por construção	√*	X	√	X	√

Atendido: √ Parcialmente Atendido: √* Não Atendido: X

Tabela 2.1: Comparativo de DCL com outras técnicas

Como se pode observar, nenhuma das soluções atende a todos os requisitos propostos. Daí a motivação para o projeto de uma nova solução para verificação de conformação arquitetural.

Capítulo 3

A Linguagem DCL

Após a avaliação das ferramentas de verificação de conformação arquitetural realizada no Capítulo 2, este capítulo propõe uma nova solução para verificação arquitetural estática de sistemas que visa atender aos requisitos mencionados no final do Capítulo 2.

Este capítulo está organizado como descrito a seguir. A Seção 3.1 apresenta uma visão geral da solução proposta. A Seção 3.2 apresenta a linguagem de restrição de dependência proposta nesta dissertação, chamada DCL (*Dependency Constraint Language)* [TV08a, TV08b, TV10]. Essa linguagem constitui o componente central da solução para verificação de conformação arquitetural proposta. A Seção 3.3 apresenta alguns exemplos de uso prático da linguagem DCL, inclusive na verificação das restrições arquiteturais prescritas para o sistema `myAppointments`, descrito na Seção 2.1. A Seção 3.4 descreve o projeto e a implementação da ferramenta `dclcheck`, que verifica se o código fonte de um sistema respeita um conjunto de restrições de dependência definidas em DCL. E, por fim, a Seção 3.5 apresenta algumas considerações finais.

3.1 Visão Geral

A solução para verificação arquitetural proposta nesta dissertação utiliza técnicas de análise estática para detectar dependências inter-modulares impróprias, as quais considera-se que constituem uma fonte importante de violações arquiteturais. Conforme ilustrado na Figura 3.1, inicialmente o arquiteto de software deve definir as restrições de dependência do sistema, utilizando para isso a linguagem DCL, descrita na Seção 3.2. Para definição dessas restrições, o arquiteto deve se basear em informações e modelos do código fonte (tais como diagramas de classes) e em documentos e modelos arquiteturais (tais como diagramas de pacotes e componentes).

A solução proposta inclui também uma ferramenta de verificação de conformação ar-

quitetural, chamada `dclcheck` (*Dependency Constraint Language Checker*), que automaticamente detecta violações das restrições de dependência especificadas. Pelo menos dois cenários podem ser vislumbrados para aplicação da ferramenta `dclcheck`: como parte do processo de compilação ou durante os *builds* noturnos automatizados. Ao se integrar `dclcheck` ao processo de compilação, tem-se a vantagem de garantir que as restrições arquiteturais serão continuamente asseguradas à medida em que o sistema de software é implementado. Por exemplo, ele pode ser ativado nas fases iniciais da implementação ou para desenvolvedores que não possuem familiaridade com a arquitetura (como é o caso de novatos incorporados ao time de desenvolvedores). Por outro lado, verificação de conformação de forma contínua pode ser intrusiva, detectando violações que foram estabelecidas somente para depuração, testes ou outros propósitos temporários. Por essa razão, acredita-se que a maioria dos arquitetos irá preferir aplicar `dclcheck` durante os *builds* noturnos, como parte de suas atividades de controle e análise de qualidade.

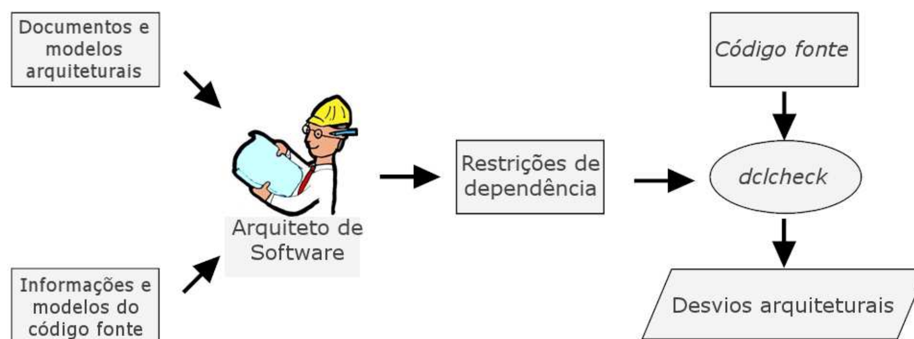


Figura 3.1: Solução proposta para conformação arquitetural

Como resultado da verificação de uma restrição, os seguintes tipos de violação podem ser detectados [MNS95, KP07]:

- Divergência: quando uma dependência existente no código fonte viola uma restrição de dependência especificada. Por exemplo, quando os desenvolvedores criam uma instância de uma classe `A` (utilizando o operador `new`) em um módulo que não foi especificado como sendo uma fábrica dessa classe `A`.

- Ausência: quando o código fonte não possui uma dependência que deveria existir de acordo com uma restrição de dependência especificada. Por exemplo, quando as classes do pacote `P` não implementam uma determinada interface `I`, como determinado pela arquitetura planejada do sistema.

Além disso, a solução proposta possui as seguintes características: (1) ela é baseada em técnicas de análise estática, o que é importante para evitar qualquer *overhead* durante

a execução dos sistemas; (2) ela é não-invasiva, uma vez que não depende do código fonte para efetuar a verificação das restrições de dependência (que é realizada diretamente sobre os *bytecodes*) nem realiza a instrumentação de código intermediário; (3) ela detecta violações de maneira incremental, isto é, os arquitetos podem começar aplicando DCL em uma pequena parte ou na parte mais crítica de seus sistemas; (4) ela provê conformação arquitetural por construção, isto é, violações na arquitetura planejada são detectadas logo que são implementadas no código fonte (similar ao que ocorre, por exemplo, com violações de modificadores de visibilidade, como `public`, `private`, etc); (5) ela é compatível com linguagens atuais de programação orientada a objetos (contudo, a ferramenta `dclcheck` está atualmente disponível somente para a linguagem Java).

3.2 Linguagem DCL

DCL (*Dependency Constraint Language*) é uma linguagem de domínio específico, declarativa e estaticamente verificável que permite a definição de restrições estruturais entre módulos. Assim, o objetivo principal da linguagem é restringir a organização modular de um sistema de software e não o seu comportamento. DCL utiliza um modelo de granularidade fina para especificação de dependências estruturais comuns em sistemas orientados a objetos. Esse modelo permite a definição de dependências originadas a partir do acesso a atributos e métodos, declaração de variáveis, criação de objetos, extensão de classes, implementação de interfaces, ativação de exceções e uso de anotações.

Como as atuais linguagens orientadas a objetos permitem a módulos clientes referenciar quaisquer tipos públicos de outros módulos, a lógica por trás do projeto da linguagem DCL consiste em prover meios para controlar tais dependências. Basicamente, para capturar divergências, arquitetos devem especificar que certas dependências podem (*only can* e *can-only*) ou não podem (*cannot*) ser estabelecidas por módulos específicos. Além disso, para capturar ausências, DCL permite aos arquitetos especificar que certas dependências devem estar presentes (*must*) em determinados módulos do sistema. Em resumo, o objetivo principal da linguagem é indicar violações estruturais que claramente representam anomalias arquiteturais e que, portanto, estão contribuindo para a erosão arquitetural do sistema.

A seguir, são descritos os principais elementos da linguagem DCL. A gramática completa da linguagem é apresentada no Apêndice A.

Módulos: Um módulo é basicamente um conjunto de classes. Suponha, por exemplo, as seguintes definições de módulos:

```
module View: org.foo.view.*
module DataStructure: org.foo.util.*, org.foo.view.Table, org.foo.view.Tree
module Model: org.foo.model.**
module Remote: java.rmi.UnicastRemoteObject+
module Frame: "org.foo.[a-zA-Z0-9/.]*Frame"
```

O módulo `View` inclui todas as classes do pacote `org.foo.view`. O módulo `DataStructure` inclui todas as classes do pacote `org.foo.util` e as classes `Table` e `Tree` do pacote `org.foo.view`. Portanto, o operador `*` designa todas as classes de um pacote[1]. Além disso, o módulo `Model` inclui todas as classes do pacote `org.foo.model` e de qualquer outro pacote interno a ele, tais como `org.foo.model.dao`, `org.foo.model.dao.impl`, `org.foo.model.dto`, etc. Isso se deve ao uso do operador `**`, que seleciona todas as classes de pacotes com o prefixo especificado. Por outro lado, o operador `+` seleciona subtipos de um dado tipo. Por exemplo, o módulo `Remote` denota todas as subclasses de `java.rmi.UnicastRemoteObject`. Ainda, a definição de módulos pode ser feita por meio de uma expressão regular delimitada por aspas. Por exemplo, o módulo `Frame` é composto por todas as classes cujo nome qualificado inicia-se com `org.foo.` e termina com `Frame`. Além disso, DCL provê dois módulos pré-declarados: `$java`, que inclui todos os tipos da biblioteca de Java, e `$system`, que se refere aos tipos específicos do sistema. Esses módulos pré-definidos são úteis quando necessita-se, por exemplo, restringir módulos a utilizar tipos do sistema (geralmente módulos reutilizáveis) ou para permitir que módulos utilizem toda a biblioteca de Java.

A linguagem empregada na especificação de módulos em DCL provê três benefícios importantes. Primeiro, ela permite um controle completo sobre a granularidade dos módulos. Por exemplo, ela unifica a sintaxe necessária para restringir dependências entre módulos contendo desde uma única classe até módulos contendo classes de pacotes distintos. Essa característica é fundamental para a escalabilidade de especificações DCL, uma vez que reduz o número de regras necessárias para restringir a arquitetura de sistemas complexos. Em segundo lugar, ela provê suporte a expressões regulares, o que permite que padrões de nomenclatura possam ser agrupados em um único módulo. Em terceiro lugar, ela permite a arquitetos de software mapear nomes do código fonte para componentes de alto nível, tais como `View`, `DataStructure` e `Model`. Portanto, DCL promove o desacoplamento das restrições de dependência de nomes de elementos do código fonte, o que aumenta o nível de abstração empregado na especificação de tais restrições.

[1] Veja que o operador `*` em DCL possui uma semântica diferente da sua utilização em comandos `import` de Java, em que somente as classes públicas de um pacote específico são selecionadas.

Divergências: Para capturar divergências, DCL suporta a definição das seguintes restrições entre módulos:

- *Somente as classes do módulo A podem depender dos tipos definidos no módulo B*, em que as possíveis dependências são as seguintes:
  - `only A can-access B`: somente as classes do módulo `A` podem acessar membros não-privados das classes declaradas no módulo `B`. Nesse caso, acessar significa ler e escrever em atributos ou invocar métodos.
  - `only A can-declare B`: somente as classes do módulo `A` podem declarar parâmetros formais ou variáveis dos tipos declarados no módulo `B`.
  - `only A can-handle B`: somente as classes do módulo `A` podem acessar membros não-privados e declarar parâmetros formais ou variáveis dos tipos declarados no módulo `B`. Em outras palavras, essa restrição é uma abreviação para `only A can-access, can-declare B`[2].
  - `only A can-create B`: somente as classes do módulo `A` podem criar objetos de classes declaradas no módulo `B`.
  - `only A can-extend B`: somente as classes do módulo `A` podem estender classes declaradas no módulo `B`.
  - `only A can-implement B`: somente as classes do módulo `A` podem implementar interfaces declaradas no módulo `B`.
  - `only A can-derive B`: somente as classes do módulo `A` podem estender classes ou implementar interfaces declaradas no módulo `B`, ou seja, somente as classes declaradas no módulo `A` podem ser subtipos de tipos declarados no módulo `B`. Assim, essa restrição é uma abreviação para `only A can-extend, can-implement B`.
  - `only A can-throw B`: somente os métodos das classes do módulo `A` podem retornar com exceções declaradas no módulo `B` ativadas.
  - `only A can-useannotation B`: somente as classes do módulo `A` podem utilizar anotações declaradas no módulo `B`.

[2] A princípio, poderia ser considerado que as restrições `access` e `declare` sempre serão utilizadas em conjunto e, portanto, poderiam ser substituídas pela restrição `handle`. Contudo, existem situações em que uma classe é acessada sem requerer a declaração de uma variável de seu tipo, como em acessos a atributos ou métodos estáticos. Por outro lado, existem situações em que uma variável é declarada, mas nenhum membro da classe é acessado, como nos parâmetros formais de métodos em classes que representam Adaptadores ou Fachadas. Tipicamente, esses parâmetros são utilizados somente para redirecionar a invocação para um outro método.

- `only A can-depend B`: somente as classes do módulo `A` podem depender dos tipos declarados no módulo `B`. Assim, essa restrição é uma abreviação para `only A can-access, can-declare, can-create, can-extend, can-implement, can-throw, can-useannotation B`, isto é, para qualquer tipo de dependência.

- *Classes do módulo A somente podem depender dos tipos definidos no módulo B*, em que as possíveis dependências são as seguintes:

  - `A can-only-access B`: as classes do módulo `A` somente podem acessar membros não-privados das classes declaradas no módulo `B`.
  - `A can-only-declare B`: as classes do módulo `A` somente podem declarar parâmetros formais ou variáveis dos tipos declarados no módulo `B`.
  - `A can-only-handle B`: as classes do módulo `A` somente podem acessar membros não-privados e declarar parâmetros formais ou variáveis dos tipos declarados no módulo `B`.
  - `A can-only-create B`: as classes do módulo `A` somente podem criar objetos de classes declaradas no módulo `B`.
  - `A can-only-extend B`: as classes do módulo `A` somente podem estender classes declaradas no módulo `B`.
  - `A can-only-implement B`: as classes do módulo `A` somente podem implementar interfaces declaradas no módulo `B`.
  - `A can-only-derive B`: as classes do módulo `A` somente podem estender classes ou implementar interfaces declaradas no módulo `B`.
  - `A can-only-throw B`: os métodos das classes do módulo `A` somente podem retornar com exceções declaradas no módulo `B` ativadas.
  - `A can-only-useannotation B`: as classes do módulo `A` somente podem utilizar anotações declaradas no módulo `B`.
  - `A can-only-depend B`: as classes do módulo `A` somente podem depender dos tipos declarados no módulo `B`.

- *Classes do módulo A não podem depender dos tipos definidos no módulo B*, em que as dependências que podem ser proibidas são as seguintes:

  - `A cannot-access B`: nenhuma classe do módulo `A` pode acessar membros não-privados das classes declaradas no módulo `B`.

– `A cannot-declare B`: nenhuma classe do módulo `A` pode declarar parâmetros formais ou variáveis dos tipos declarados no módulo `B`.

– `A cannot-handle B`: nenhuma classe do módulo `A` pode acessar membros não-privados ou declarar parâmetros formais ou variáveis dos tipos declarados no módulo `B`.

– `A cannot-create B`: nenhuma classe do módulo `A` pode criar objetos de classes declaradas no módulo `B`.

– `A cannot-extend B`: nenhuma classe do módulo `A` pode estender classes declaradas no módulo `B`.

– `A cannot-implement B`: nenhuma classe do módulo `A` pode implementar interfaces declaradas no módulo `B`.

– `A cannot-derive B`: nenhuma classe do módulo `A` pode estender classes ou implementar interfaces declaradas no módulo `B`.

– `A cannot-throw B`: nenhum método das classes do módulo `A` pode retornar com exceções declaradas no módulo `B` ativadas.

– `A cannot-useannotation B`: nenhuma classe do módulo `A` pode utilizar anotações declaradas no módulo `B`.

– `A cannot-depend B`: nenhuma classe do módulo `A` pode depender dos tipos declarados no módulo `B`.

Uma vez que o objetivo das restrições descritas anteriormente é capturar divergências, elas definem dependências que não podem ser estabelecidas no código fonte. Inclusive, esse também é o objetivo principal das restrições *only can* e *can-only*. Restrições *only can* automaticamente proíbem dependências originadas de classes não especificadas nos módulos de origem da restrição e restrições *can-only* automaticamente proíbem dependências para classes não especificadas nos módulos de destino da restrição. Isto é, restrições *only can* e *can-only* geram, 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`, em que `x = access`, `create`, `handle`, etc. Do mesmo modo, quando se define uma restrição da forma `A can-only-x B`, implica automaticamente na definição das seguintes restrições: `A cannot-x A` e `A cannot-x C`. Pela mesma razão, DCL não provê restrições *can*, uma vez que o padrão em linguagens orientadas a objetos já é permitir que qualquer módulo estabeleça dependências com tipos públicos de outros módulos.

Inicialmente, poderia se pensar que somente as restrições *only-can* e *cannot* são suficientes para restringir o espectro de dependências. Contudo, desse modo, não é possível restringir as dependências que um dado módulo pode estabelecer. Por exemplo, suponha o sistema previamente descrito, com três módulos `A`, `B` e `C`. Se o módulo `A`, além de possuir dependências internas para ele mesmo, somente pode depender do módulo `B`, bastaria criar uma restrição da forma `A cannot-depend C`. Porém, caso sejam inseridos mais módulos nesse sistema, essa restrição deve ser manualmente atualizada. Para resolver tal problema, diversas soluções foram avaliadas: (i) prover restrições *can* e permitir restrição contendo exceções (isto é, `A cannot-depend $system` seguida por `A can-depend B`), assim como ocorre na ferramenta LDM. Contudo, restrições com exceções a regras mais gerais dificultam o entendimento e ainda "poluem" o conjunto de restrições, uma vez que se torna mandatório definir duas restrições para a solução de um único problema; (ii) prover uma claúsula *except* na restrição `cannot` (isto é, `A cannot-depend $system except B`). Contudo, mesmo resolvendo o problema com apenas uma única restrição, são gerados dois novos problemas: modificação da sintaxe da restrição *cannot* e criação de uma restrição com duplo objetivo, uma vez que uma restrição *cannot* também poderia ser utilizada em um problema *can-only*; (iii) prover um operador de complemento de módulo (isto é, `A cannot-depend !B`). No entanto, mesmo não alterando a sintaxe da restrição *cannot*, essa restrição dificulta o entendimento da restrição já que combina duas negações: uma do *cannot* e outra do operador de complemento; (iv) prover uma nova restrição do tipo *can-only* (isto é, `A can-only-depend B`). Essa última alternativa foi considerada como a mais apropriada, pois soluciona o problema com apenas uma restrição, não utiliza nem modifica a sintaxe de um outro tipo de restrição, além de ser auto-explicativa.

Ausências: Para capturar ausências, DCL suporta a definição das seguintes restrições:

- *Classes do módulo A devem depender de tipos definidos no módulo B*, em que as dependências que podem ser requeridas são as seguintes:
  - `A must-extend B`: todas as classes do módulo `A` devem estender uma classe declarada no módulo `B`.
  - `A must-implement B`: todas as classes do módulo `A` devem implementar pelo menos uma interface declarada no módulo `B`.
  - `A must-derive B`: todas as classes do módulo `A` devem estender uma classe ou implementar pelo menos uma interface declarada no módulo `B`.
  - `A must-throw B`: todos os métodos das classes do módulo `A` devem declarar que podem retornar com exceções declaradas no módulo `B` ativadas.

– `A must-useannotation B`: todas as classes do módulo `A` devem utilizar pelo menos uma anotação declarada no módulo `B`.

No caso de restrições `must`, não é possível definir que classes devem acessar (*must-access*) serviços providos por outras classes, que devem declarar (*must-declare*) variáveis de um tipo específico, que devem criar (*must-create*) objetos de classes específicas, etc. A razão é que o objetivo principal da linguagem DCL não é prover especificações sobre a estrutura interna ou comportamental de classes. De fato, tal especificação detalhada somente é realizada durante o projeto ou durante a própria implementação. Por outro lado, decidiu-se prover suporte a restrições `must` no nível de classes e assinaturas de métodos, pois tais restrições podem ser utilizadas para obrigar certos módulos a utilizar serviços providos por *frameworks* ou sistemas externos, o que é uma decisão importante do ponto de vista arquitetural.

A Figura 3.2 resume a sintaxe da linguagem proposta para declaração de restrições de dependência. A gramática da linguagem é apresentada no Apêndice A.

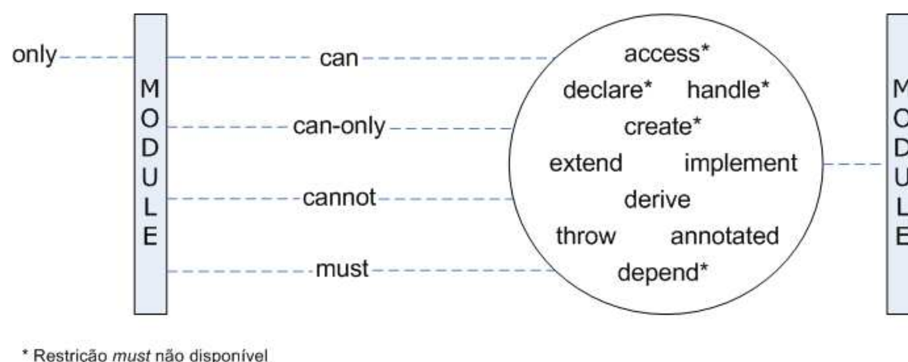


Figura 3.2: Sintaxe para declaração de restrições de dependência na linguagem DCL

Inconsistências: Como descrito, DCL suporta os seguintes tipos de dependência: `access`, `declare`, `create`, `extend`, `implement`, `throw` e `useannotation`. No exemplo a seguir, são definidas duas restrições de dependências inconsistentes, uma vez que é impossível estabelecer uma dependência que respeite ambas as restrições:

```
only A1 can-create B
only A2 can-create B
```

Supondo que $A1 \cap A2 = \emptyset$, é impossível respeitar ambas as restrições quando da criação de um objeto de uma classe declarada no módulo `B`. Em sua implementação atual, `dclcheck` não identifica inconsistências em sua especificação. Ao invés disso, a

ferramenta somente indica que uma das restrições é violada quando uma dependência autorizada pela outra restrição é estabelecida. Em tais casos, os desenvolvedores necessitam manualmente refatorar as restrições para remover a inconsistência. Por exemplo, uma possível refatoração para permitir aos módulos `A1` e `A2` criar classes declaradas no módulo `B` seria:

```
only A1, A2 can-create B
```

Está previsto o desenvolvimento de um módulo de edição de restrições de dependência em DCL, a ser integrado à ferramenta `dclcheck`. Esse módulo de edição agregará diversos recursos como autocompletar, destaque de sintaxe e identificação de inconsistências.

3.3 Exemplos

Esta seção tem como objetivo apresentar exemplos práticos de cada um dos tipos de dependência (`access`, `declare`, `create`, etc) da linguagem DCL. Além disso, apresenta-se um conjunto de restrições de dependência que preservam as restrições arquiteturais prescritas para o sistema `myAppointments` (descritas na Seção 2.1).

3.3.1 Tipos de Dependência

Conforme descrito na Seção 3.2, cada quantificador de restrição (*only can*, *can only*, *cannot* e *must*) é seguido por um tipo de dependência. Esse tipo de dependência pode ser mais abrangente (`depend`) ou mais específico (`access`, `declare`, `create`, `extend`, `implement`, `throw` e `useannotation`). Esta subseção tem como objetivo apresentar utilizações práticas de cada um desses tipos de dependência, a fim de justificar a sua inserção na linguagem.

Access: Suponha, por exemplo, um sistema em que somente um módulo pode acessar os atributos e métodos de uma dada classe estática.

```
1: module Formulas:  org.foo.formulas.*
2: only Formulas can-access java.lang.Math
```

Nesse exemplo, o módulo `Formulas` agrupa as classes do pacote `org.foo.formulas` (linha 1). Em seguida, na linha 2, foi criada uma restrição que somente permite ao módulo `Formulas` acessar a classe padrão de Java que oferece operações matemáticas

(`java.lang.Math`).

Declare: Suponha, por exemplo, o padrão de projeto Fachada [GHJV94]. As classes de fachada fornecem uma interface única e simplificada para os recursos e facilidades mais comuns de um sistema e geralmente apenas encaminham a invocação a outros objetos.

```
1: module DomainFacades:  org.foo.facades.*
2: module Domain:         org.foo.model.Domain+
3: DomainFacades can-only-declare Domain, $java
4: DomainFacades cannot-access Domain
```

Nesse exemplo, o módulo `DomainFacades` agrupa todas as classes de fachada do sistema (linha 1) e o módulo `Domain` agrupa as classes de domínio do sistema, subclasses de `org.foo.model.Domain` (linha 2). Em seguida, na linha 3, foi criada uma restrição que define que o módulo `DomainFacades` somente pode declarar objetos de domínio e tipos da biblioteca de Java. Na linha 4, uma outra restrição proíbe `DomainFacades` de acessar objetos de domínio.

Create: Suponha, por exemplo, o padrão *Abstract Factory* [GHJV94]. Nesse padrão de projeto, somente uma determinada classe, que funciona como a fábrica de uma família de produtos, pode realizar a criação desses objetos.

```
1: module Factory:  org.foo.Factory
2: module Product:  org.foo.Product+
3: only Factory can-create Product
```

Nesse exemplo, o módulo `Factory` possui apenas a classe `org.foo.Factory` (linha 1) e o módulo `Product` agrupa a classe `org.foo.Product` e as suas subclasses (linha 2). Em seguida, na linha 3, foi criada uma restrição que define que somente o módulo `Factory` pode criar produtos, ou seja, instanciar classes do módulo `Product`.

Extend: Suponha, por exemplo, um sistema Java *Web*. Com o intuito de organizar o sistema e facilitar a sua manutenibilidade, todos os *servlets* devem estar localizados em um pacote específico.

```
1: module Servlets:         org.foo.controller.servlets.*
2: module BaseHttpServlet:  javax.servlet.http.HttpServlet
3: Servlets must-extend BaseHttpServlet
4: only Servlets can-extend BaseHttpServlet
```

Nesse exemplo, o módulo `Servlets` agrupa as classes do pacote `org.foo.controller.servlets` (linha 1) e o módulo `BaseHttpServlet` inclui somente a classe `javax.servlet.http.HttpServlet`, que é a classe padrão de Java para manipular requisições e respostas usando o protocolo HTTP (linha 2). Em seguida, na linha 3, foi criada uma restrição que exige que as classes do módulo `Servlets` estendam a classe `javax.servlet.http.HttpServlet` e, na linha 4, proíbe-se que classes de outros módulos estendam essa classe.

Implement: Suponha ainda o sistema Java *Web* previamente utilizado como exemplo. Nesse sistema foram também definidas algumas convenções de nomenclatura.

```
1: module Filters:  "org.foo.[a-zA-Z0-9/.]*Filter"
2: Filters must-implement java.util.EventListener
```

Nesse exemplo, o módulo `Filters` agrupa as classes que possuem sufixo `Filter` e pertencem ao pacote `org.foo` (linha 1). Em seguida, na linha 2, foi criada uma restrição que define que as classes do módulo `Filters` devem implementar a interface `java.util.EventListener`.

Throw: Suponha, por exemplo, um sistema em que classes DAO utilizam JDBC para implementar o serviço de persistência. Nesse sistema, além de somente classes DAO poderem utilizar JDBC, os métodos dessas classes não podem retornar com exceções dessa biblioteca ativadas.

```
1: module DAO:  "org.foo.model.dao.[a-zA-Z0-9/.]*DAO"
2: DAO cannot-throw java.sql.SQLException+
```

Nesse exemplo, o módulo `DAO` agrupa classes que possuem sufixo `DAO` e que pertencem ao pacote `org.foo.model.dao` (linha 1). Em seguida, na linha 2, foi criada uma restrição que proíbe que métodos de classes DAO retornem com exceções da biblioteca JDBC ativadas (isto é, exceções do tipo `java.sql.SQLException` e de suas subclasses).

Useannotation: Suponha, por exemplo, um sistema em que práticas pouco recomendadas de programação como atributos não utilizados, métodos privados nunca invocados, utilização de classes descontinuadas, estão sendo encobertas através de uma anotação específica da linguagem Java.

```
1: $system cannot-useannotation java.lang.SuppressWarnings
```

Nesse exemplo, não foi necessária a definição de um módulo, uma vez que DCL provê o módulo pré-definido `$system`, que se refere aos tipos específicos do sistema. Logo, essa restrição proíbe qualquer tipo do sistema de utilizar a anotação `java.lang.-SuppressWarnings`, que é uma anotação nativa da linguagem Java responsável por suprimir *warnings*. Desse modo, obtêm-se a garantia de que tais práticas não recomendadas de programação serão devidamente reportadas.

3.3.2 Sistema `myAppointments`

A seção anterior apresentou alguns exemplos de uso de DCL em sistemas hipotéticos. Já nesta seção, utiliza-se DCL na verificação completa da arquitetura de um sistema. Para isso, foi utilizado o sistema `myAppointments`, descrito na Seção 2.1. Basicamente, esse sistema segue o padrão arquitetural MVC e possui seis restrições arquiteturais que devem ser seguidas durante toda a sua evolução. Desse modo, foi criado um conjunto de restrições de dependência para garantir que o sistema não sofra erosão arquitetural.

A Figura 3.3 apresenta o conjunto das restrições de dependência definidas para o sistema `myAppointments`[3]. Inicialmente, uma seqüência de definições de módulos é utilizada para agrupar classes relacionadas (linhas 1-9). Os módulos `Controller` e `View` agrupam subclasses de `myapp.controller.IController` e `myapp.view.View`, respectivamente (linhas 2 e 3). O módulo `Model` agrupa as classes do pacote `myapp.model` e de qualquer outro pacote interno a ele (linha 4). Os módulos `Domain` e `Util` agrupam as classes dos pacotes `myapp.model.domain` e `myapp.util`, respectivamente (linhas 5 e 6). O módulo `DAO` é definido por uma expressão regular que agrupa todas as classes que estão abaixo do pacote `myapp.model` e que possuem sufixo `DAO` (linha 7). Por fim, os módulos `JavaAwtSwing` e `JavaSql` agrupam, respectivamente, componentes gráficos e de persistência de Java (linhas 8 e 9).

Pode-se observar que os módulos definidos se assemelham àqueles existentes na visão arquitetural do sistema apresentada na Figura 2.2 (Seção 2.1). Isso evidencia que DCL permite expressar dependências entre entidades que são normalmente utilizadas por arquitetos de software para descrever seus sistemas. Além disso, convém salientar que pode-se definir módulos a partir de classes de bibliotecas externas, como os módulos `JavaAwtSwing` e `JavaSql` que se referem a módulos da biblioteca de Java.

Nas linhas 10-16, uma sequência de restrições de dependência é definida. Essencialmente, tais restrições são fundamentais para garantir que as restrições arquiteturais prescritas para o sistema `myAppointments` sejam preservadas durante a implementação

[3]Por questões de legibilidade, o nome do pacote `myappointments` foi abreviado para `myapp`.

```
 1: %Módulos
 2: module Controller:    myapp.controller.IController+
 3: module View:          myapp.view.View+
 4: module Model:         myapp.model.**
 5: module Domain:        myapp.model.domain.*
 6: module Util:          myapp.util.*
 7: module DAO:           "myapp.model.[a-zA-Z0-9/.]*DAO"
 8: module JavaAwtSwing:  java.awt.**, javax.swing.**
 9: module JavaSql:       java.sql.**

10: %Restrições
11: only View can-depend JavaAwtSwing
12: only DAO can-depend JavaSql
13: View cannot-depend Model
14: Domain can-only-depend $java
15: DAO can-only-depend Domain, Util, JavaSql
16: Util cannot-depend $system
```

Figura 3.3: Restrições de dependência do sistema `myAppointments`

e evolução do sistema. A restrição descrita na linha 11 define que somente a camada de visão pode utilizar serviços das bibliotecas AWT e Swing, conforme requerido pela RA1 (definida na Seção 2.1). A restrição da linha 12 define que somente classes DAO podem utilizar serviços SQL, conforme requerido pela RA2. A restrição da linha 13 impede que a camada de visão dependa da camada de modelo, conforme especificado pela RA3. A restrição da linha 14 define que objetos de domínio somente podem depender da API de Java, o que implica que eles não podem depender de classes específicas do sistema motivador, conforme requerido pela RA4. A restrição da linha 15 garante que as classes DAO somente dependem de objetos de domínio, de classes do pacote `util` e dos serviços SQL. Por fim, a restrição da linha 16 garante que as classes do módulo `util` não dependem de classes específicas do sistema motivador, conforme especificado pela RA6.

Convém salientar que foi necessária uma única restrição de dependência para especificar cada uma das seis restrições arquiteturais definidas para o sistema `myAppointments`. Isso sugere que a linguagem DCL possui uma sintaxe muito próxima daquela utilizada por arquitetos de software para definir restrições arquiteturais.

3.4 `dclcheck`

Com o intuito de avaliar a aplicabilidade da solução proposta, foi implementado um protótipo de uma ferramenta, chamada `dclcheck`, que verifica se o código fonte (isto é, a arquitetura concreta de um sistema) respeita restrições de dependência definidas

em DCL. Essa ferramenta foi implementada como um *plug-in* para a IDE Eclipse. Além disso, ela utiliza um *framework* de análise e manipulação de *bytecode*, chamado ASM[4], para extrair, a partir dos *bytecodes*, todas as dependências existentes entre os módulos de um sistema (descritas na Seção 3.2). A implementação atual da ferramenta `dclcheck` possui 28 tipos (classes, interfaces e enumerações), sete pacotes e 2.262 LOC. Basicamente, a implementação da ferramenta possui quatro módulos principais:

- *Dependency Map Builder*: este módulo é responsável por extrair todas as dependências existentes entre os módulos de um sistema. Além do mais, ele é o único módulo que utiliza o *framework* ASM. O mapa de dependências construído é uma estrutura de dados que associa cada classe `A` de um sistema a uma lista de classes dependentes `B1`, `B2`, ..., `Bn`. Além disso, cada entrada do mapa de dependências inclui o tipo da dependência (por exemplo, `access`, `declare`, `create`, etc).

- *Parser*: este módulo efetua a leitura do conjunto de restrições de dependência em DCL e constrói, para cada restrição, uma estrutura de dados que contém seu quantificador, seu tipo, seus módulos de origem e de destino. Por exemplo, na restrição `only A can-create B`, o quantificador da restrição é `only-can`, o tipo é `create`, o módulo de origem é `A` e o módulo de destino é `B`.

- *Validator*: este módulo é responsável por verificar se o sistema respeita cada restrição de dependência analisada. A forma de detecção de violações se diferencia para cada quantificador de restrição. Para detectar violações da forma `only A can-x B`, este módulo procura no mapa de dependências por classes não localizadas no módulo `A` que estabelecem uma dependência do tipo `x` com o módulo `B` (onde `x = access`, `declare`, `create`, etc). Por outro lado, para detectar violações em restrições da forma `A can-only-x B`, este módulo procura por classes localizadas no módulo `A` que estabelecem uma dependência do tipo `x` com classes não localizadas no módulo `B`. Para detectar violações em restrições da forma `A cannot-x B`, este módulo procura por classes localizadas no módulo `A` que estabelecem uma dependência do tipo `x` com o módulo `B`. Por fim, para detectar violações em restrições da forma `A must-x B`, este módulo procura por classes localizadas no módulo `A` que não estabelecem uma dependência do tipo `x` com o módulo `B`.

- *Output*: este módulo apresenta as violações detectadas pelo módulo anterior em uma visão da IDE Eclipse, conforme pode ser observado na Figura 3.5. A visão chamada "*Architectural Drifts View*" apresenta o resultado da verificação de conformação

[4] http://asm.objectweb.org.

arquitetural em forma de uma tabela. Nessa tabela, cada linha corresponde a um desvio arquitetural que apresenta diversas informações, tais como o nome do projeto, a restrição de dependência violada, o nome da classe violada e sua localização no código fonte. Além disso, uma descrição detalhada da violação e um deslocamento para o ponto da violação no código fonte podem ser solicitados.

A Figura 3.4 apresenta um diagrama arquitetural da ferramenta `dclcheck`, destacando as entradas e saídas de cada módulo e suas dependências.

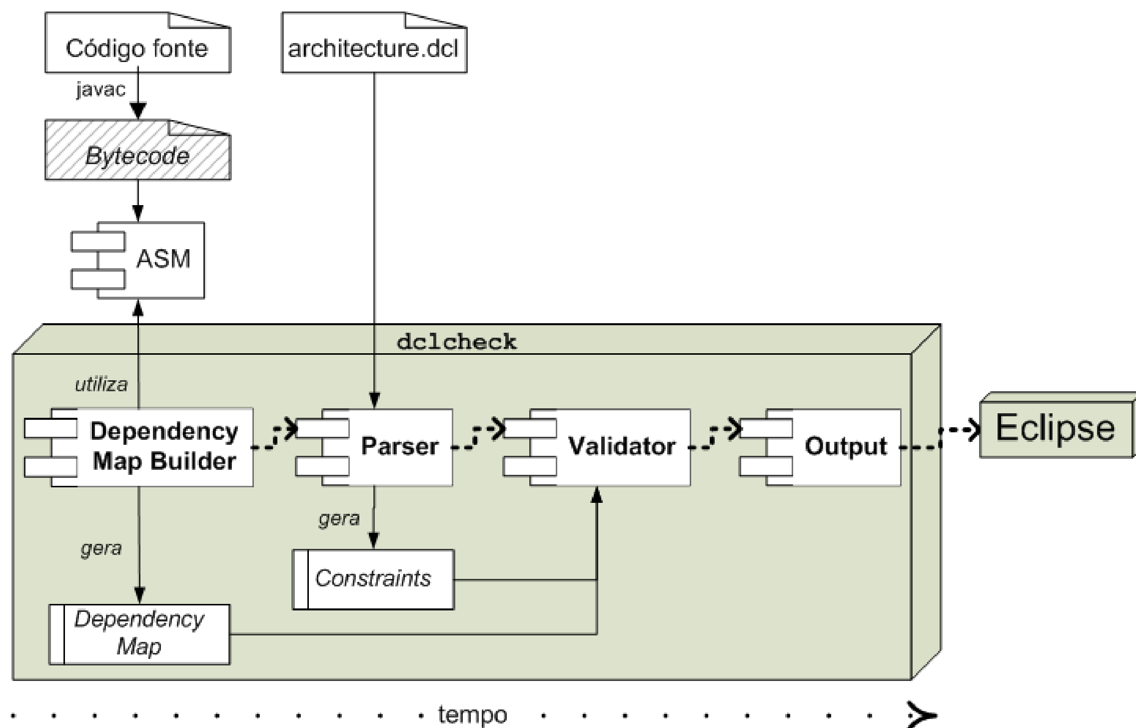


Figura 3.4: Funcionamento da ferramenta `dclcheck`

A aplicação da ferramenta `dclcheck` no sistema `myAppointments` é ilustrada na Figura 3.5. Primeiramente, a verificação da arquitetura inicia-se a partir da invocação da ação de *pop-up* chamada "*Check Architecture using* `dclcheck`". Assim que invocada, o processamento é encaminhado ao *Dependency Map Builder* que constrói, a partir dos arquivos já compilados, o mapa de dependências. Logo após esse processo, o *Parser* efetua a leitura do arquivo `architecture.dcl` (arquivo onde os módulos e as restrições encontram-se definidos). Então, o *Validator* verifica cada uma das restrições e, por fim, o *Output* apresenta as violações detectadas, caso existam.

Para ilustrar a detecção de violações arquiteturais, foram inseridos propositalmente alguns desvios arquiteturais na implementação do sistema `myAppointments`. Primeiramente, foi inserida, em uma classe do módulo `util`, uma dependência com a API Swing,

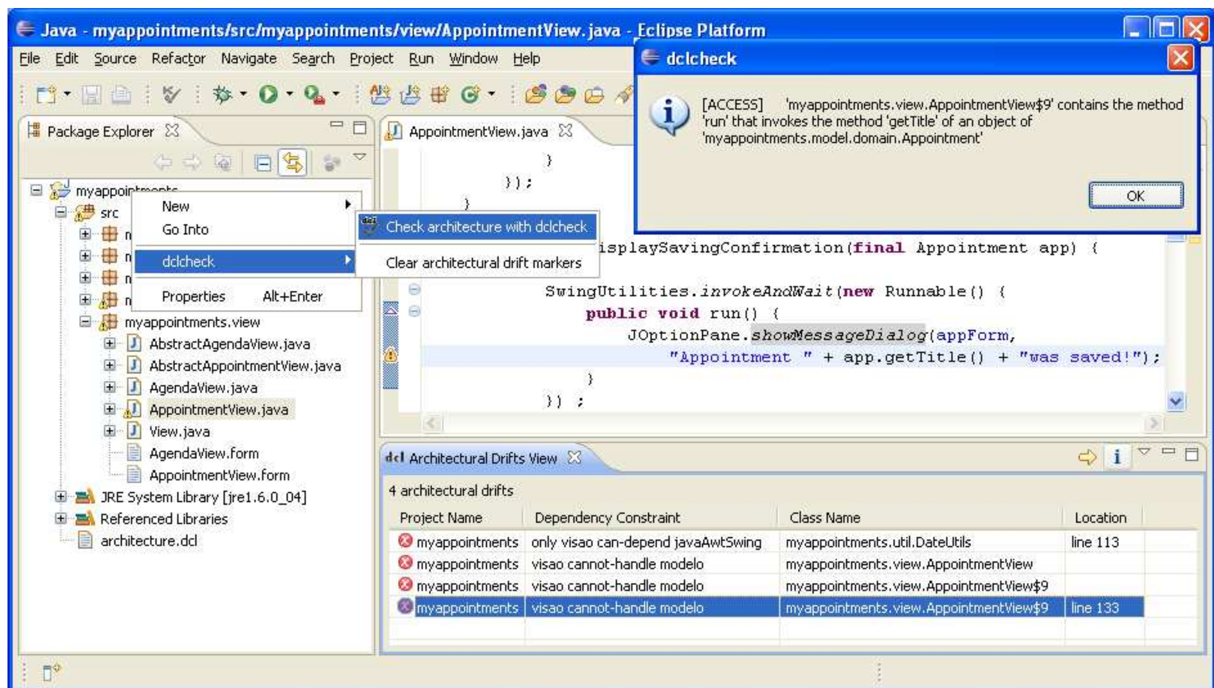


Figura 3.5: Tela do `dclcheck` com violações detectadas no sistema `myAppointments`

o que contradiz a RA1, uma vez que somente classes da camada de visão podem estabelecer tal dependência. Depois, foi inserido, em uma classe da camada de visão, um trecho de código que acessa diretamente a camada de modelo, o que contradiz a RA3. Conforme pode ser observado na Figura 3.5, a ferramenta `dclcheck` foi capaz de detectar esses desvios arquiteturais .

3.5 Considerações Finais

Neste capítulo, foi apresentada uma nova abordagem para verificação arquitetural estática de sistemas. A linguagem DCL foi descrita e exemplificada, inclusive na verificação completa da arquitetura de um sistema. Além disso, descreveu-se a implementação de um protótipo de uma ferramenta capaz de detectar violações de restrições de dependência definidas em DCL.

A solução descrita neste capítulo atende aos requisitos propostos na Seção 2.6:

- Para atender ao REQ1, a solução proposta se baseia em um processo simples, porém bem definido, para verificação de conformação arquitetural. Enquanto ferramentas como o SAVE requerem a criação de um modelo arquitetural e a extração de um modelo de código fonte para detectar violações arquiteturais, a solução proposta

requer apenas a definição de restrições de dependência utilizando a linguagem DCL.

- Para atender ao REQ2, a solução proposta inclui a linguagem DCL. Enquanto a ferramenta LDM utiliza regras de projeto, DCL utiliza restrições de dependência que atuam de maneira similar, contudo com um maior poder de expressão. Por exemplo, em DCL existe um maior número de quantificadores de restrição (`only-can`, `can-only`, `cannot` e `must`) e um maior nível de especialização nos tipos de dependências (`access`, `declare`, `create`, etc). Além disso, a linguagem DCL permite o agrupamento de classes utilizando expressões regulares e relacionamento de subtipos;

- Para atender ao REQ3, DCL foi projetada como uma linguagem de domínio específico, com uma sintaxe clara e auto-explicativa. A simplicidade e objetividade da linguagem foi demonstrada, por exemplo, pela necessidade de uma única restrição de dependência para expressar cada uma das seis restrições arquiteturais prescritas para o sistema `myAppointments`;

- Para atender ao REQ4, a solução proposta não envolve a extensão de uma linguagem de programação de uso geral, como Java. Com isso, desenvolvedores não precisam dominar uma nova linguagem de programação. Essa característica também permite que a ferramenta possa ser aplicada em qualquer etapa da implementação de um sistema, uma vez que necessita apenas da definição das restrições de dependência e do código compilado do sistema alvo;

- Para atender ao REQ5, a solução proposta não necessita de compiladores específicos. Em vez disso, necessita apenas do código fonte compilado em um compilador compatível com a linguagem Java;

- Para atender ao REQ6, a solução proposta inclui uma ferramenta chamada `dclcheck` que verifica se o código de um sistema respeita um conjunto de restrições de dependência definidas em DCL. Essa ferramenta foi implementada como um *plug-in* para a IDE Eclipse, inicialmente voltado para sistemas Java;

- Para atender ao REQ7, a solução proposta permite verificação arquitetural por construção, sem a necessidade de geração manual de novos modelos ou artefatos.

Assim, conclui-se que a abordagem para verificação de conformação arquitetural descrita neste capítulo atende aos requisitos propostos na Seção 2.6. Isso é um indicador que a solução proposta contempla as melhores características das ferramentas avaliadas no Capítulo 2.

Capítulo 4

Estudo de Caso

Este capítulo descreve a aplicação da linguagem DCL e da ferramenta `dclcheck` em um sistema real de grande porte, chamado SGP, que é utilizado pelo Serviço Federal de Processamento de Dados (SERPRO) para gestão de seus empregados.

Este capítulo está organizado como descrito a seguir. A Seção 4.1 apresenta o sistema SGP. A Seção 4.2 descreve a metodologia aplicada no estudo de caso. A Seção 4.3 define restrições de dependência para cada uma das versões analisadas do sistema SGP. A Seção 4.4 apresenta os resultados da aplicação da ferramenta `dclcheck` em cada versão. A Seção 4.5 analisa os resultados obtidos. E, por fim, a Seção 4.6 apresenta algumas considerações finais.

4.1 Sistema de Gestão de Pessoas (SGP)

O Sistema de Gestão de Pessoas (SGP) é um sistema de gerenciamento de recursos humanos utilizado pelo SERPRO para gestão de seus empregados. O sistema automatiza atividades como folha de pagamento, benefícios (plano de cargo, plano de saúde, pensão, aposentadoria, etc) e atendimento às leis trabalhistas (férias, licenças, etc) de aproximadamente 12 mil empregados.

O projeto do sistema SGP teve início em dezembro de 2005 e sofreu uma importante reestruturação arquitetural em maio de 2006. Desse modo, seus arquitetos de software consideram que a primeira versão estável do sistema foi disponibilizada em junho de 2006. Logo, essa foi a primeira versão analisada neste estudo de caso.

A arquitetura do sistema SGP segue o padrão arquitetural MVC (*Model-View-Controller*) [Fow02], conforme ilustrado na Figura 4.1. 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 en-

capsulam regras de negócio e comportamentos. DTOs representam entidades de domínio, tais como empregados, planos de cargo, departamentos, etc. DAOs proveem uma interface para acesso ao *framework* de persistência subjacente. Particularmente, na implementação do sistema SGP utiliza-se o *framework* Hibernate[1] para persistência objeto/relacional.

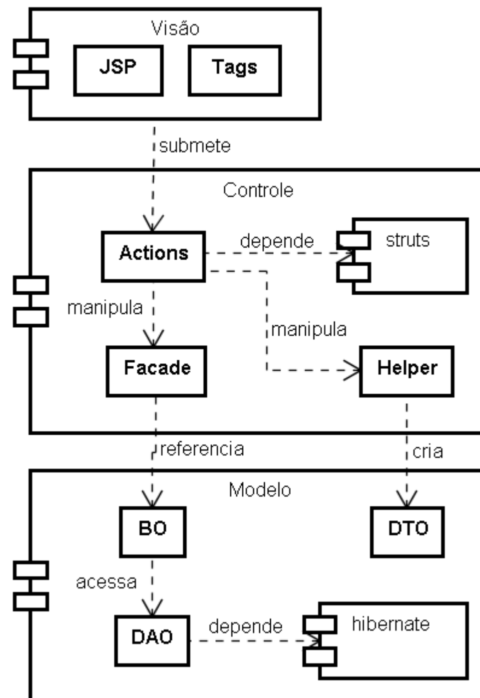


Figura 4.1: Arquitetura do Sistema SGP

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[2] é utilizado pela camada de Controle para manipular requisições HTTP. Tais requisições são adaptadas por um componente *Helper* [ACM03]. 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 *JavaServer Pages* (JSP)[3] 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 (Hibernate, Struts, JSP, etc) que atualmente são largamente utilizados no desenvolvimento de sistemas Java *Web*.

[1]http://www.hibernate.org.
[2]http://struts.apache.org.
[3]http://java.sun.com/products/jsp.

4.2 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 na Tabela 4.1. A primeira versão analisada foi disponibilizada em junho de 2006, correspondendo à primeira versão estável do sistema. A terceira versão analisada refere-se à versão do sistema de abril de 2008, que corresponde a versão mais recente do sistema no momento da realização deste estudo de caso. 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 julho de 2007. É importante observar que, além de um aumento significativo de funcionalidades entre as versões, houve a inserção de novas tecnologias de uma versão para outra. Como resultado, o sistema aumentou de 18 KLOC (primeira versão) para quase 240 KLOC (terceira versão).

	1ª versão	**2ª versão**	**3ª versão**
Data	Junho, 2006	Julho, 2007	Abril, 2008
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, Displaytag, Log4J e JSTL	Anteriores mais JAX-WS, Quartz, DWR e Hibernate Annotations	Anteriores mais JasperReports

Tabela 4.1: Versões analisadas no estudo de caso

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

1. Com apoio dos arquitetos de software responsáveis pelo sistema SGP, foram definidas, na linguagem DCL, restrições de dependência para cada uma das versões analisadas, isto é, foram definidos módulos e restrições *only can*, *can only*, *cannot* e *must*. Essa atividade é descrita com mais detalhes na Seção 4.3;

2. A ferramenta `dclcheck` foi aplicada sobre cada uma das versões para detectar divergências e ausências a partir das restrições definidas no passo anterior;

3. As relações divergentes e ausentes foram reportadas aos arquitetos de software com o intuito de confirmar se elas realmente representam violações arquiteturais. Essa

atividade é descrita com mais detalhes na Seção 4.4.

4.3 Restrições de Dependência

Inicialmente, foram explicados os principais elementos e objetivos da linguagem DCL aos arquitetos de software responsáveis pelo sistema SGP (o que não levou mais de uma hora). Em seguida, esses arquitetos descreveram os principais estilos arquiteturais utilizados pelo sistema SGP, os principais módulos e as principais relações entre eles (o que levou mais uma hora). Com base nessas informações e com apoio e orientação dos arquitetos, foram definidas restrições de dependência críticas para preservar a arquitetura de cada uma das três versões selecionadas do sistema SGP (o que levou aproximadamente oito, três e duas horas para a primeira, segunda e terceira versões, respectivamente). Convém ainda mencionar que as restrições foram definidas de acordo com as informações fornecidas pelos arquitetos e que a qualidade do resultado da verificação de conformação arquitetural depende diretamente da precisão de tais informações. Informações sobre o número e o tipo das restrições definidas são apresentadas na Tabela 4.2.

	1ª versão	**2ª versão**	**3ª versão**
Número de módulos	22	39	44
Número de restrições *only can*	15	20	24
Número de restrições *can only*	5	5	5
Número de restrições *cannot*	3	1	1
Número de restrições *must*	15	22	25
Total de restrições	**38**	**48**	**55**

Tabela 4.2: Informações sobre as restrições de dependência definidas para o sistema SGP

Observa-se na Tabela 4.2 que as restrições *only can* são mais comuns que as restrições dos tipos *can only* e *cannot*. A primeira situação se justifica pelo fato de ser mais frequente restringir quais módulos podem estabelecer dependências com outros módulos (conceito *only can*) do que restringir quais dependências podem ser estabelecidas por um módulo (conceito *can only*). Já a segunda situação 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`, automaticamente são definidas as 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*.

Alterações	Motivo	Número de Restrições	
		1ª ver. ⇒ 2ª ver.	2ª ver. ⇒ 3ª ver.
Inclusão de restrições	Novos *frameworks*	+7	+2
	Novas funcionalidades	+3	+5
	Refatorações	+2	0
Atualização de restrições	Novos *frameworks*	3	0
	Novas funcionalidades	0	2
	Refatorações	2	0
Remoção de restrições	Refatorações	-2	0
Total		+10	+7

Tabela 4.3: Alterações nas restrições definidas (quando comparadas as três versões analisadas no sistema SGP)

A Tabela 4.3 descreve as principais alterações nas restrições definidas à medida que o sistema SGP se desenvolveu da primeira para a segunda versão considerada no estudo de caso e também da segunda para a terceira versão. Como pode ser observado nessa tabela, de uma versão para outra, novas restrições foram incluídas, restrições existentes foram atualizadas e restrições anteriormente definidas foram removidas. Basicamente, tais alterações foram motivadas por três fatores. Primeiramente, como mencionada na Tabela 4.1, novos *frameworks* foram integrados ao sistema SGP. Por exemplo, a partir da segunda versão, o *framework* Quartz[4] foi empregado para prover serviços de agendamento de tarefas. Isso demandou novas restrições para especificar os módulos autorizados a utilizar esse serviço. Além disso, uma vez que tarefas agendadas devem utilizar a fachada do sistema para acessar informações da base de dados, houve a necessidade de atualizar restrições existentes. Em segundo lugar, de uma versão para outra, várias novas funcionalidades foram implementadas, o que resultou em novas restrições e atualizações nas restrições existentes. Finalmente, em terceiro lugar, refatorações aplicadas ao longo das versões analisadas também resultaram na inclusão, atualização e remoção de restrições. Como exemplos de tais refatorações, pode-se mencionar divisão de pacotes e suporte a novas estratégias de instanciação de classes de fachada.

A Figura 4.2 apresenta um subconjunto das restrições de dependência comuns às três versões analisadas do sistema SGP[5]. Inicialmente, uma sequência de definições de módulos é utilizada para agrupar classes relacionadas (linhas 1-28). Essas definições agrupam desde classes do sistema (linhas 1-20) até classes de bibliotecas externas (linhas

[4]http://www.opensymphony.com/quartz.

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

21-28). Por exemplo, o módulo `AllAction` agrupa todas as ações do sistema[6] (linha 3), o módulo `HibernateDAO` agrupa as implementações de objetos de acesso a dados (linha 11), etc. Por outro lado, o módulo `Struts` refere-se a qualquer tipo do *framework* Struts (linha 28), o módulo `Hibernate` a qualquer tipo do *framework* Hibernate (linha 23), etc.

Ainda convém mencionar que foram utilizados todos os tipos de definição de módulos da linguagem (classe específica, expressão regular, operadores `*` e `**`, etc), com exceção do agrupamento por subtipos (operador `+`). Isso ocorreu pelo fato de os arquitetos de software optarem por agrupar certos módulos por pacotes (operadores `*` e `**`) ao invés de utilizar agrupamento por subtipos. Por exemplo, o módulo `AllAction` definido por `sgp.controller.action.**` (linha 3) poderia ser equivalentemente definido por `org.-apache.struts.action.Action+`. Por fim, pode-se observar que os módulos definidos se assemelham àqueles existentes na visão arquitetural do sistema apresentada na Figura 4.1. Isto evidencia mais uma vez que DCL permite expressar dependências entre entidades que são normalmente utilizadas por arquitetos de software para descrever seus sistemas.

Nas linhas 29-42, uma sequência de restrições *only can* é definida. Essencialmente, tais restrições são fundamentais para garantir que a arquitetura MVC é preservada durante a evolução do sistema. Por exemplo, uma das restrições define que somente classes da camada de Controle podem depender de tipos do *framework* Struts (linha 30). Uma outra restrição define quais módulos podem manipular tipos do módulo `Facade` (linha 31). Isto impede que a camada de Visão ultrapasse a camada de Controle e acesse diretamente o Modelo. Além do mais, uma outra restrição especifica que o módulo `Facade` é o único módulo na camada de Controle que pode manipular tipos associados a objetos de negócio (linha 34). Assim, basicamente, as restrições das linhas 29-42 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, somente as entidades de domínio e as implementações de DAOs da camada de Modelo podem depender do *framework* de persistência Hibernate (linha 32).

Utilizando a linguagem de restrição de dependência (DCL) proposta é possível também tornar explícita a diferença entre fábricas e clientes de um certo tipo. Por exemplo, existe uma restrição que define que BOs somente podem ser criados pela classe `BOFactory` (linha 39). Além do mais, uma outra restrição indica que somente BOs podem manipular interfaces de DAO (linha 35), porém não podem manipular diretamente implementações de DAOs (linha 50). É possível também restringir métodos que podem retornar com

[6] No *framework* Struts, ações (*actions*) é o nome dado para classes que manipulam requisições e respostas.

```
 1: %Módulos do sistema SGP
 2: module Ajax:                  sgp.controller.ajax.*
 3: module AllAction:             sgp.controller.action.**
 4: module BaseDTO:               sgp.model.dto.ParentPersistent
 5: module BO:                    sgp.model.bo.*
 6: module Controller:            sgp.controller.**
 7: module ControllerExcp:        sgp.controller.exception.*
 8: module Facade:                sgp.facade.*
 9: module Form:                  sgp.controller.form.**
10: module Helper:                sgp.controller.helper.*
11: module HibernateDAO:          "sgp.model.dao.hibernate[a-zA-Z]*HibernateDAO"
12: module IDAO:                  "sgp.model.dao.I[a-zA-Z]*DAO"
13: module ModelExcp:             sgp.model.exception.*
14: module DTO:                   sgp.model.dto.**
15: module QuartzJob:             sgp.quartz.job.*
16: module QuartzPkgs:            sgp.quartz.**
17: module QuartzScheduler:       sgp.controller.action.QuartzAction,
                                  sgp.controller.action.TaskSchedulerAction
18: module Tags:                  sgp.taglib.**
19: module Util:                  sgp.util.**
20: module WS:                    sgp.webservice.**

21: %Módulos externos
22: module ApacheCommonsUtil:     org.apache.commons.**
23: module Hibernate:             org.hibernate.**
24: module HibernateAnnotations:  org.hibernate.annotations.*
25: module JPA:                   javax.persistence.**
26: module JWS:                   javax.jws.**
27: module Quartz:                org.quartz.**
28: module Struts:                org.apache.struts.**

29: %Restrições only can
30: only Controller can-depend Struts
31: only Ajax, Controller, QuartzPkgs, Tags, WS can-handle Facade
32: only DTO, HibernateDAO can-depend Hibernate
33: only QuartzScheduler, QuartzPkgs can-depend Quartz
34: only Facade, BO can-handle BO
35: only BO can-handle IDAO
36: only WS can-depend JWS
37: only DTO can-useannotation JPA, HibernateAnnotations
38: only sgp.model.dao.hibernate.HibernateDAOFactory can-create HibernateDAO
39: only sgp.model.bo.BOFactory can-create BO
40: only sgp.controller.service.HelperLocator can-create Helper
41: only AllAction, Helper, Facade can-throw ControllerExcp
42: only BO, IDAO, HibernateDAO can-throw ModelExcp

43: %Restrições can only
44: Helper can-only-declare Helper, IFacade, DTO, Form, $java
45: IDAO, HibernateDAO, BO can-only-throw ModelExcp
46: Util can-only-depend Util, $java, ApacheCommonsUtil
47: DTO can-only-depend DTO, $java, Hibernate, Util, ApacheCommonsUtil
48: Form can-only-depend Form, $java, Struts, Util, ApacheCommonsUtil

49: %Restrições cannot
50: BO cannot-handle HibernateDAO

51: %Restrições must
52: Tags must-implement javax.servlet.jsp.tagext.JspTag
53: IDAO must-implement sgp.model.dao.DataAccessObject
54: HibernateDAO must-implement IDAO
55: HibernateDAO must-extend sgp.model.dao.hibernate.HibernateImplDAO
56: QuartzJob must-extend sgp.quartz.job.SGPJob
57: AllAction must-extend sgp.controller.action.BaseAction
58: DTO must-derive BaseDTO, java.io.Serializable
59: DTO must-useannotation JPA, HibernateAnnotations
60: Facade must-useannotation sgp.annotations.Facade
```

Figura 4.2: Restrições de dependência do sistema SGP

exceções de determinado tipo ativadas. Por exemplo, somente métodos das classes dos módulos `AllAction`, `Helper` e `Facade` podem retornar com exceções definidas no módulo `ControllerExcp` ativadas (linha 41).

Nas linhas 43-48, define-se uma sequência de restrições *can only*. Basicamente, essas restrições são utilizadas para restringir o espectro de dependências de um dado módulo e, usualmente, complementam restrições *only can*. Por exemplo, objetos de negócio, interfaces e implementações de DAOs somente podem retornar com exceções definidas no módulo `ModelExcp` ativadas (linha 45), complementando a restrição que somente permite a esses módulos retornar com essas exceções ativadas (linha 42). Por outro lado, esse tipo de restrição é também utilizado com o intuito de prover módulos reutilizáveis. Por exemplo, as classes utilitárias do sistema somente podem depender entre si, de classes utilitárias providas pelo Apache Commons[7] e de tipos da biblioteca Java (linha 46).

Nas linhas 51-60, define-se uma sequência de restrições *must*. Inicialmente, tais restrições são utilizadas para garantir que todas as classes que compõem um certo módulo estendam ou implementem um certo tipo. Geralmente, esse tipo é definido em um outro módulo do sistema ou é fornecido por algum *framework* externo. Como um exemplo do primeiro caso, todas as classes do módulo `AllAction` devem estender uma classe interna chamada `BaseAction` (linha 57), o que assegura a existência de somente ações no padrão do sistema nesse módulo. Como exemplo do segundo caso, todas as classes no módulo `Tags` devem implementar a interface `javax.servlet.jsp.tagext.JspTag` (linha 52). Por fim, restrições *must* também foram utilizadas para garantir que o padrão DAO, responsável por prover uma forma abstrata de acesso à base de dados, seja corretamente utilizado (linhas 53-55) e para garantir que todos os DTOs utilizem anotações JPA[8] e do Hibernate (linha 59). Em resumo, tais restrições são importantes para garantir que o sistema SGP utilize corretamente os serviços providos por outras classes ou *frameworks*. Do mesmo modo, elas contribuem para orientar os desenvolvedores a utilizar *frameworks* corretamente, conforme prescrito pela arquitetura do sistema.

4.4 Resultados

A ferramenta `dclcheck` foi utilizada para detectar violações das restrições propostas para cada uma das três versões analisadas neste estudo de caso A Figura 4.4 apresenta uma tela do *plug-in* com algumas das violações arquiteturais detectadas na primeira versão

[7] Apache Commons é um projeto Apache cujo propósito é fornecer bibliotecas reutilizáveis de código aberto. http://commons.apache.org.

[8] Java Persistence API. http://java.sun.com/developer/technicalArticles/J2EE/jpa.

analisada do sistema SGP. As violações são listadas em uma visão da IDE Eclipse, a qual fornece uma descrição detalhada da violação. Ainda, é possível solicitar o deslocamento para o ponto da violação no código fonte.

A Tabela 4.4 apresenta as violações detectadas pela ferramenta `dclcheck` sumarizadas por número de classes com violações (ausências ou divergências). Em outras palavras, classes com várias violações da mesma restrição foram contabilizadas uma única vez. Essa decisão foi tomada devido a uma mesma violação de uma restrição de dependência poder ocorrer em vários pontos do código fonte. Por exemplo, a Figura 4.3 apresenta um trecho de código de um BO que estabelece dependência indevida com o *framework* Hibernate (restrição 32). Contudo, foram detectadas violações nessa restrição em vários pontos desse trecho, como na declaração da variável `sess` (linha 2), na declaração da variável `criteria` (linha 3), no acesso ao método `createCriteria` (linha 3), etc.

```
1: try {
2:    Session sess = hibHolder.getSession();
3:    Criteria criteria = sess.createCriteria(UORG.class);
4:    criteria.add(Restrictions.eq("codigo",obj));
5:    criteria.add(Restrictions.isNull("dataFim"));
6:    criteria.setProjection(Projections.property("descricao"));
7: } catch(HibernateException e){
8:    throw new DataAccessException(e);
9: }
```

Figura 4.3: Trecho de um BO contendo a mesma violação em diversos pontos

É importante mencionar que algumas restrições não foram avaliadas para certas versões e, por isso, foram marcadas com um `NP` (não presente). Isso ocorreu pelo fato de a restrição ainda não estar presente naquela versão, o que só veio a ocorrer em versões posteriores pela incorporação de novas tecnologias ou por uma evolução arquitetural. Um exemplo seriam restrições relacionadas à utilização do *framework* JAX-WS (restrição 36) e de anotações do Hibernate (restrição 37 e 59). Esses dois *frameworks* somente foram incorporados a partir da segunda versão analisada.

Como pode ser observado nessa tabela, ocorreu um número considerável de violações das restrições de dependência propostas. Por exemplo, a arquitetura do sistema SGP define um serviço de criação de DAOs (restrição 38). Contudo, foram encontradas seis classes na segunda e terceira versões que criam diretamente DAOs sem a utilização desse serviço de fábrica. Como um outro exemplo, existem duas restrições definindo que BOs somente podem manipular interfaces de DAOs (restrição 35) e nunca as respectivas implementações (restrição 50). Essas restrições são importantes para desacoplar a camada de Modelo do serviço de persistência subjacente, o que permite tornar as regras de negócio

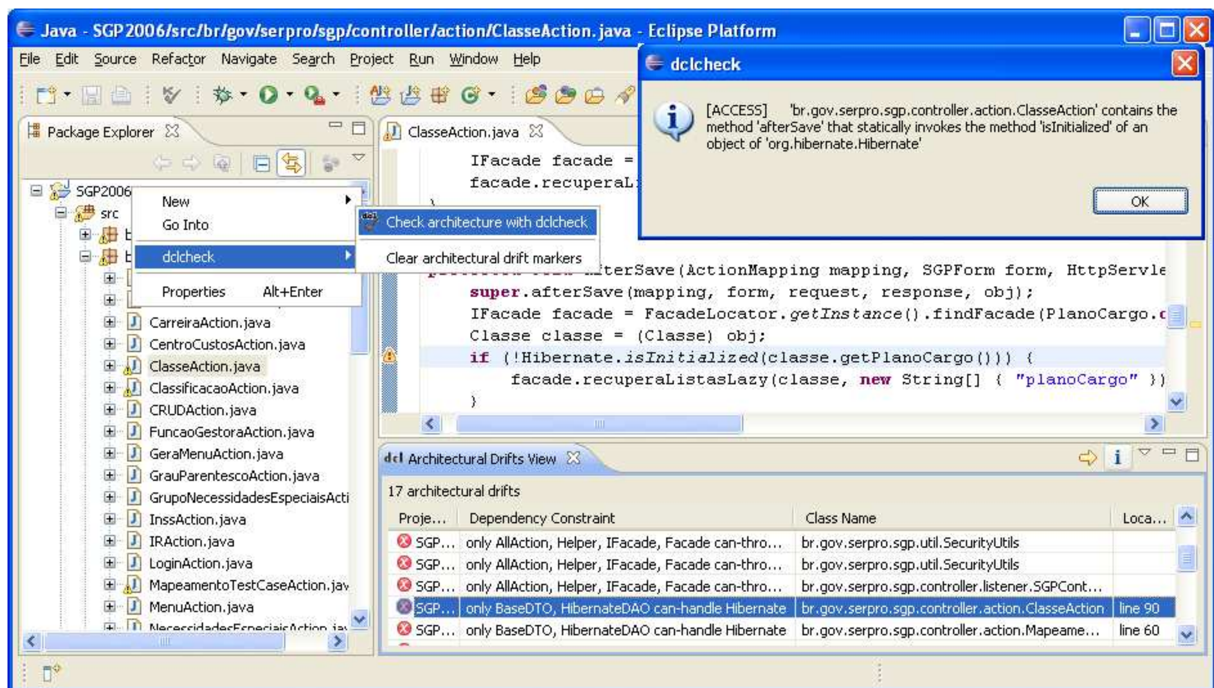


Figura 4.4: Tela do `dclcheck` com violações detectadas no sistema SGP

independentes do serviço de persistência utilizado. Porém, foram encontrados nove BOs na segunda versão e dez BOs na terceira versão que acessam diretamente implementações de DAOs.

Além do mais, restrições que definem que somente os métodos das classes das camadas de Controle e Modelo podem retornar com exceções específicas ativadas (restrições 41 e 42) foram violadas em várias partes do sistema. Basicamente, isso demonstra a existência de métodos de uma camada retornando exceções de uma outra camada. As restrições *can only* também apresentaram violações. Por exemplo, as classes utilitárias deveriam ser implementadas de forma que fossem reutilizáveis. Contudo, foram encontradas quatro classes na primeira versão, 16 classes na segunda versão e 21 classes na terceira versão que estabelecem dependências com outros módulos diferente daqueles especificados (restrição 46). Essas violações demonstram que um grande número de classes utilitárias não poderiam ser reutilizadas em outros projetos.

Por outro lado, restrições *must* demonstraram um alto grau de convergência, apresentando violações somente nas restrições 54, 55, 57 e 59. Além disso, essas restrições foram violadas em, no máximo, duas classes. Por exemplo, existem apenas duas classes, que representam objetos de domínio, na segunda e terceira versões que não estão utilizando anotações JPA ou do Hibernate (restrição 59).

A Tabela 4.4 também demostra que o número de classes com violações aumentou consideravelmente ao longo das versões analisadas: onze classes com violações foram detectadas na primeira versão (isto é, 3,5% das classes do sistema), 175 classes na segunda versão (9,1%) e 245 classes na terceira versão (10,5%). Após dois anos da data de entrega da primeira versão estável, esses resultados indicam que a arquitetura do sistema SGP encontra-se em um processo relevante de degeneração.

Análise dos Resultados: As violações reportadas na Tabela 4.4 foram apresentadas aos arquitetos do sistema SGP que confirmaram que elas realmente representam violações arquiteturais, sem exceção. Com a orientação do arquiteto responsável, as violações detectadas foram classificadas nas seguintes categorias: violações das camadas MVC (tal como acesso à camada de Modelo diretamente da camada de Visão), uso inapropriado de padrões de persistência (na maioria das vezes BO e DAO), uso não autorizado de *frameworks* (isto é, módulos acessando *frameworks* aos quais lhes foi negado o acesso), não uso de *frameworks* (isto é, módulos não estão utilizando *frameworks* que eles deveriam utilizar), comprometimento de reúso (geralmente acoplamentos com tipos específicos do sistema) e uso inapropriado de padrões de projeto (principalmente fábricas). Além disso, os arquitetos avaliaram a relevância das categorias propostas como baixa, média ou alta de acordo com o seu potencial em erodir a arquitetura do sistema SGP.

As categorias propostas são descritas na Tabela 4.5, incluindo o número de violações detectadas para cada categoria na terceira versão analisada do sistema (isto é, o número de pontos do código fonte em que as restrições classificadas na categoria foram violadas). A Tabela 4.5 também apresenta uma estimativa preliminar sobre o número de homens-horas necessários para corrigir cada tipo de violação.

Como descrito na Tabela 4.5, violações das camadas MVC e uso inapropriado de padrões de persistência foram classificados pelos arquitetos do sistema SGP como os tipos de violações arquiteturais mais importantes. Particularmente, violações das camadas MVC ocorreram em 283 pontos do código fonte. Para corrigir tais violações, desenvolvedores devem mover os campos e métodos relevantes da classe violada para uma nova classe na camada correta (isto é, aplicar a refatoração extrair classe [FBB+99]). Eles também devem revisar o código de tratamento de exceção, para evitar a propagação de exceções de modo não autorizado. Estimou-se que 60 homens-horas são necessários para corrigir esses tipos de violação.

Os arquitetos do sistema SGP classificaram o uso não autorizado de *frameworks*, não uso de *frameworks* e comprometimento de reúso como tendo média relevância. Para corrigir violações classificadas como uso não autorizado de *frameworks* – o tipo mais comum

Restrições de Dependência	**Classes violadas**		
	1ª ver.	**2ª ver.**	**3ª ver.**
Restrições *only can*			
30: **only** Controller **can-depend** Struts	-	1	3
31: **only** Ajax, Controller, QuartzPkgs, Tags, WS **can-handle** Facade	1	44	55
32: **only** DTO, HibernateDAO **can-depend** Hibernate	3	38	47
33: **only** QuartzScheduler, QuartzPkgs **can-depend** Quartz	NP	1	3
34: **only** Facade, BO **can-handle** BO	-	-	-
35: **only** BO **can-handle** IDAO	-	-	-
36: **only** WS **can-depend** JWS	NP	-	-
37: **only** DTO **can-useannotation** JPA, HibernateAnnotations	NP	-	-
38: **only** sgp.model.dao.hibernate.HibernateDAOFactory **can-create** HibernateDAO	-	6	6
39: **only** sgp.model.bo.BOFactory **can-create** BO	-	18	26
40: **only** sgp.controller.service.HelperLocator **can-create** Helper	-	2	2
41: **only** AllAction, Helper, Facade **can-throw** ControllerExcp	2	20	31
42: **only** BO, IDAO, HibernateDAO **can-throw** ModelExcp	-	3	4
Restrições *can only*			
44: Helper **can-only-declare** Helper, IFacade, DTO, Form, $java	-	1	1
45: IDAO, HibernateDAO, BO **can-only-throw** ModelExcp	-	3	10
46: Util **can-only-depend** Util, $java, ApacheCommonsUtil	4	16	21
47: DTO **can-only-depend** DTO, $java, Hibernate, Util, ApacheCommonsUtil	-	-	3
48: Form **can-only-depend** Form, $java, Struts, Util, ApacheCommonsUtil	-	8	18
Restrições *cannot*			
50: BO **cannot-handle** HibernateDAO	-	9	10
Restrições *must*			
52: Tags **must-implement** javax.servlet.jsp.tagext.JspTag	-	-	-
53: IDAO **must-implement** sgp.model.dao.DataAccessObject	-	-	-
54: HibernateDAO **must-implement** IDAO	-	1	1
55: HibernateDAO **must-extend** sgp.model.dao.hibernate.HibernateImplDAO	-	1	1
56: QuartzJob **must-extend** sgp.quartz.job.SGPJob	NP	-	-
57: AllAction **must-extend** sgp.controller.action.BaseAction	1	1	1
58: DTO **must-derive** BaseDTO, java.io.Serializable	-	-	-
59: DTO **must-useannotation** JPA, HibernateAnnotations	NP	2	2
60: Facade **must-useannotation** sgp.annotations.Facade	NP	-	-
Total	**11**	**175**	**245**

NP: Não Presente

Tabela 4.4: Violações arquiteturais detectadas no sistema SGP

de violação encontrado no estudo de caso – desenvolvedores devem utilizar refatorações como extrair classe e mover método [FBB+99], o que estimou-se que deve demandar aproximadamente 40 homens-horas. Finalmente, os arquitetos do sistema SGP classificaram

Tipo de Violação	Restrições	Relevância	Localizações	Acerto
Violações das camadas MVC	31, 34, 41, 42, 45	Alta	283	60h
Uso inapropriado de padrões de persistência	35, 50, 53-55	Alta	33	2h
Uso não autorizado de *frameworks*	30, 32, 33, 36, 37, 59	Média	1867	40h
Não uso de *frameworks*	52, 56-58, 60	Média	1	0,5h
Comprometimento de reúso	46-48	Média	577	50
Uso inapropriado de padrões de projeto	38-40, 44	Baixa	62	2h
Total	-	-	**2823**	**154,5**

Tabela 4.5: Tipos de Violação (somente com base na terceira versão do sistema SGP)

o uso inapropriado de padrões de projeto como tendo baixa relevância. Nesse caso, a correção consiste na utilização apropriada de métodos de fábricas (o que deve demandar aproximadamente dois homens-horas).

Desempenho: A Tabela 4.6 descreve o tempo que foi necessário para verificar a arquitetura do sistema SGP utilizando a ferramenta `dclcheck`. Com o intuito de ajudar na avaliação do desempenho da ferramenta, esta tabela também apresenta informações sobre o tempo de compilação completa do sistema na plataforma Eclipse. Os tempos apresentados foram medidos em uma máquina Intel com processador Core 2 Duo 1.66GHz, 3GB de RAM, sistema operacional Microsoft Windows XP versão 2002 com *Service Pack* 2. Além disso, foram utilizados a IDE Eclipse versão 3.3.2 e a JVM versão 1.6.0_10-rc.

Como apresentado na Tabela 4.6, o tempo despendido pela ferramenta `dclckeck` para verificar as restrições de dependência definidas variou entre 18% e 36% do tempo de compilação do sistema. Por exemplo, na terceira versão analisada do sistema SGP, que possui quase 240 KLOC, `dclcheck` executou em tempo equivalente a 35% do tempo de compilação. Esses resultados são estimuladores, pois demonstram que a ferramenta `dclcheck` pode ser frequentemente aplicada por desenvolvedores e arquitetos para prover verificação de conformação arquitetural.

4.5 Análise Crítica

Com base principalmente na experiência adquirida com o estudo de caso do sistema SGP, esta seção inclui uma análise crítica da solução proposta para verificação de con-

	1ª versão	2ª versão	3ª versão
Compilação completa (A)	2,21	10,24	13,89
`dclcheck` (B)	0,39	3,67	4,75
B/A	0,18	0,36	0,35

Tabela 4.6: Desempenho da ferramenta `dclcheck` (em segundos)

formação arquitetural. A análise é baseada nos seguintes critérios:

- Poder de expressão: A solução proposta é centrada na definição de dependências estruturais que não podem ou que devem ser estabelecidas entre módulos de sistemas orientados a objetos. Como observado no estudo de caso do sistema SGP, várias violações arquiteturais foram introduzidas pelo estabelecimento de dependência inter-modulares impróprias. Por exemplo, na terceira versão analisada do sistema SGP, foram detectadas 245 classes (isto é, 10,5% das classes do sistema) com violações relacionadas a dependências inter-modulares impróprias. Tais violações envolveram todos os tipos de dependência suportados pela linguagem DCL (**`access`**, **`declare`**, **`create`**, **`extend`**, **`useannotation`**, etc). Entretanto, não se pode afirmar que a solução proposta é completa (isto é, que ela detecta todas as possíveis formas de violações arquiteturais). Mais especificamente, assumindo a correção das restrições de dependência definidas, a ferramenta `dclcheck` não reporta falsos positivos (isto é, ela não reporta violações arquiteturais equivocadamente). Por outro lado, ela pode levar a falsos negativos, no sentido que ela não detecta violações que não podem ser expressas em DCL. Por exemplo, uma vez que a linguagem DCL é baseada em técnicas de análise estática, ela não trata informações dinâmicas, tais como: execuções de um método `X` devem chamar um método `Y`, objetos da classe `A` devem referenciar objetos do tipo `B`, etc. Além do mais, não é possível regular dependências estabelecidas por meio de reflexão computacional. Por exemplo, se o código fonte incluir 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. Entretanto, tais limitações não se constituíram em grandes obstáculos ao se verificar a arquitetura do sistema SGP.

- Nível de abstração: A solução proposta permite aos arquitetos definir restrições utilizando entidades de alto nível de seus próprios sistemas. Basicamente, módulos são utilizados para definir tais entidades de uma forma flexível. Por exemplo, no estudo de caso do sistema SGP, dois tipos de módulos foram utilizados. Primeiro, existem

módulos que denotam classes relacionadas a padrões arquiteturais ou de projeto, tais como `Controller`, `Facade`, `DAO`, etc. Em segundo lugar, existem módulos que abrangem sistemas externos ou *frameworks*, tais como `Struts` e `Hibernate`. Em ambos os casos, os módulos propostos refletem o vocabulário utilizado pelos arquitetos de software do sistema SGP para descrever a arquitetura do sistema. Além disso, a linguagem permite aos arquitetos especificar parcialmente restrições arquiteturais, isto é, antes da finalização do projeto detalhado e das fases de implementação. Essa especificação é dita parcial, pois definições de módulos mapeiam entidades arquiteturais de alto nível para seus elementos de código fonte correspondentes. Por essa razão, requere-se que ao menos o projeto do sistema tenha sido concluído. Por outro lado, a experiência adquirida no estudo de caso do sistema SGP sugere que a seção de restrições de programas em DCL pode ser reutilizada em outros sistemas que sigam a mesma arquitetura.

- Aplicabilidade: A solução proposta tem pelo menos três propriedades chaves que contribuem para sua aplicação no mundo real. Primeiramente, ela é baseada em uma linguagem de domínio específico pequena e relativamente de fácil aprendizado. Essa característica distingue DCL de outras técnicas e ferramentas com o mesmo propósito, mas baseadas, por exemplo, em linguagens lógicas [HH06, HHR04, EKKM08, MKPW06]. Em segundo lugar, como a solução proposta é baseada em técnicas de análise estática, a ferramenta `dclcheck` não requer qualquer forma de anotação em código fonte ou instrumentação de *bytecode*, além de não requerer a execução do sistema. Desse modo, ela não tem qualquer impacto em tempo de execução e também não requer acesso à base de dados do sistema. Como se trata de um sistema de grande porte cujos dados são 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 de produção. Acredita-se que essa seja uma preocupação comum a gerentes de grandes sistemas de software. Em terceiro lugar, a ferramenta `dclcheck` é adequada para uso diário. Por exemplo, resultados de desempenho prelimilares demostraram que o processo de verificação de conformação arquitetural leva cerca de um terço do tempo de compilação do sistema.

4.6 Considerações Finais

Neste capítulo, foi descrita a aplicação da linguagem DCL e da ferramenta `dclcheck` em um sistema de grande porte, chamado SGP. A partir de uma metodologia bem definida, foram levantadas restrições de dependência em DCL no sentido de garantir que a implementação desse sistema segue a arquitetura planejada. Como resultado, constatou-se que a arquitetura do sistema SGP vem sofrendo um processo crescente de erosão arquitetural. Por exemplo, foram detectadas 11 classes na primeira versão, 175 classes na segunda versão e 245 classes na terceira versão desse sistema que estabelecem alguma forma de dependência que representa uma violação perante a arquitetura planejada do sistema. Além disso, a experiência adquirida neste estudo de caso permitiu avaliar o poder de expressão, o nível de abstração e a aplicabilidade da solução proposta.

Capítulo 5

Conclusão

Este capítulo apresenta as considerações finais da dissertação, ressaltando os pontos principais da solução proposta, as suas contribuições e linhas de trabalhos futuros. Além disso, compara-se a solução proposta com outras abordagens.

O restante deste capítulo está organizado como descrito a seguir. A Seção 5.1 descreve os pontos mais significativos da solução proposta, como seus objetivos, o propósito da linguagem DCL, etc. A Seção 5.2 compara a solução proposta com trabalhos relacionados. A Seção 5.3 apresenta as principais contribuições da solução. E, por fim, a Seção 5.4 apresenta possíveis trabalhos futuros.

5.1 Visão Geral da Solução Proposta

A solução para verificação de conformação arquitetural proposta nesta dissertação se baseia na idéia de que dependências inter-modulares impróprias constituem uma fonte importante de violações arquiteturais e, portanto, contribuem para o processo de erosão arquitetural.

Diante disso, a solução proposta inclui a linguagem DCL, uma linguagem de domínio específico, declarativa e estaticamente verificável. DCL permite a definição de restrições estruturais entre módulos no sentido de restringir o espectro de dependências que são permitidas em sistemas orientados a objetos. A linguagem foi elaborada com base em requisitos considerados indispensáveis a uma solução para verificação de conformação arquitetural, os quais foram levantados a partir da avaliação e comparação de diversas ferramentas e técnicas já existentes.

Em resumo, DCL permite a definição de módulos e restrições. Os módulos podem variar desde uma única classe até todas as classes de uma biblioteca externa, o que permite que módulos sejam definidos de forma que se assemelhem àqueles existentes na visão arqui-

tetural do sistema. Isso se deve a variedade de formas possíveis para definição de módulos, como hierarquia de pacotes, expressões regulares, relacionamento de subtipos, etc. Por outro lado, as restrições são formadas por um quantificador (`only can`, `can-only`, `cannot` ou `must`) acrescido de um tipo de dependência, que pode ser mais abrangente (`depend`) ou mais específico (`access`, `declare`, `create`, `extend`, `implement`, `throw` e `useannotation`). Enquanto os quantificadores `only can`, `can-only` e `cannot` detectam divergências e suportam qualquer tipo de dependência, o quantificador `must` detecta ausências em nível de classes e assinaturas de métodos.

Além disso, a solução proposta inclui uma ferramenta chamada `dclcheck` que verifica se o código fonte de um sistema respeita restrições de dependência definidas em DCL. A ferramenta `dclcheck` utiliza técnicas de análise estática não-invasivas e possui desempenho suficiente para uso diário.

Para demonstrar a aplicabilidade da solução proposta, foi conduzido um estudo de caso com três versões de um sistema de recursos humanos de grande porte chamado SGP. Na terceira versão avaliada desse sistema, foram detectadas 245 classes – ou 10,5% das classes do sistema – com alguma forma de dependência estrutural que representa uma violação perante a arquitetura planejada do sistema. Baseado nos resultados desse estudo de caso, foi realizada uma análise crítica das seguintes características da solução proposta:

- Poder de expressão, em que a solução foi capaz de restringir dependências críticas para preservar a arquitetura do sistema SGP. Além disso, desde que assumida a correção das restrições de dependência definidas, ela não reporta falsos positivos. Por outro lado, a solução proposta não é completa, uma vez que pode levar a falsos negativos (pois não detecta violações relacionadas a informações dinâmicas ou geradas por meio de reflexão);

- Nível de abstração, em que a definição de módulos baseou-se no mesmo vocabulário utilizado pelos arquitetos do sistema SGP para descrever a arquitetura do sistema. Também é possível aplicar a solução em uma pequena parte ou na parte mais crítica de um sistema, sem a necessidade de abranger toda a arquitetura. Além disso, destacou-se a possibilidade de reutilização das restrições propostas em outros sistemas baseados na mesma arquitetura;

- Aplicabilidade, em que foi observada a facilidade de uso da linguagem DCL em relação a linguagens de restrições lógicas. Ainda, destacou-se a não necessidade de anotações em código fonte, instrumentação de *bytecode* e execução do sistema. Por fim, o desempenho da ferramenta `dclcheck` também foi analisado. Por exemplo,

resultados prelimilares relativos ao sistema SGP demostraram que o processo de verificação de conformação arquitetural leva cerca de um terço do tempo de compilação do sistema.

5.2 Comparação com Trabalhos Relacionados

No Capítulo 2, foram comparadas e avaliadas três ferramentas que proveem suporte a técnicas representativas para prevenir erosão arquitetural. Como resultado desse estudo, foram levantados sete requisitos considerados relevantes em soluções para verificação de conformação arquitetural. Esses requisitos são atendidos pela solução proposta, conforme pode ser observado na Tabela 5.1.

		DCL	**LDM**	**SAVE**	**.QL**
REQ1	Processo bem definido	√	√	√	X
REQ2	Poder de expressão	√	√*	√*	√
REQ3	Simplicidade	√	√	√	√*
REQ4	Não estende a linguagem alvo	√	√	√	√
REQ5	Não requer compiladores específicos	√	√	√	√
REQ6	Ferramenta compatível com linguagens atuais	√	√	√	√
REQ7	Conformação arquitetural por construção	√	√*	X	√

Atendido: √ Parcialmente Atendido: √* Não Atendido: X

Tabela 5.1: Comparativo de DCL com outras técnicas

A intenção desses requisitos foi guiar a criação de uma solução que contemple as melhores características das técnicas existentes. Assim, as principais melhorias que DCL oferece em relação às ferramentas avaliadas são descritas a seguir:

- LDM: Em relação ao REQ2, as regras de projeto de LDM possuem somente duas formas (`A can-use B` ou `A cannot-use B`), enquanto que as restrições de dependência de DCL permitem uma maior variedade de quantificadores e tipos de dependência, como acesso a atributos e métodos, declaração de variáveis, criação de objetos, etc. Já em relação ao REQ7, LDM demanda a recriação das DSMs para verificação de conformação arquitetural sempre que ocorrerem alterações na implementação do sistema;

- SAVE: Em relação ao REQ2, SAVE não provê suporte a relacionamentos de subtipos na definição de módulos. Isso faz com que, em alguns casos, desenvolvedores tenham que manualmente ajustar o mapeamento entre o modelo de código fonte e o modelo arquitetural. Já em relação ao REQ7, SAVE requer a geração de um novo modelo de código fonte para verificação de conformação arquitetural sempre que ocorrerem alterações na implementação do sistema;

- .QL: Em relação ao REQ1, .QL não inclui um processo bem definido e específico para verificação de arquitetura, visto que trata-se de uma ferramenta que tem propósitos mais gerais do que prover conformação arquitetural. Já em relação ao REQ3, suas consultas – mesmo sendo inspiradas na sintaxe de SQL – não são tão simples.

Já em relação a outras técnicas, também existem melhorias. Por exemplo, ao contrário de linguagens de restrições lógicas e ADLs, a solução proposta inclui uma linguagem simples (atendendo ao REQ3) e uma ferramenta compatível com a linguagem Java (atendendo ao REQ6). Ainda, ao contrário da maioria das ADLs, a solução proposta não estende linguagens de programação (atendendo ao REQ4) nem requer compiladores específicos (atendendo ao REQ5).

5.3 Contribuições

As principais contribuições desta dissertação são as seguintes:

- Avaliação crítica de ferramentas e técnicas para conformação arquitetural. Esta dissertação comparou e avaliou as seguintes ferramentas: LDM (baseada em matrizes de dependência estrutural), .QL (baseada em linguagem de consulta em código fonte) e SAVE (baseada em modelos de reflexão). Além disso, foram analisadas outras técnicas que podem ser utilizadas para verificação de conformação arquitetural, tais como linguagens de restrições lógicas e linguagens de descrição arquitetural;

- Projeto da linguagem DCL. A solução para verificação de conformação arquitetural proposta na dissertação inclui uma linguagem de domínio específico, auto-explicativa, declarativa e estaticamente verificável, chamada DCL, que permite a definição de restrições estruturais entre módulos;

- Implementação da ferramenta `dclcheck`. Com o intuito de avaliar a aplicabilidade da solução proposta, foi implementado um protótipo de uma ferramenta, chamada

`dclcheck`, que verifica se o código fonte (isto é, a arquitetura concreta de um sistema) respeita restrições de dependência definidas em DCL. Esse protótipo está publicamente disponível em: `http://www.inf.pucminas.br/prof/mtov/dcl`;

- Estudo de caso. Com o intuito de demonstrar a aplicabilidade da solução proposta, a linguagem DCL e a ferramenta `dclcheck` foram aplicadas no sistema SGP, utilizado pelo SERPRO para gestão de seus empregados.

5.4 Trabalhos Futuros

Como trabalho futuro, pretende-se prosseguir com pesquisas na área de verificação de arquiteturas de software. Assim, são descritas a seguir algumas das possíveis linhas de pesquisa:

- Investigação da utilização de DCL em outros sistemas, preferencialmente que sigam padrões arquiteturais diferentes. Isso poderá trazer novas evidências sobre a relação entre erosão arquitetural e dependências inter-modulares impróprias. Além disso, esses novos estudos de caso podem contribuir para adição de novas características em DCL;

- Investigação de novas linguagens e abstrações para verificação arquitetural, capazes de detectar violações não-estruturais e/ou propriedades dinâmicas. Atualmente, a solução proposta não é capaz de detectar propriedades comportamentais ou dependências geradas por meio dos recursos de reflexão computacional de Java.

- Investigação de técnicas que permitam extrair restrições arquiteturais de forma automática ou semi-automática. Atualmente, a solução proposta não lida com o problema de extração arquitetural, o que dificulta a visualização da arquitetura pelos arquitetos de software. Assim, a exemplo das ferramentas LDM e SAVE, descritas no Capítulo 2, que extraem uma visão arquitetural de forma automática, acredita-se ser promissora uma linha de pesquisa sobre técnicas de extração automática de restrições arquiteturais;

- Investigações de métricas e modelos quantitativos que permitam detectar o grau de erosão arquitetural. A idéia é que tais modelos e métricas possam ser aplicados antes e depois da correção das violações detectadas para confirmar a melhoria arquitetural [KP07];

- Aprimorar a implementação da ferramenta `dclcheck`, incorporando novas funcionalidades como o módulo de edição de restrições de dependência. Pretende-se também integrar o processo de verificação de conformação arquitetural ao processo de compilação incremental da IDE Eclipse.

Apêndice A

Gramática da Linguagem DCL

Este apêndice apresenta a gramática completa da linguagem DCL na notação BNF (*Backus-Nahur Form*) [Sud05]. Na versão de BNF utilizada, símbolos terminais são grafados em negrito e símbolos não-terminais iniciam-se com maiúsculas. Além disso, {`A`} indica zero ou mais repetições de `A` e `[A]` indica que `A` é opcional.

```
S: ModDecl | DCDecl

ModDecl: module ModId: ModDef {, ModDef} (ModDecl | DCDecl)

ModDef: ClassName | ClassName+ | Pkg* | Pkg** | RegExpr

DCDecl:
    only RefMod can-Type {, can-Type} RefMod [DCDecl] |
    RefMod must-MustType {, must-MustType} RefMod [DCDecl] |
    RefMod Quantifier-Type {, Quantifier-Type} RefMod [DCDecl]

RefMod: (ModDef | ModId) {, RefMod}

Quantifier: cannot | can-only

MustType: extend | implement | derive | throw | annotated

Type: access | declare | handle | create | depend | MustType
```

Com o intuito de simplificar o entendimento da gramática, foram ocultadas a definição de quatro variáveis que geram terminais de uso comum: `ModId` que se refere a um nome de um identificador, `ClassName` ao nome qualificado de uma classe, `Pkg` ao nome de um pacote e `RegExpr` a qualquer expressão regular.

Bibliografia

[ACM03] Deepak Alur, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall, 2nd edition, 2003.

[ACN02] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *22nd International Conference on Software Engineering*, pages 187–197, Orlando, FL, USA, 2002.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[Bal99] Thomas Ball. The concept of dynamic analysis. *SIGSOFT Software Engineering Notes*, 24(6):216–234, 1999.

[BC99] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity*, volume 1. MIT Press, 1999.

[BCK03] L. Ball, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, 1st edition, 2003.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *30th International Conference on Software Engineering*, pages 391–400, Leipzig, Germany, 2008.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[GB02] Jilles Gurp and Jan Bosch. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GMW97] David Garlan, Robert Monroe, and David Wile. Acme: an architecture description interchange language. In *the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 1–15, Toronto, ON, Canada, 1997.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1993.

[GS96] David Garlan and Mary Shaw. *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[GZ05] Ian Gorton and Liming Zhu. Tool support for just-in-time architecture reconstruction and evaluation: an experience report. In *27th International Conference on Software Engineering*, pages 514–523, New York, NY, USA, 2005. ACM.

[HH06] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[HHR04] Daqing Hou, H. James Hoover, and Piotr Rudnicki. Specifying framework constraints with FCL. In *the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 96–110, Markham, ON, Canada, 2004.

[HLL04] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge. A survey of trace exploration tools and techniques. In *the 2004 Conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55, Markham, ON, Canada, 2004. IBM Press.

[JR97] Dean Jerding and Spencer Rugaber. Using visualization for architectural localization and extraction. In *the 1997 Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 56–65, Amsterdam, Netherlands, 1997. IEEE Computer Society.

[KC99] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engineering*, 6(2):107–138, 1999.

[KMHM08] Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. Architecture compliance checking - experiences from successful technology transfer to industry. In *12th European Conference on Software Maintenance and Reengineering*, pages 43–52, Athens, Greece, 2008.

[KMNL06] Jens Knodel, Dirk Muthig, Matthias Naab, and Mikael Lindvall. Static evaluation of software architectures. In *10th European Conference on Software Maintenance and Reengineering*, pages 279–294, Bari, Italy, 2006.

[KP88] G. E. Krasner and S. T. Pope. A cookbook for using the model–view–controller user interface paradigm in smalltalk-80. In *Journal of Object Oriented Programming*, pages 26–49, 1988.

[KP07] Jens Knodel and Daniel Popescu. A comparison of static architecture compliance checking approaches. In *IEEE/IFIP Working Conference on Software Architecture*, pages 44–53, Mumbai, India, 2007.

[LKA+95] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–354, Apr 1995.

[LV95] D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, Sep 1995.

[MDEK95] Jeff Magee, Naranker Dulay, Susan Eisenbach, and Jeff Kramer. Specifying distributed software architectures. In *5th European Software Engineering Conference*, pages 137–153, Lisbon, Portugal, 1995.

[MEG03] Nenad Medvidovic, Alexander Egyed, and Paul Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *Second International Software Requirements to Architectures Workshop*, pages 61–68, Portland, OR, USA, 2003.

[MK88] Hausi A. Muller and K. Klashinsky. Rigi a system for programming-in-the-large. In *International Conference on Software Engineering*, pages 80–87, Raffles City, Singapore, 1988.

[MK06] Kim Mens and Andy Kellens. Intensive, a toolsuite for documenting and checking structural source-code regularities. In *10th European Conference on Software Maintenance and Reengineering*, pages 239–248, Bari, Italy, 2006.

[MKPW06] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. Co-evolving code and design with intensional views: A case study. *Computer Languages, Systems & Structures*, 32(2-3):140–156, 2006.

[MMW02] Kim Mens, Isabel Michiels, and Roel Wuyts. Supporting software development through declaratively codified programming patterns. *Expert Systems with Applications,*, 23(4):405–413, 2002.

[MNS95] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *3rd Symposium on Foundations of Software Engineering*, pages 18–28, New York, NY, USA, 1995.

[MNS01] 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(4):364–380, 2001.

[MT00] 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.

[MVH+07] Oege Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *7th IEEE International Conference on Source Code Analysis and Manipulation*, pages 3–14, Paris, France, 2007.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *Software Engineering Notes*, 17(4):40–52, 1992.

[SAG+06] Bradley R. Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *9th International Symposium on Foundations of Software Engineering*, pages 99–108, Vienna, Austria, 2001.

[SJSJ05] 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*, pages 167–176, San Diego, CA, USA, 2005.

[Sud05] Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison Wesley, 3rd edition, 2005.

[TV08a] Ricardo Terra and Marco Tulio Valente. Towards a dependency constraint language to manage software architectures. In *Second European Conference on Software Architecture*, volume 5292 of *Lecture Notes in Computer Science*, pages 256–263, Paphos, Cyprus, 2008. Springer.

[TV08b] Ricardo Terra and Marco Tulio Valente. Verificação estática de arquiteturas de software utilizando restrições de dependência. In *II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software*, pages 1–14, Porto Alegre, RS, Brasil, 2008.

[TV10] Ricardo Terra and Marco Tulio Valente. A dependency constraint language to manage object-oriented software architectures. In *Software, Practice & Experience*, 2010. Aceito para publicação.

[YGS+04] 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*, pages 470–479, Edinburgh, Scotland, 2004.