ESPECIFICAÇÃO MODULAR DE RESTRIÇÕES ARQUITETURAIS

SÂNDALO CARLELO D ELRIO EUZÉBIO E BESSA

ESPECIFICAÇÃO MODULAR DE RESTRIÇÕES ARQUITETURAIS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente
Coorientador: Ricardo Terra Nunes Bueno Villela

Belo Horizonte
Junho de 2016

Bessa, Sândalo Carlelo D Elrio Euzébio e

B557e Especificação modular de restrições arquiteturais / Sândalo Carlelo D Elrio Euzébio e Bessa. — Belo Horizonte, 2016
xix, 108 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais

Orientador: Marco Túlio de Oliveira Valente
Coorientador: Ricardo Terra Nunes Bueno Villela

1. Computação – Teses. 2. Engenharia de software. 3. Software – Arquitetura. 4. Software – Validação. I. Orientador. II. Coorientador. III. Título.

CDU 519.6*32(043)

Agradecimentos

Agradeço à minha amada esposa Kelly pelo apoio incondicional. Aos meus dois queridos filhos, Samuel e Davi, agradeço por entenderem, mesmo com tão pouca idade, esse passo tão importante na vida do papai.

Agradeço aos meus colegas da PRODEMGE pela colaboração, sem a qual essa tarefa seria muito mais difícil.

Agradeço à direção da PRODEMGE pelo suporte e apoio na realização desse trabalho.

Agradeço de forma muito especial aos meus orientadores, Marco Túlio e Ricardo Terra, pela paciência e sabedoria durante a realização deste trabalho.

Agradeço a todas as pessoas que contribuíram, de forma direta ou indireta, para que eu alcançasse esse objetivo.

Resumo

Arquitetura de software pode ser considerada como o conjunto de decisões e convenções que determinam como um sistema será construído, ou seja, a arquitetura deve definir quais serão as partes fundamentais do software, qual a responsabilidade de cada uma dessas partes e como essas partes devem interagir umas com as outras. Aspectos importantes como custo, evolução, desempenho, modularidade, segurança, robustez e manutenibilidade estão associados a uma correta definição da arquitetura de um software. Portanto, garantir que as decisões arquiteturais serão seguidas diminui os riscos de projetos de software. No entanto, dada a natureza abstrata do conceito de arquitetura de software, garantir a correta implementação de decisões arquiteturais não é uma tarefa trivial. Divergências entre código fonte e arquitetura planejada podem ocorrer já em fases iniciais do desenvolvimento do software, dando origem a um fenômeno conhecido como erosão arquitetural. Algumas técnicas de conformidade arquitetural têm sido então propostas para revelar divergências entre a arquitetura planejada e o código fonte. Dentre essas abordagens, destaca-se a linguagem DCL *(Dependency Constraint Language)*, uma linguagem de domínio específico que apresenta importantes resultados no contexto de conformidade arquitetural. No entanto, a versão atual de DCL possui algumas características como, ausência de recursos de modularização e dificuldade de reúso, que podem dificultar sua adoção em cenários reais de desenvolvimento de software. Assim, nesta dissertação de mestrado, estende-se DCL com um modelo de especificação modular, reutilizável e hierárquico. Essa extensão, chamada DCL 2.0, foi avaliada em um sistema de grande porte do governo do Estado de Minas Gerais. Como principal resultado, foram detectadas 771 violações arquiteturais, sendo que 74% delas somente puderam ser detectadas devido aos novos tipos de violações propostas em DCL 2.0.

Abstract

Software architecture is the set of design decisions and conventions that determines how a system is built, i.e., software architecture specifies the main parts of the software, the responsibility of each part, and how the parts should interact with each other. Important aspects such as cost, evolution, performance, modularity, security, robustness, and maintainability are associated with the correct definition of a software architecture. Therefore, ensuring that architectural decisions are strictly followed reduces the risk of software projects. However, due to the abstract nature of software architecture concepts, ensuring the correct implementation of architectural decisions is not a trivial task. Divergences between the planned architecture and source code may occur in the early stages of the software development, which denotes a phenomenon known as software architectural erosion. Architectural Conformance Checking techniques have been proposed to tackle the problem of divergences between the planned architecture and source code. Among such techniques, we can note the DCL language (Dependency Constraint Language), which is a domain-specific language that has interesting results in architectural conformance contexts. However, the current version of DCL has some limitations, such as lack of modularity and low degree of reuse, which may prevent its adoption in real software development scenarios. In this master dissertation, we extend DCL with a reusable, modular, and hierarchical specification. We evaluate the extended DCL—named DCL 2.0 by us—in a real-world system used by public State Government of Minas Gerais, Brazil. As main result, we were able to detect 771 architectural violations where 74% of them could only be detected due to the new violations types proposed in DCL 2.0.

Lista de Figuras

Lista de Tabelas

Sumário

Capítulo 1

Introdução

A arquitetura de um software é normalmente considerada como o conjunto de decisões e convenções que determinam como o sistema de software será construído, ou seja, a arquitetura deve dizer quais são as partes fundamentais do software, qual a responsabilidade de cada uma dessas partes e como essas partes devem interagir umas com as outras. O conceito de arquitetura de software vem sendo discutido há algumas décadas [Shaw & Clements, 2006]. Outros trabalhos como Kruchten [2008]; Knodel et al. [2006] associam aspectos importantes como custo, evolução, desempenho, decomposabilidade, segurança, robustez e manutenibilidade a uma correta definição da arquitetura de um software. Rosik et al. [2011] também ressaltam os impactos positivos de boas decisões arquiteturais em atividades como manutenção e evolução de software. Outra evidência da importância de arquitetura de software se deve ao fato de que vários processos passaram a considerá-la, criando técnicas e modelos de documentação dedicados exclusivamente à captura das decisões arquiteturais [Kruchten, 1995]. Portanto, garantir que as decisões arquiteturais sejam seguidas diminui os riscos de falha de projeto de software.

1.1 Motivação e Problema

Dada a natureza abstrata do conceito de arquitetura de software, garantir a correta implementação de decisões arquiteturais não é uma tarefa trivial. Alguns trabalhos apontam que divergências entre código e arquitetura planejada podem ocorrer já em fases iniciais de desenvolvimento [Rosik et al., 2011; Knodel et al., 2008b]. Outros trabalhos apresentam cenários onde o software, ao longo de sua evolução, perde gradualmente o alinhamento entre código e arquitetura dando origem a um fenômeno conhecido como erosão arquitetural [Sangal et al., 2005; Mitschke et al., 2013; Maffort

et al., 2013a]. Em outras palavras, após a realização de várias tarefas de manutenção e evolução no software, as regras arquiteturais (decisões) vão sendo ignoradas dando origem a violações arquiteturais. As violações arquiteturais podem ser classificadas em dois grupos: aquelas que violam a estrutura hierárquica da arquitetura, neste trabalho são chamadas *Violações de Estrutura* e aquelas que violam as regras de relacionamento entre um componente e outro, nesse trabalho classificadas como *Violações de Relacionamento*. Na prática, *Violações de estrutura* revelam inconsistências relacionadas a criação do componente, ou seja, essas violações ocorrem quando questões como localização, convenção de nomes e caracterização (estereotipagem) do componente divergem do modelo arquitetural planejado. Por outro lado, *Violações de relacionamento* revelam inconsistências na forma com a qual determinado componente se relaciona com outro componente. Em um sistema orientado por objeto os relacionamentos entre componentes se dão por meio de *leitura e escrita de atributos, chamadas de método, herança e instanciação*. Na Figura 1.1 pode-se observar os dois tipos de violações ocorrendo em um sistema hipotético. Na Figura 1.1-a observa-se o modelo arquitetural planejado. Na Figura 1.1-b observa-se a projeção do que seria o código fonte do sistema. A *Violação #1* corresponde a uma violação estrutural, uma vez que o componente *Baz* não foi previsto na estrutura hierárquica da arquitetura planejada. A *Violação #2* corresponde a uma violação de relacionamento, pois, *Foo* faz chamadas a *Bar*, situação não prevista no modelo arquitetural da Figura 1.1-a.

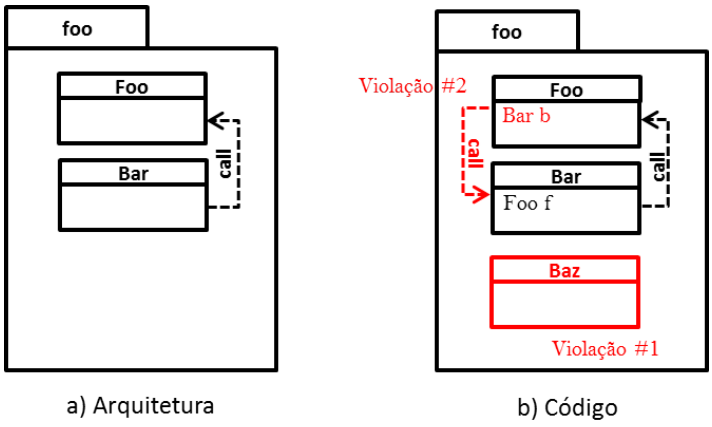

Figura 1.1: Violações de *Estrutura e Relacionamento*

Algumas técnicas vêm sendo propostas para atacar o problema de divergência entre arquitetura planejada e código fonte [Passos et al., 2010]. Ao conjunto de atividades relacionadas a essa questão dá-se o nome de técnicas de verificação de conformidade arquitetural. De maneira geral, a maioria das técnicas baseia-se em três passos: i) Especificar o modelo conceitual (arquitetura) e suas regras; ii) Definir o mapeamento

entre modelo conceitual e código; e iii) Confrontar modelo conceitual e código. Uma diferença básica entre as técnicas está no estilo de modelagem, ou seja, as técnicas se diferem na forma com que elas especificam as decisões arquiteturais. Em função do tipo de modelagem, as técnicas podem ser divididas em dois grupos: técnicas baseadas em Diagramas *(box-and-line)* e técnicas baseadas em DSL *(Domain-Specific Language)*. Exemplos dos dois tipos de abordagens são, respectivamente: Modelo de Reflexão *(RM - Reflexion Model)* proposto por Murphy et al. [1995] e a linguagem DCL *(Dependency Constraint Language)*, proposta por Terra & Valente [2009].

Em função da facilidade de uso e dos resultados já apresentados por DCL, esta dissertação aprofunda a análise sobre essa linguagem. DCL é uma linguagem declarativa que tem por objetivo controlar dependências entre módulos de sistemas orientados por objeto. Terra & Valente [2009] apresentam um estudo de caso utilizando a linguagem DCL e a ferramenta DCLSuite para verificar a conformidade arquitetural em um sistema real. Os autores apresentam como pontos positivos em favor de DCL: *expressividade*, uma vez que DCL suporta restrições arquiteturais que cobrem um amplo espectro de violações de dependência; *nível de abstração*, devido ao fato de que é possível associar partes específicas de código (baixo nível) com elementos abstratos (alto nível) via definição de módulos; *aplicabilidade*, em que os autores ressaltam a facilidade de aprendizado de DCL. No entanto, DCL apresenta algumas características que podem impedir a sua rápida adoção em cenários reais de desenvolvimento de software. Essas características são:

- *Acoplamento*: em DCL, a especificação de arquitetura é feita em um arquivo com extensão *.dcl* localizado dentro da estrutura do projeto alvo, dificultando o reúso da especificação da arquitetura em outros projetos.

- *Monolítica*: a especificação de arquitetura em DCL é feita de maneira monolítica, dificultando a manutenção das especificações, sobretudo em sistemas maiores e modularizados.

- *Modelagem não Hierárquica*: embora de fácil assimilação, a modelagem *flat* como é feita em DCL não permite estabelecer uma relação hierárquica entre os elementos arquiteturais. Essa limitação dificulta a captura do modelo arquitetural com mais precisão, uma vez que tais modelos são inerentemente hierárquicos.

- *Restrições de Estrutura*: DCL possui um amplo conjunto de restrições dedicadas a controlar violações de relacionamento entre os componentes da arquitetura. No entanto, DCL carece de certos tipos de restrições estruturais que agreguem mais poder de expressão às especificações de arquitetura.

- *Suporte à Rastreabilidade Conceitual*: avanços têm sido feitos no sentido de aumentar a produtividade no desenvolvimento de software, como linguagens de alto nível e ambientes integrados de desenvolvimento. No entanto, essas soluções tratam problemas puramente técnicos e poucas soluções são direcionadas a resolver problemas conceituais [Brooks, 1987]. Especificamente no âmbito de conformidade arquitetural, o problema persiste, uma vez que a maior parte das técnicas não oferece mecanismos que possam relacionar os elementos conceituais de um sistema com o seu código fonte. Kruchten [1995] também coloca os elementos conceituais (cenários, funcionalidades e requisitos) como sendo relevantes para todas as ações arquiteturais. Portanto, é importante que esses elementos sejam considerados no contexto de conformidade arquitetural. A Figura 1.2 ilustra um exemplo de um sistema fictício onde ocorre uma divergência entre o modelo conceitual e o código fonte. Na Figura 1.2-a, pode-se observar os conceitos (*Carro, Pessoa*) definidos para um sistema fictício. Na Figura 1.2-b observa-se o código que representa o sistema. Uma violação ocorre no artefato *ClienteUtil.java*, pois não existe na Figura 1.2-a um conceito essencial que justifique sua existência.

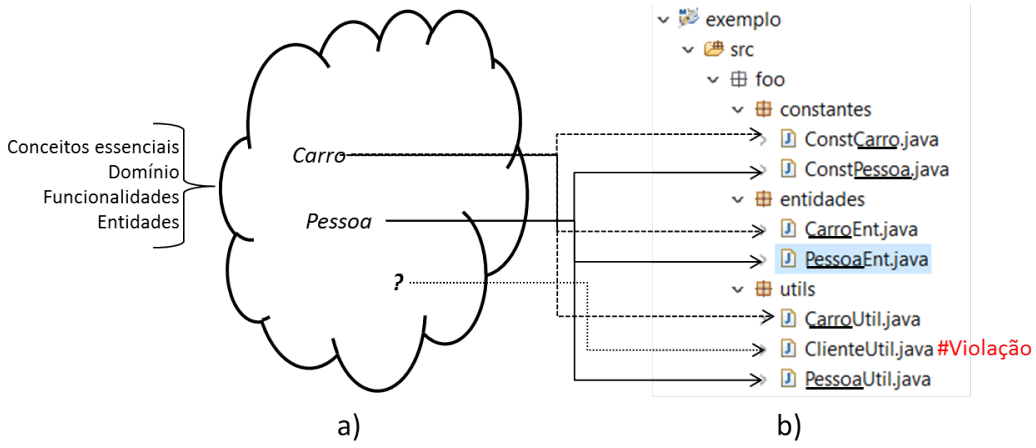

Figura 1.2: *Violação relacionada a conceitos*

1.2 Solução Proposta

1.2.1 A Linguagem DCL 2.0

Nesta dissertação é apresentada uma extensão de DCL, denominada DCL 2.0, na qual foram implementados novos conceitos e funcionalidades com o intuito de preencher lacunas da linguagem, e aumentar seu poder de expressão e reutilização. As principais extensões foram:

- *Modelagem Hierárquica*: a modelagem hierárquica foi introduzida em DCL 2.0 para permitir uma captura mais fiel dos modelos lógicos das arquiteturas de sistemas, além de facilitar a visualização da arquitetura em diferentes formatos.

- *Novos Tipos de Restrições*: novas restrições foram adicionadas à linguagem de forma a permitir um maior controle sobre violações estruturais, incluindo restrições que garantem a rastreabilidade entre conceitos essenciais e código fonte.

- *Reúso*: em DCL 2.0 a modelagem passa ser totalmente independente e desacoplada do projeto alvo. Essa mudança facilita o reúso e evolução da especificação.

- *Modularização*: em DCL 2.0, permite-se a especificação de forma modular, favorecendo o reúso de componentes entre módulos via referências.

- *Ferramenta de Suporte*: em DCL 2.0, a implementação da linguagem foi reformulada para facilitar a realização de atividades como edição, validação e visualização de arquiteturas.

1.2.2 Avaliação da Solução Proposta

Com a intenção de avaliar a linguagem proposta neste trabalho, realizou-se um estudo, no qual um sistema real de grande porte passou por um processo de conformidade arquitetural utilizando a linguagem DCL 2.0. Os resultados apresentados mostraram que a abordagem foi capaz de capturar com precisão o modelo arquitetural do sistema, além de inibir a geração de violações arquiteturais no período analisado. Após o período do estudo de caso, a ferramenta foi incorporada ao processo de desenvolvimento da empresa mantenedora do sistema avaliado. Como forma de propiciar trabalhos futuros, pode-se destacar achados importantes relacionados ao fenômeno de erosão arquitetural: 1) *Violações complexas de arquitetura são realizadas principalmente por desenvolvedores mais experientes*; 2) *violações estruturais podem esconder outros tipos de violações*; 3) *violações complexas tendem a permanecer no sistema, ocasionando a erosão arquitetural*; 4) *violações ocorrem já nas fases iniciais do sistema*; e 5) *violações ocorrem com maior frequência em manutenções evolutivas do que em manutenções corretivas*.

1.2.3 A Ferramenta

Para atestar a viabilidade da solução proposta, uma ferramenta foi construída tendo como requisitos as principais questões colocadas neste trabalho. A funcionalidade de

especificação de arquitetura foi melhorada em DCL 2.0, e outras funcionalidades como visualização e medição de arquitetura foram adicionadas. A ferramenta foi implementada na forma de um conjunto de *plug-ins* para a plataforma Eclipse.

1.3 Estrutura da Dissertação

Esta dissertação está organizada da seguinte forma: No Capítulo 1, são apresentadas a motivação e proposta de solução para as questões tratadas no trabalho. No Capítulo 2, discute-se o estado da arte acerca do tema conformidade arquitetural. Também são apresentados alguns estudos empíricos relacionados à abordagem proposta. O Capítulo 3 apresenta a proposta de solução para as questões discutidas nessa dissertação. O Capítulo 4 apresenta a ferramenta que implementa os principais conceitos e funcionalidades propostos em DCL 2.0. No Capítulo 5, reporta-se uma avaliação do uso de DCL 2.0 em uma grande empresa desenvolvedora de sistemas de software do Estado de Minas Gerais. Por fim, no Capítulo 6 são apresentadas as conclusões do trabalho, bem como são listadas algumas oportunidades de trabalhos futuros. As principais contribuições desse trabalho foram.

1. Novos conceitos e funcionalidades (DCL 2.0).
2. Um processo remoção de violações.
3. Uma ferramenta de apoio.

Capítulo 2

Trabalhos Relacionados

Com a intenção de combater erosões arquiteturais em sistemas de software, pesquisadores vêm propondo diferentes técnicas para, de alguma forma, garantir a conformidade arquitetural desses sistemas. Neste capítulo, discute-se o estado da arte sobre o tema conformidade arquitetural. Também são apresentados alguns estudos empíricos relacionados a esse tema. Finalmente, são abordados os pontos fortes e fracos de cada técnica.

2.1 Dependency Constraint Language

DCL é uma linguagem declarativa que tem por objetivo controlar dependências entre módulos de sistemas orientados por objetos [Terra & Valente, 2009, 2010, 2008]. A linguagem restringe o espectro de dependências permitidas entre os módulos de um sistema utilizando-se de uma abordagem baseada em regras de dependência entre componentes arquiteturais. Basicamente, cada regra tem sempre dois elementos e um tipo de relacionamento entre eles, como a seguinte:

$$M_A \textit{ must-extend } M_B$$

Uma restrição declarada na forma $\mathbf{M_A}$ **must-extend** $\mathbf{M_B}$ indica que toda classe pertencente ao módulo M_A deve estender uma classe pertencente ao módulo M_B. De forma geral, todas as declarações de restrição seguem essa sintaxe. Certamente, uma das principais vantagens de DCL é sua sintaxe simples e autoexplicativa. Sendo assim, uma vez que o arquiteto do sistema conheça as características e restrições arquiteturais, aplicar a técnica de DCL sobre a arquitetura do sistema é um processo relativamente simples.

Para se fazer uso de DCL é necessário a execução de algumas etapas. Primeiramente, definem-se os componentes de alto nível que, em DCL, são chamados de módulos. Em seguida, define-se o mapeamento entre os módulos e código fonte, ou seja, nessa etapa deve-se explicitar qual parte do código será representado por determinado módulo. Por fim, definem-se as restrições entre os módulos.

Uma vez definida a especificação da arquitetura na linguagem DCL, ferramentas podem ler a especificação e confrontá-la com o código fonte a fim de detectar violações arquiteturais. Basicamente, uma especificação DCL é formada por módulos e restrições entre tais módulos. Um módulo é um conjunto de classes, como no seguinte exemplo:

```
1 module Math: java.lang.Math
2 module Exception: java.lang.RuntimeException, java.io.IOException
3 module JavaUtil: java.util.*
4 module JavaSwing: javax.swing.**
```

Listagem 2.1: Especificação DCL

Neste exemplo, o módulo *Math* (linha 1) representa uma e somente uma classe, a classe *java.lang.Math*. Na linha 2, o módulo *Exception* representa duas classes, *java.lang.RuntimeException* e *java.io.IOException*. Na linha 3, o módulo *JavaUtil* representa todas as classes do pacote *java.util*. Na linha 4, o módulo *JavaSwing* refere-se a todos os tipos definidos no pacote *javax.swing* ou em quaisquer de seus sub-pacotes.

DCL classifica as violações em dois grupos: *divergência* e *ausência*. Uma *divergência* ocorre quando uma dependência existente no código-fonte viola uma restrição definida em DCL. Por outro lado, uma *ausência* ocorre quando o código-fonte não estabelece uma dependência que foi previamente definida na linguagem DCL. A fim de capturar divergências, DCL suporta a definição dos seguintes tipos de restrições entre os módulos:

- *only* M_A *can-**dep*** M_B: somente classes do módulo M_A podem depender de tipos definidos no modulo M_B.
- M_A *can-**dep**-only* M_B: classes do módulo M_A podem depender somente de tipos definidos no módulo M_B.
- M_A *cannot-**dep*** M_B: classes do módulo M_A não podem depender de tipos definidos em M_B.

Para capturar ausências, DCL suporta a definição da seguinte restrição:

- M_A *must-**dep*** M_B: classes do módulo M_A devem depender de tipos definidos no módulo M_B.

Nas declarações apresentadas, o literal ***dep*** refere-se ao tipo de dependência, a qual pode ser mais genérica como *depend* ou mais específica como *access*, *declare*, *create*, *extend*, *implement*, *throw* e *useannotation*. A Figura 2.1 (adaptada de [Terra & Valente, 2009]) resume a sintaxe da linguagem.

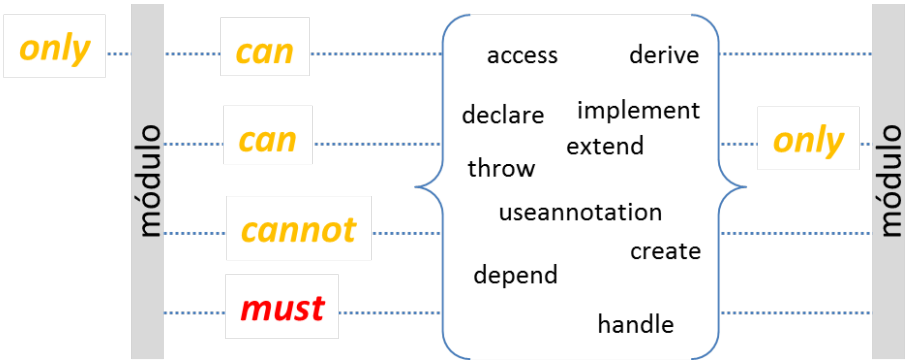

Figura 2.1: Restrições DCL

DCLSuite[1] é a ferramenta que implementa a abordagem DCL. A ferramenta é implementada sob a forma de um *plug-in* para a plataforma Eclipse. DCLSuite possui um editor para criar e editar as especificações DCL, as quais são armazenadas em um arquivo com a extensão *.dcl* que deve estar localizado na pasta raiz do projeto alvo. Na prática, DCLSuite lê este arquivo e verifica o código fonte com intuito de encontrar possíveis violações arquiteturais e, quando as encontra, elas são apresentadas como erros na *visão de Problemas* da plataforma Eclipse.

DCL também possui uma extensão, chamada DCLfix, que fornece diretrizes para desenvolvedores resolverem problemas relacionados a erosão arquitetural [Terra et al., 2012b,a, 2015]. Na prática, DCLfix sugere recomendações de refatoração para violações detectadas após um processo de conformidade arquitetural com DCL. Outra extensão para DCL, chamada *ArchRuby*, foi proposta para verificação de arquiteturas baseadas em linguagens dinamicamente tipadas [Miranda et al., 2016, 2015b,a].

2.2 Modelos de Reflexão

Primeiramente proposta por Murphy et al. [1995], essa técnica se baseia em dois modelos de componentes (baixo e alto nível) e na comparação entre eles. Componentes

[1] http://aserg.labsoft.dcc.ufmg.br/dclsuite/

de baixo nível são representados pelo código fonte. Componentes de alto nível são elementos conceituais e abstratos que representam a arquitetura do sistema. Uma etapa de mapeamento é necessária para que seja possível a comparação entre os dois modelos. Do processo de comparação entre os dois modelos, surge o modelo de reflexão que revela três tipos de relações entre os modelos: *convergência*, *divergência* e *ausência*. A abordagem considera *convergência* o código que está em conformidade com o modelo de alto nível; a *divergência* ocorre quando o código implementado está em discordância com o modelo de alto nível, causando uma violação arquitetural; e *ausência* é observada em situações onde elementos de alto nível não são identificados no código do sistema alvo, também considerada uma violação arquitetural. O uso de Modelos de Reflexão como técnica de verificação arquitetural envolve três fases distintas:

1. **Definição do Modelo de Alto Nível:** componentes de alto nível podem ser recuperados via documentação do sistema, por meio da observação do código, mas podem também ser adquiridos de maneira informal via conhecimento e experiência dos arquitetos de software. As três situações podem ocorrer de forma simultânea. Por essa razão, a abordagem sugere que, nessa fase, os arquitetos que dominam a arquitetura do sistema devem participar ativamente da construção do modelo de alto nível.

2. **Mapeamento Entre os Modelos:** o mapeamento é uma lista ordenada de entradas definidas pelo arquiteto. Cada entrada mapeia de um componente do modelo de alto nível para um ou mais artefatos no código fonte (modelo de baixo nível).

3. **Confrontação Entre Modelos:** a comparação entre os dois modelos resulta no modelo de reflexão, o qual revela *convergências*, *divergências* e *ausências*. De acordo com a abordagem, essas fases podem ser executadas várias vezes até que se obtenha o refinamento necessário, isto é, até que os dois modelos atinjam o nível de convergência desejado pelos arquitetos.

Para ilustrar a técnica de *Modelo de Reflexão*, a Figura 2.2-a (adaptada de [Terra & Valente, 2009]) apresenta o modelo de alto nível da arquitetura de um sistema em camadas. O modelo é composto por camadas e por relacionamentos entre as camadas. Esses relacionamentos refletem as restrições arquiteturais impostas à arquitetura do sistema. Em termos práticos, as regras arquiteturais aplicadas a esse modelo devem ser entendidas da seguinte forma: *Camada3* deve se comunicar apenas com *Camada2*, e *Camada2* somente com *Camada1*.

A correspondência entre os dois modelos é definida no mapeamento apresentado na Figura 2.2-b. Neste exemplo, todas as classes dos pacotes *org.foo.camada3.x* e *org.foo.camada3.y* (Figura 2.2-c) são mapeadas para o módulo *Camada3*. Todas as classes de *org.foo.camada2* mapeiam para o módulo *Camada2*. Por fim, as classes em *org.foo.camada1* correspondem ao módulo *Camada1*. O código fonte do sistema é apresentado na Figura 2.2-c. Finalmente, o modelo de reflexão resultante da comparação entre os dois modelos é apresentado na Figura 2.2-d.

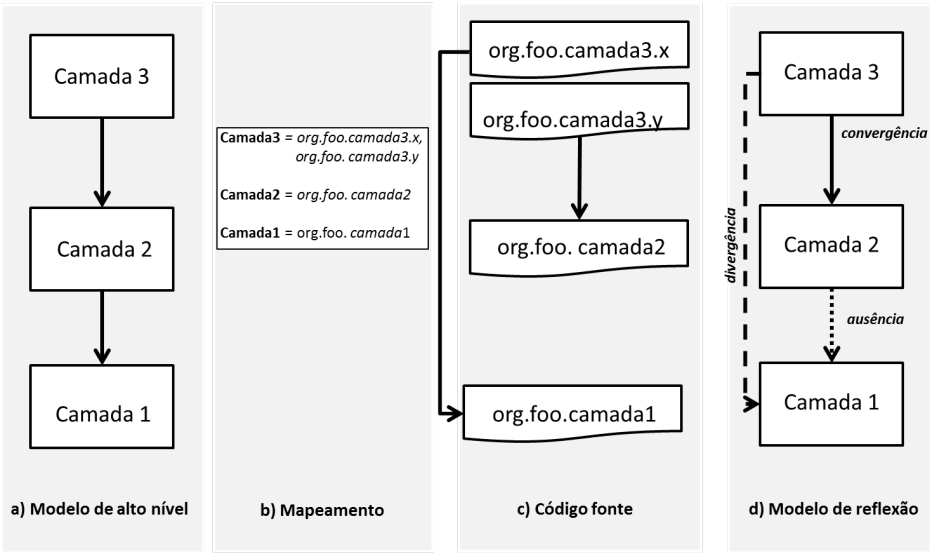

Figura 2.2: Exemplo da técnica de Modelo de Reflexão.

No exemplo, uma divergência (indicada pela linha tracejada) foi encontrada uma vez que o módulo *Camada3* está usando diretamente serviços providos pelo módulo *Camada1* e tal situação não está prevista no modelo de alto nível (Figura 2.2-a). Uma ausência (indicada pela linha pontilhada) também foi revelada uma vez que o módulo *Camada2* não está usando os serviços providos por *Camada1*, situação definida no modelo de alto nível e não refletida no código. Observa-se que a abordagem também revela convergências, por exemplo, a comunicação prevista entre *Camada3* e *Camada2* está corretamente implementada. Como Murphy et al. [1995] relatam, informações sobre convergências são importantes, uma vez que o modelo de reflexão pode também ser usado como uma ferramenta para se obter conhecimento acerca da arquitetura do sistema.

Existem ferramentas que implementam os conceitos de modelo de reflexão, por exemplo a ferramenta SAVE[2] (Software Architecture Visualization and Evaluation) [Miodonski et al., 2004]. SAVE é disponibilizado na forma de *plug-in* para a

[2]http://www.fc-md.umd.edu/save/

plataforma Eclipse, o qual permite que usuários (arquitetos) possam modelar e validar arquiteturas de software. SAVE tem visões e funcionalidades que apoiam os arquitetos em tarefas arquiteturais. A Figura 2.3 traz um diagrama apresentado por Knodel et al. [2006], o qual mostra o resultado de uma verificação de conformidade arquitetural baseada em modelos de reflexão utilizando SAVE, onde marcas de checagem representam as convergências, marcas de exclamação são as divergências e *Xs* são as ausências. Quando ocorre mais de uma situação na mesma aresta (convergência, divergência e ausência), elas são agrupadas em uma marca de interrogação.

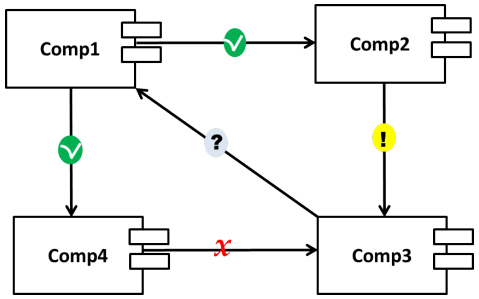

Figura 2.3: Verificação de conformidade arquitetural com a ferramenta SAVE.

Na implementação original de SAVE, o processo de verificação de conformidade arquitetural é executado manualmente pelos arquitetos. No entanto, existe uma variante da ferramenta, chamada SAVE *live*, a qual fornece aos desenvolvedores respostas imediatas de verificação arquitetural à medida que o código fonte e a arquitetura evoluem [Knodel et al., 2008b].

2.3 Matrizes de Dependências Estruturais

Apresentada por Baldwin & Clark [1999], matrizes de dependências estruturais (DSM do inglês *Dependency Structure Matrix*) foram inicialmente usadas para explicar a importância de projetos modulares na indústria de hardware. Em seguida, Sullivan et al. [2001] demostraram que a técnica também poderia ser utilizada na indústria de software. O conceito é baseado em uma matriz quadrada, na qual a interseção entre linhas e colunas denota a relação entre classes de um sistema orientado por objetos. Como ilustra a Figura 2.4-a, um **X** na coluna **A** e linha **B** da DSM significa que a classe (ou módulo) **A** depende da classe (ou módulo) **B**.

Outro interessante aspecto dessa técnica é a possibilidade de se visualizar o quanto determinado componente está associado com outro. Essa métrica é provida pelo número de referências informado na célula de interseção entre os dois elementos. Se a célula **(A/B)** mostra o número 10, essa informação revela que existem 10 referências de **A**

para **B**, como ilustrado na Figura 2.4-b. Adicionalmente, com o intuito de facilitar a análise das relações entre elementos, DSM suporta agrupamento de componentes. A Figura 2.4-c mostra a DSM resultante após os pacotes *foo.m2.a* e *foo.m2.b* terem sido agrupados em um módulo chamado **M2**, e pacotes *foo.m1* e *foo.m0* terem sido renomeados para **M1** e **M0**, respectivamente. A estratégia de agrupamento permite aos arquitetos trabalhar com DSM de forma hierárquica, permitindo que a arquitetura do sistema alvo possa ser analisada em diferentes níveis de abstração.

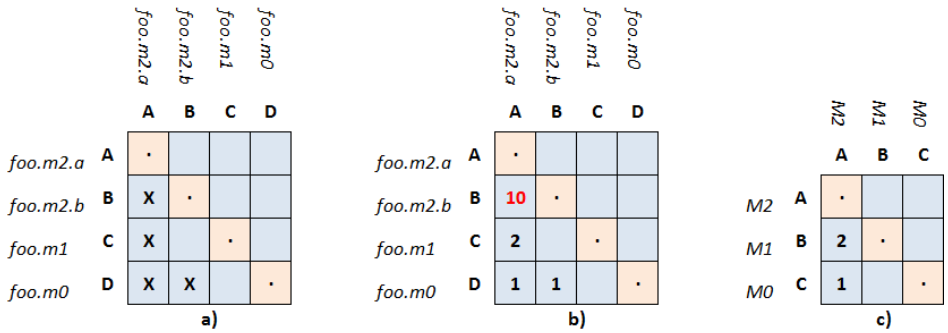

Figura 2.4: Exemplo de DSM.

[Sangal et al., 2005] propõem uma ferramenta chamada **Lattix**[3], baseada no conceito de DSM, para monitorar arquiteturas de software usando matrizes de dependências. A ferramenta extrai as dependências do código e revela potenciais problemas relacionados a essas dependências. Lattix combina matrizes de dependências estruturais com a definição de regras textuais, portanto, arquitetos podem especificar regras arquiteturais entre os componentes.

Basicamente, a definição de uma regra pode ocorrer de duas formas: *A* ***can-use*** *B* e *A* ***cannot-use*** *B*, ou seja, classes do módulo *A* podem depender de classes do módulo *B*, ou classes do módulo *A* não podem depender de classes do módulo *B*. Para ilustrar a técnica de DSM, Sangal et al. [2005] usam um sistema estritamente em camadas, para o qual foram especificadas regras de dependências entre as camadas. O sistema alvo é dividido em cinco módulos: *application, model, domain, framework* e *util*. A Listagem 2.2 mostra o exemplo da especificação do sistema em camadas usando Lattix.

[3]http://lattix.com/product-overview

```
1 $root cannot-use $root
2 application can-use application, model
3 model can-use model, domain
4 domain can-use domain, framework
5 framework can-use framework, util
6 util can-use util
```

Listagem 2.2: Especificação Lattix

Na linha 1, a regra *$root* ***cannot-use*** *$root*—onde *$root* refere-se ao nó de nível mais alto na hierarquia—indica que, em princípio, nenhum módulo pode usar outro módulo. Sangal et al. [2005] ressalta que, em sistemas estritamente baseados em camadas é mais conveniente especificar regras em módulos de mais alto nível. *Lattix* explora o conceito de estrutura hierárquica da *DSM* para alcançar esse objetivo. Sendo assim, uma vez que uma regra é especificada para um componente, ela é automaticamente propagada para os componentes filhos. No entanto, qualquer sub-módulo pode sobrescrever regras de módulos superiores na hierarquia. Sobrescritas são usadas nas declarações das linhas 2 a 6. Por exemplo, a linha 2 flexibiliza a regra geral, linha 1, permitindo que o módulo *application* possa usar o próprio módulo *application* e também o módulo *model*.

2.4 Linguagens de Consulta em Código Fonte

De Moor et al. [2007] descrevem Linguagens de consulta em código fonte (SCQL do inglês *Source Code Query Language*) como uma técnica para auxiliar na localização de determinados elementos de código. No contexto de conformidade arquitetural, a abordagem pode ser usada para detectar padrões de codificação que não estão de acordo com a arquitetura desejada. A semântica de uso de SCQL é bastante similar à SQL (*Structured Query Language)*, o que permite aos autores afirmar que essa similaridade é um facilitador na adoção da abordagem, uma vez que muitos desenvolvedores já tem familiaridade com SQL.

Assim como SQL, SCQL também é baseado em consultas, ou seja, consultas resultam em tuplas contendo informações sobre o código fonte. O resultado pode ser restringido por uma cláusula *where* de forma similar ao que ocorre em SQL. Adicionalmente, a linguagem SCQL é enriquecida com novas funções, as quais também podem ser utilizadas para restringir os resultados. Essas funções são disponibilizadas para os arquitetos por meio de novas palavras chaves, além daquelas já existentes em SQL, por exemplo, *getPackage()* recupera o pacote de um determinado tipo. Para verificar a

conformidade arquitetural de um sistema alvo, os arquitetos devem escrever consultas (*queries*) para cada restrição arquitetural.

Uma das linguagens mais conhecidas relacionadas à abordagem de SCQL é a linguagem .QL, que implementa uma linguagem de propósito geral que pode ser usada em tarefas de análise estática de código e, consequentemente, em tarefas de verificação de arquitetura. Para ilustrar a técnica de SCQL, considere um sistema baseado em camadas contendo os módulos M0 e M1. A Listagem 2.3 descreve uma consulta usando .QL com a intenção de detectar acessos não autorizados ao módulo M0 feitos por qualquer módulo que não seja o próprio módulo M0 ou o módulo M1.

```
1  from RefType type_x , RefType type_y
2  where
3      type_x.fromSource()
4      and not (type_x.getPackage().getName().matches("foo.m0")
5          or type_x.getPackage().getName().matches("foo.m1"))
6      and type_y.getPackage().getName().matches("foo.m0")
7      and depends (type_x,type_y)
8  select type_x as Type,
9      "Violação Arquitetural :" + type_x.getQualifiedName()
10     " usa " + type_y. getQualifiedName() as Violation
```

Listagem 2.3: Especificação usando .QL

Na Listagem 2.3, *RefType* representa os tipos existentes no código fonte. Comparando com SQL, *RefType* poderia ser uma tabela contendo informações de todos os tipos (classes e interfaces) de um determinado sistema. Na linha 1 tem-se uma *"junção"* da tabela *RefType* que é restringido pela cláusula *where* na linha 2. A função *fromSource()* (linha 3) restringe a busca ao projeto corrente. A função *getPackage()* (linhas 4-6) retorna o pacote onde os elementos foram declarados. A consulta procura por tipos no projeto corrente que não fazem parte dos módulos M0 or M1 (linhas 4-5) e que dependam (*depend*) de algum tipo existente em M0 (linha 6-7). No fim, a consulta projeta os nomes dos tipos envolvidos como uma violação arquitetural (linha 8-10).

2.5 Vespucci

Mitschke et al. [2013] propõem um modelo modularizado de especificação e controle de dependência baseado em blocos de especificação de arquitetura chamados *slices*, os quais são constituídos de pequenos blocos conceituais nomeados *ensembles*. Os autores argumentam que a abordagem permite a formalização de arquiteturas em diferentes

níveis de abstração, combinando elementos de mais alto nível com elementos de mais baixo nível por meio de um modelo hierárquico de especificação de arquitetura. Os autores defendem que a abordagem também facilita a manutenção e evolução de especificações de arquitetura de software, uma vez que elas são feitas de maneira hierárquica e modular. A técnica combina diagramas *box-and-line*, que são usados para definir a estrutura da arquitetura, com linguagem textual para fazer o mapeamento entre arquitetura e código. Cada *box* representa um elemento conceitual, o *ensemble*, e as linhas representam qualquer tipo de dependência permitida entre dois *ensembles*. Os principais elementos de *Vespucci* são:

Ensembles: são blocos conceituais reutilizáveis que representam o código fonte. Especificamente, *ensembles* representam agrupamentos de artefatos de código, como classes, métodos e atributos. Mitschke et al. [2013] referem-se ao conjunto de artefatos de código representados por um *ensemble* como *ensemble's extension*. A notação visual de um *ensemble* é um retângulo (*box*), contendo o nome como rótulo, inseridos em um *slice*. A Figura 2.5 ilustra dois *ensembles*, um chamado *Service* e outro chamado *DAO*.

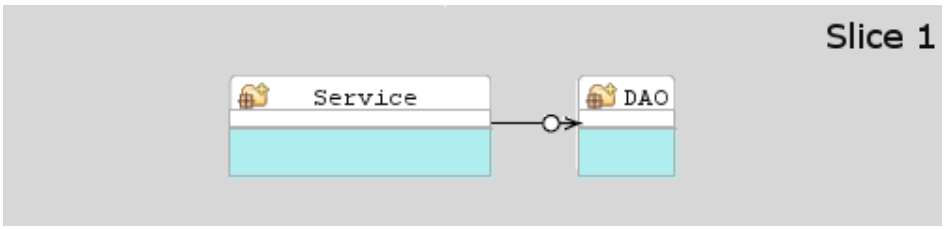

Figura 2.5: Notação visual de um *ensembles*.

Slice: é uma unidade de especificação que representa uma porção menor da arquitetura, uma fatia, onde as restrições são aplicadas aos *ensembles* de forma modular. As restrições são formalizadas via conexões entre os *ensembles*. Visualmente, restrições são representadas por linhas entre os *ensembles* e podem ter escopo global ou local. Restrições globais afetam todos os *slices* e são caracterizadas pelo símbolo *"o"* posicionado na linha entre os *ensembles*, enquanto que no escopo local as restrições são aplicadas para um *slice* e são caracterizadas pela ausência do símbolo *"o"*. Na Figura 2.5, a linha de *Service* para *DAO* representa uma restrição de escopo global, ou seja, para todo *ensemble* no repositório de *ensembles*, essa é a única dependência permitida para o *ensemble* DAO. Adicionalmente, é possível refinar o tipo das restrições, especificando o tipo de dependência entre os *ensembles*. Os tipos de dependência podem ser, por exemplo, *leitura e escrita de atributos, chamada de método, herança* e

instanciação, ou seja, dependências tipicamente encontradas em linguagens orientadas por objetos.

Linguagem de consulta: permite associar elementos conceituais (*ensembles*) com os artefatos do código fonte. Para cada *ensemble*, haverá sempre uma consulta (query) associada. A linguagem oferece alguns predicados básicos que podem ser usados para selecionar classes, pacotes, ou outros arquivos para formar os *ensembles*. Por exemplo, a seguinte declaração define um *ensemble* chamado *Helpers*:

package("org.foo.helpers").

De fato, essa declaração define que todas as classes do pacote *"org.foo.helpers"* são *ensemble's extension* do *ensemble Helper*.

Repositórios de Ensemble: a abordagem propõe o uso de um repositório de *ensembles* onde devem ser armazenadas as definições de todos os *ensembles*. O repositório serve como um módulo de reúso para os projetos e proporciona um ponto de partida para a modelagem de aplicações que necessitam fazer controle de dependências. De forma geral, o propósito do repositório é permitir o reúso de *ensembles* entre projetos, garantindo que todos os *slices* façam referência aos mesmos *ensemble's extension* (artefatos de código), para um particular *ensemble*. Outra função importante do repositório é permitir uso de restrições globais e locais.

Ferramenta de verificação: a ferramenta verifica a consistência entre a arquitetura planejada e a arquitetura implementada. Em geral, o processo de especificação e verificação usando *Vespucci* ocorre em três fases:

1. **Definição de *Ensembles*:** arquitetos definem os *ensembles* usando a linguagem de consulta para selecionar o código fonte que será representado por cada *ensemble*. Os *ensembles* são armazenados no repositório.

2. **Definição de *Slices*:** os arquitetos definem as partes da arquitetura para as quais se deseja definir as regras para controle de dependência, criando *slices* e reusando *ensembles*. Como mencionado acima, *slices* são os módulos de especificação onde são definidas as restrições entre *ensembles*.

3. **Verificação:** a ferramenta avalia cada *slice* para verificar se as conexões entre os *ensembles* estão sendo violadas.

Os autores também descrevem uma ferramenta chamada *Vespucci dependency checker*[4], que é integrada ao IDE Eclipse na forma de *plug-in*. A ferramenta possui algumas visualizações que suportam a criação de *ensembles* e *slices*, como ilustrado na Figura 2.6 reproduzida de [Mitschke et al., 2013].

Figura 2.6: Visualização de *ensembles* e *slices*

2.6 Testes de *Design*

Teste de *Design* é uma abordagem proposta por Brunet et al. [2009], na qual a conformidade arquitetural é verificada usando testes automatizados de maneira similar ao que é realizado em técnicas de testes funcionais. De acordo com a abordagem, embora semelhantes, testes de *design* diferem de testes funcionais principalmente pelas suas intenções de uso. Testes funcionais são usados para validar se o software atende a seus requisitos externos, enquanto testes de *design* são usados para validar se os componentes internos do software estão construídos corretamente. Segundo os autores, a principal vantagem de testes de *design* está no fato de que os arquitetos não precisam aprender uma nova linguagem de programação. Em outras palavras, eles podem escrever seus testes usando a mesma linguagem adotada na implementação do software.

Para apoiar o uso de testes de *design*, os autores desenvolveram a ferramenta *Design Wizard*, que é uma *API* que possibilita escrever testes arquiteturais usando o *framework* de teste JUnit. A Listagem 2.4 ilustra um exemplo da utilização de *Design Wizard API*.

[4]http://www.opal-project.de/vespucci_project

```
1  public class LayerTest extends TestCase {
2      public void testDAOCall() {
3          DesignWizard dw = new DesignWizard("mysystem.jar");
4          PackageNode dao = dw.getPackage("org.mysystem.dao");
5          PackageNode ctr = dw.getPackage("org.mysystem.ctr");
6          Set<ClassNode> callers = null;
7          for (ClassNode clazz : dao.getAllClasses()) {
8              callers = clazz.getCallers();
9              for (ClassNode caller : callers) {
10                 assertTrue(caller.getPackage().equals(dao)
11                 || caller.getPackage().equals(ctr))
12             }
13         }
14     }
15 }
```

Listagem 2.4: Exemplo de um teste de *Design*

Na Listagem 2.4, o teste verifica uma regra arquitetural para um sistema fictício chamado *mysystem*. O teste verifica se classes do pacote *dao* estão sendo chamadas somente por classes pertencentes ao pacote *controller (ctr)*. Nas linhas 1 e 2, mostra-se uma classe e um método respectivamente, definidos de acordo com o padrão JUnit. Na linha 3 é realizada a instanciação da classe *DesignWizard*, a qual recebe como parâmetro o nome do arquivo *.jar* com o conjunto de classes sobre as quais o teste será realizado. Nas linhas 4 e 5 temos a declaração de duas variáveis do tipo *PackageNode*, as quais representam, respectivamente os pacotes *org.mysystem.dao* e *org.mysystem.ctr*. Na linha 7, observa-se o comando *for* iterando sobre todas as classes do pacote *dao*. Na linha 9, observa-se outro comando *for* iterando em cada classe que faz alguma chamada às classes do pacote *dao*. Na linha 10, é chamado o método *assertTrue*, o qual verifica se chamadas de métodos são realizadas apenas por classes de *controller (ctr)* ou por classes do próprio pacote *dao*.

2.7 Framework-Specific Model Language

Atualmente vários *frameworks* são utilizados como parte de arquiteturas de software como forma de acelerar o desenvolvimento dos sistemas. No entanto, a utilização dessas plataformas não é uma tarefa trivial, como relatado por Brunet et al. [2015] onde um estudo é apresentado reportando desafios e dificuldades de se estender a plataforma Eclipse. A instanciação incorreta desses *frameworks* pode trazer problemas à arquitetura de um software e anular o ganhos de produtividades que, em tese, seriam trazidos pelo *frameworks*. Antkiewicz & Czarnecki [2006] sugerem FSML como uma alternativa para esse problema. Derivada do conceito de DSML *(Domain Specific Model Language)* apresentado por Roberts & Johnson [1997], FSML, em contraste com outras

abordagens, não se baseia apenas no controle de dependência, mas sim, em um rígido modelo de instanciação de *frameworks* que garante a conformidade arquitetural, principalmente em sistemas que fazem o uso intenso de *frameworks*. A abordagem propõe a criação de um FSML para cada *framework*, logo, cada *FSML* contém informações e regras necessárias para se utilizar corretamente um determinado *framework*. A abordagem propõe o mapeamento de cada ponto de extensão do *framework* e de suas regras de instanciação de forma que essa definição possa ser usada para instanciar, entender e verificar a arquitetura de sistemas baseados em *frameworks*.

As seguintes fases são necessárias para o uso da abordagem. (i) definir o modelo conceitual, chamado de componentes de alto nível por outras abordagens; (ii) realizar o mapeamento entre o modelo conceitual e os artefatos concretos, tais como: classes, pacotes e arquivos; e (iii) utilizar uma ferramenta para instanciar o *framework*. A técnica pode ser usada para várias tarefas arquiteturais, por exemplo, análise, validação e instanciação.

Lee et al. [2008] apresentam uma ferramenta baseada no conceito de *FSML* nomeada *framework-specific IDE extensions*, a qual permite que qualquer *framework* que tenha sua arquitetura formalizada com *FSML* possa usar vários recursos de uma IDE, tais como visualização, geração e verificação de código. A funcionalidade de verificação permite garantir as restrições declaradas no *FSML* de cada *framework*, logo pode ser usada como uma técnica de conformidade arquitetural.

2.8 ArchLint

ArchLint é uma abordagem que utiliza-se da combinação de técnicas de análise estática e histórica para verificação de conformidade arquitetural [Maffort et al., 2013a,b, 2016]. Os autores apresentam resultados onde foi possível detectar ausências (falta de uma dependência) e divergências (dependências indesejadas) com precisão superior a 50%. Eles ainda argumentam como pontos positivos a favor da utilização da técnica: primeiro, o fato de que a mesma não necessita de uma especificação muito detalhada da arquitetura, diminuindo a dependência dos arquitetos dos sistema; segundo, o processo de detecção de violações é realizado via análise estática código e de forma não-invasiva, uma vez que nenhuma alteração é necessária no código fonte. Sendo assim, o impacto nas atividades do processo de desenvolvimento é bastante pequeno.

Obrigatoriamente a técnica necessita de duas fontes de informações para realizar as validações: (a) histórico de versões do sistema; (b) um modelo documental de alto nível do sistema. O modelo de componentes deve incluir informações que permitam

identificar os componentes por meio de seus nomes. Em uma estratégia combinada de análise estática código, análise do histórico de alterações e um conjunto de heurísticas, ArchLint classifica as dependências (ou a falta delas) como suspeitas baseando-se em hipóteses de frequência de uso e manutenções realizadas sobre essas dependências. ArchLint considera todas as dependências estáticas estabelecidas entre classes do sistema alvo. A Figura 2.7 (adaptada de [Maffort et al., 2013a]) apresenta o modelo geral do uso de ArchLint para detecção de violações arquiteturais.

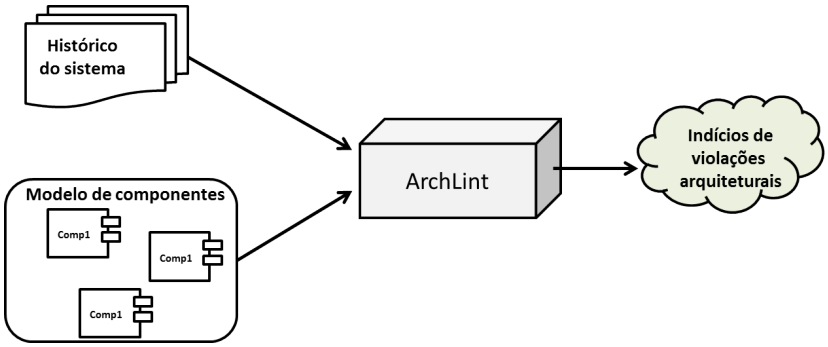

Figura 2.7: Modelo ArchLint

2.9 Estudos Empíricos

Com o intuito de entender o comportamento de equipes de desenvolvimento de software em relação a violações arquiteturais, Rosik et al. [2011] realizaram uma avaliação durante o desenvolvimento de um sistema de software comercial real, em que *Modelos de Reflexão* foram utilizados para detectar violações arquiteturais. O estudo centrou-se em responder a seguinte pergunta: *Uma vez que uma violação é detectada, ela é removida do código-fonte?* A resposta obtida no estudo foi *não*. Embora a ferramenta tenha revelado diversas violações, os membros da equipe não se sentiram motivados e seguros para removê-las do código fonte. Os autores argumentaram que, embora a técnica de *Modelos de Reflexão* tenha sido capaz de encontrar várias violações durante o desenvolvimento os resultados indicam que, em situações específicas, algumas violações foram ocultadas devido a forma de mapear as restrições proposta pela ferramenta SAVE. Por exemplo, na modelagem entre um componente A e um componente B, pode ser criada uma aresta para permitir que a classe B possa usar serviços da classe A. No entanto, hipoteticamente, poderia existir uma regra na qual B não devesse estender A, ou seja, a dependência é válida se a intenção for uma chamada de métodos da classe A, mas não é válida se o propósito for extensão da classe. Os autores relatam que essa situação ocorre porque cada aresta no RM pode representar diversos tipos de relacionamento

(e.g.,chamada de métodos, herança ou declaração) simultaneamente. Nesta situação, arestas "consistentes", podem representar dependências válidas no código-fonte (B chamar A), mas também podem esconder relacionamentos proibidos (B estender A).

Brunet et al. [2015] analisaram cinco anos de evolução da plataforma Eclipse IDE para entender melhor as violações ocorridas durante o desenvolvimento dessa plataforma. Os pesquisadores avaliaram quais tipos de regras foram estabelecidas pelos desenvolvedores. Além disso, eles investigaram as características das violações arquiteturais e também investigaram como os desenvolvedores trataram tais violações. Como resultado, os autores ressaltam o alto número de restrições relacionadas ao controle de extensão de classes ou interfaces, contrastando com a literatura onde a maioria dos trabalhos relacionados à validação de conformidade arquitetural baseiam-se em regras do tipo "A não pode depender de B". Ao analisar vários relatórios relacionados com a plataforma Eclipse e entrevistar vários desenvolvedores, os autores apresentaram algumas observações interessantes, como:

- *"Regras não são criadas para culpar, mas para orientar, sobretudo, aqueles desenvolvedores que não estão cientes do que eles estão fazendo."*

- *"Proibir todos os tipos de dependência é, por vezes, muito restritivo para os desenvolvedores. Em alguns cenários, é preciso permitir o uso da API, mas também é necessário controlar a extensão dos tipos e em menor grau, controlar a instanciação desses tipos."*

- *"Desenvolvedores não removem a maioria das violações, principalmente se o custo de refatoração é alto ou se determinada violação é considerada uma exceção."*

Por fim, os autores afirmam que são necessários estudos com outros sistemas a fim de generalizar as suas conclusões.

Knodel et al. [2008b] introduzem o conceito de *Verificação de Conformidade Construtiva* na abordagem de *Modelos de Reflexão*. Em geral, a técnica consiste em fornecer respostas instantâneas ao desenvolvedor em relação às violações. À medida que o software vai sendo desenvolvido, os desenvolvedores recebem alertas no IDE Eclipse, caso seu código esteja violando alguma regra arquitetural. Para avaliar o conceito, uma avaliação foi realizada durante o desenvolvimento de um software acadêmico. Os resultados mostraram que equipes de desenvolvimento que utilizaram *Verificação de Conformidade Construtiva* inseriram cerca de 60% menos violações na arquitetura do que aqueles que não utilizaram. No entanto, os autores apresentam como ameaças à validade do experimento o fato de que ele foi executado em um sistema acadêmico.

[Knodel et al., 2008a] apresentam as lições aprendidas e experiência adquirida com implantação de técnicas de conformidade arquitetural em um caso real da indústria. Os autores argumentam que ferramentas em estágio de protótipos não são adequadas para se fazer uma avaliação em cenários reais e geralmente exigem esforço adicional de adaptação. É ressaltada a necessidade de se ter cenários reais para se testar novas abordagens antes de torná-las disponíveis às fábricas de software. No estudo, é utilizada a ferramenta SAVE, a qual é aplicada em uma linha de produtos de software embarcados. Os resultados mostraram que a técnica foi capaz de diminuir o grau de violação para a maioria dos produtos da linha para menos de 5%.

Terra & Valente [2009] realizaram um estudo de caso utilizando a linguagem DCL (*Dependency Constraint Language*) e a ferramenta DCLSuite para aferir a conformidade arquitetural em um sistema real. Segundo os autores, a principal motivação do trabalho foi comprovar a hipótese de que violações ocorrem à medida que relações de dependência inadequadas entre os módulos vão sendo estabelecidas no código fonte. No experimento, a ferramenta DCLSuite é aplicada em três versões de um sistema de gestão de recursos humanos. Após acompanhar a evolução do sistema nas duas versões anteriores, os autores, ao avaliarem a terceira versão, apresentam resultados que atestam que DCL foi capaz de identificar 179 classes ou 8% das classes do sistema com algum tipo de violação arquitetural. Por fim, os autores argumentam os pontos positivos em favor de DCL: *expressividade*, uma vez que DCL suporta vários tipos de restrições arquiteturais que cobrem um amplo espectro de violações de dependência; *nível de abstração*, devido ao fato de que é possível associar partes específicas de código (baixo nível) com elementos abstratos (alto nível) por meio da definição de módulos; *aplicabilidade*, em que os autores ressaltam a facilidade de aprendizado de DCL, a natureza não intrusiva e o bom desempenho de DCLSuite. Essas características favorecem a aplicação da técnica em cenários reais de desenvolvimento de software.

2.10 Discussões Finais

Este capítulo apresentou uma visão geral de trabalhos relacionados ao tema desta dissertação. Foram apresentadas várias técnicas e suas respectivas ferramentas destinadas a detecção de violações. Por fim, com o propósito de apresentar uma visão prática do tema, foram apresentados alguns estudos relacionados às técnicas mencionadas anteriormente. As considerações para cada técnica são:

Dependency Constraint Language: o ponto forte de DCL está na facilidade

com que a técnica pode ser aplicada, isto é, a técnica possui conceitos simples e fáceis de serem aprendidos. São necessárias poucas construções (palavras chaves), as quais podem ser combinadas para criar um amplo número de restrições arquiteturais, favorecendo sua expressividade. O fato de ser uma técnica não intrusiva também é um fator importante, uma vez que não é exigida nenhuma intervenção no código alvo para que DCL seja utilizada. DCL pode ser uma boa escolha quando a intenção é o controle de dependências em sistemas orientados por objetos. No entanto, DCL não é uma linguagem completa para especificação de arquiteturas. Seu foco principal está no controle de dependência como forma de garantir a integridade arquitetural, e esse controle pode não ser suficiente para se capturar todos os aspectos arquiteturais, uma vez que outras características estruturais dos componentes também são importantes no contexto de arquitetura de software [Zapalowski et al., 2014]. Por exemplo, convenção de nomes. Em DCL, as definições são realizadas de maneira monolítica e a falta de modularidade em especificações arquiteturais pode ser um problema [Mitschke et al., 2013]. Por exemplo, em DCL não é possível criar componentes de forma hierárquica, situação comum em modelagens orientadas por objeto e um aspecto importante para lidar com vários níveis de decomposição [Koschke & Simon, 2003; Knodel, 2002]. Por fim, DCL não aponta convergências explicitamente, aspecto relevante para a compreensão da arquitetura do sistema [Murphy et al., 1995].

Modelos de Reflexão: dentre as abordagens citadas, *Modelos de Reflexão* é uma das mais utilizadas. Existem vários estudos e experimentos que mostram que a técnica pode ser usada de forma efetiva em diferentes cenários [Rosik et al., 2011; Knodel et al., 2008a]. Em geral, os principais conceitos de *Modelos de Reflexão* são de fácil entendimento e bastante intuitivos [Knodel & Popescu, 2007]. Isto é um ponto positivo em uma possível adoção da técnica. No entanto, não se percebe na técnica aspectos relacionados a reúso, isto é, não é simples reutilizar *Modelos de Reflexão* em diferentes sistemas com arquiteturas similares. Esse aspecto é caracterizado como falta de *reusabilidade (facilidade de reúso)* por Knodel & Popescu [2007], cujo trabalho também reporta a dificuldade em se reutilizar definições de *Modelos de Reflexão* já existentes quando se avalia um novo sistema ou uma versão diferente do mesmo sistema. Assim, a falta de *reusabilidade* pode ser uma barreira para a adoção da técnica.

Matrizes de Dependências Estruturais: a simplicidade dos conceitos de DSMs facilitam seu uso em tarefas de conformidade arquitetural. Com pouco esforço por parte dos arquitetos, é possível se ter uma visão geral da arquitetura em termos de dependência entre os componentes. Outra característica interessante de DSMs é a

possibilidade de se obter a matriz de dependência sem a necessidade de mapeamento de componentes de alto nível, como ocorre em outras abordagens. Por outro lado, essa mesma característica impede o reúso em outros sistemas, uma vez que a formalização de conceitos de alto nível não existe.

Linguagens de consulta em código fonte: é uma técnica bastante flexível para se usar na busca por padrões de códigos indesejados (violações). A similaridade com SQL facilita sua adoção, embora as funções baseadas em orientação por objetos adicionadas à linguagem a tornem relativamente complexa. A abordagem também não parece interessante para capturar detalhes de arquitetura antes de realizar as verificações, pois não é intuitivo expressar detalhes arquiteturais por meio de consultas em código fonte.

Vespucci: a técnica traz conceitos interessantes como modularidade na definição de arquitetura e reúso de componentes por meio do conceito de *ensembles*, *slices* e *repositórios*. Outra importante característica de *Vespucci* é a possibilidade de se trabalhar em vários níveis de decomposição por meio de *ensembles* aninhados. Um ponto de reflexão sobre a técnica é que existem poucos estudos atestando a sua aplicabilidade em cenários reais. Outro ponto de dificuldade em relação a abordagem, diz respeito a ferramenta de apoio, a qual possui pouca documentação e parece estar em processo de descontinuação.

Testes de *Design*: a abordagem propõe a implementação de teste de arquitetura de forma similar a implementação de testes funcionais. Esse fator facilita a absorção da técnica e diminui a curva de aprendizagem, uma vez que os usuários da ferramenta já possuem domínio sobre o uso do *framework* de testes. O ponto negativo é que a técnica não favorece a modelagem de arquitetura, isto é, não é intuitivo obter detalhes arquiteturais a partir do código fonte dos testes, o que pode prejudicar o entendimento do sistema.

Framework-Specific Model Language: *FSML* é diferente de outras abordagens, uma vez que sua proposta não baseia-se somente no controle de dependências estruturais. A ideia central de *FSML* é auxiliar na correta instanciação de arquiteturas, principalmente arquiteturas baseadas em *frameworks*. *FSML* alcança a conformidade arquitetural de um sistema ao avaliar se sua arquitetura foi corretamente instanciada. A abordagem favorece o reúso, uma vez que, definida a *FSML* para um determinado *framework*, ela pode ser instanciada quantas vezes for necessária. A técnica pode ser útil em cenários onde diversos sistemas de software têm arquiteturas similares. No

entanto, existem poucos exemplos reais de uso da técnica e o estilo de modelagem não trivial

ArchLint: a técnica é bastante indicada sobretudo nos casos onde a documentação do sistema é escassa ou inexistente. A estratégia de verificação não intrusiva também é um fator importante, uma vez que o impacto no processo de software tende a ser mínimo. No entanto, as premissas assumidas nas heurísticas de ArchLint pode não ser aplicável em todos os cenários. Além disso, ArchLint ainda depende de um modelo de alto nível definido por um arquiteto para que o mapeamento entre arquitetura e código seja feito.

Capítulo 3

Proposta de Solução

Este capítulo apresenta a proposta de solução para as questões discutidas nessa dissertação, especificamente, propõe-se uma nova versão para a linguagem de verificação de conformidade DCL (*Dependency Constraint Language*). Essa versão, chamada de DCL 2.0, apresenta novos conceitos, que têm por objetivo tornar DCL uma linguagem de propósito mais amplo, permitindo a detecção de novos tipos de violações de arquitetura, além de melhorar aspectos de DCL como especificação, documentação e visualização de arquiteturas. Este capítulo está organizado como descrito a seguir. A Seção 3.1 apresenta uma visão geral da solução proposta. A Seção 3.2 apresenta o novo modelo de especificação de DCL 2.0. A Seção 3.3 apresenta os novos tipos de violações detectados por DCL 2.0. A Seção 3.4 apresenta novos conceitos propostos em DCL 2.0. E, por fim, a Seção 3.5 apresenta as considerações finais.

3.1 A Linguagem DCL 2.0

Como o próprio nome indica, DCL 2.0 é uma evolução da linguagem DCL 1.0. A escolha da abordagem ocorreu devido aos resultados já apresentados pela técnica e também pela sua facilidade de uso. A principal motivação para a evolução de DCL se deu no sentido de dotá-la de funcionalidades consideradas importantes para que uma linguagem de definição de arquitetura seja utilizada de forma efetiva em processos de desenvolvimento de software, principalmente no que diz respeito ao suporte a tarefas de verificação de conformidade arquitetural. Embora DCL tenha sido a abordagem central para a proposta de solução dessa dissertação, os aspectos positivos e negativos das demais abordagens discutidas no Capítulo 2 também serviram de insumo para a identificação de novas funcionalidades, além de revelarem pontos de melhoria em funcionalidades já existentes em DCL 1.0. A Tabela 3.1 mostra as funcionalidades

impactadas pela proposta, a situação atual e a solução proposta em DCL 2.0.

.	Funcionalidade	DCL 1.0	DCL 2.0
F1	Especificação	Monolítica e *flat*.	Tornar a especificação modular e hierárquica.
F2	Verificação	Limitada a artefatos Java. Tipos de verificação restritos a controle de dependências.	Possibilitar a verificação de artefatos não Java e adicionar novos tipos de violações arquiteturais.
F3	Reusabilidade	Limitada.	Integrar a linguagem a ferramentas de gestão e configuração, como *IBM-RTC (Rational Team Concert)*[1] e *Maven*[2].
F4	Visualização de arquitetura	Não há.	Implementar uma API (AST) para facilitar a visualização de arquitetura em qualquer modelo de visualização.
F5	Grau de Cobertura Arquitetural	Não há	Implementar funcionalidade que permita visualizar a porção do código fonte que está sendo coberto pela especificação DCL 2.0.

Tabela 3.1: Comparação entre DCL 1.0 e DCL 2.0

Em seguida são apresentados em detalhes os pontos de melhoria de acordo com cada funcionalidade.

3.2 Especificação Hierárquica e Modular

3.2.1 Hierarquia

Grande parte dos métodos e processos de desenvolvimento de software atuais utilizam-se de modelos compostos por componentes de mais alto nível de abstração — por exemplo: sistemas, módulos — para representar arquiteturas de software. Essa situação ocorre sobretudo em estágio iniciais do desenvolvimento, onde não se tem total clareza de todos os requisitos arquiteturais. No entanto, à medida que essas arquiteturas vão sendo detalhadas e seus componentes vão sendo decompostos em partes menores — por exemplo: sub-módulos, pacotes, diretórios, classes e arquivos — observa-se o surgimento de um modelo hierárquico de componentes. Na prática, esses modelos acabam por refletir a estrutura interna do sistema e servem, em um primeiro momento, para comunicar a arquitetura do sistema. Algumas abordagens permitem esse tipo de especificação de arquitetura, argumentando que o modelo hierárquico permite trabalhar de forma fácil em vários níveis de abstração [Mitschke et al., 2013; Sangal et al., 2005].

Ainda que a especificação hierárquica seja uma importante característica em especificações de arquitetura, em DCL 1.0 esse tipo de especificação não é possível. DCL 1.0 segue um modelo de especificação no estilo *flat*, onde cada módulo é definido de forma independente e sem estabelecer uma relação de hierarquia com os demais

[1]https://jazz.net/products/rational-team-concert/
[2]https://maven.apache.org/index.html

módulos previstos na arquitetura. Em DCL 2.0, o modelo de especificação hierárquica foi introduzido na linguagem, habilitando arquitetos e usuários a especificar suas arquiteturas em vários níveis de abstração. Em DCL 2.0, os elementos que constituem o modelo hierárquico são chamados de componentes, por outro lado, os artefatos de código relativos a esses componentes são chamados de instâncias do componente. A Figura 3.1 mostra a estrutura lógica (a) e física (b) de um sistema fictício e em seguida são apresentadas as especificações DCL 1.0 (Listagem 3.1) e DCL 2.0 (Listagem 3.2) para tal sistema. Ao analisar a Listagem 3.1, percebe-se que, embora seja possível

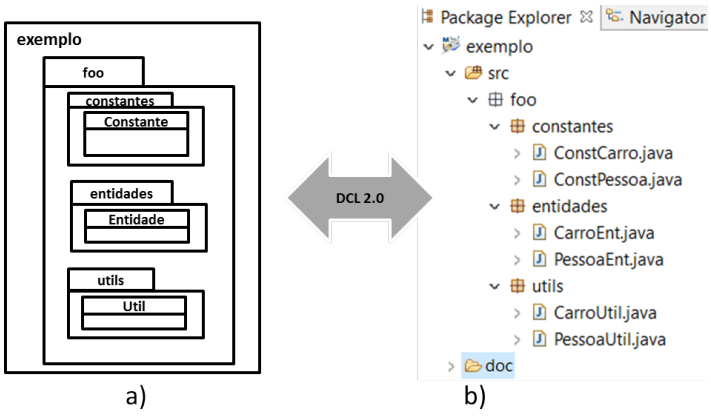

Figura 3.1: Modelagem de um sistema fictício - Modelo lógico e Modelo físico

realizar a especificação da arquitetura desse sistema fictício em DCL 1.0, a mesma não captura as características de aninhamento entre os módulos. Na Listagem 3.1, a definição de hierarquia dos componentes em DCL 1.0 foi alcançada colocando-se os módulos de mais alto nível primeiro e os de mais baixo nível depois, adicionando um carácter informal à hierarquia do sistema, uma vez que a definição de módulos em DCL 1.0 não precisaria necessariamente obedecer a essa ordem.

```
1  module java_lang : "java.lang.*"
2  module java_util : "java.util.*"
3  module java_text:  "java.text.*"
4
5  module foo:          "foo.**"
6  module entidades:    "foo.entidades.**"
7  module Entidade:     "foo.constantes.Ent[a-zA-Z0-9]"
8  module constantes:   "foo.constantes.**"
9  module Constante:    "foo.constantes.Const[a-zA-Z0-9]"
10 module utils:        "foo.utils.**"
11 module Util:         "foo.utils.[a-zA-Z0-9]Util"
12 Util       can-declare only java_lang
13 Constante  can-declare only java_lang, java_util, java_text, prod_comun
```

Listagem 3.1: Especificação DCL 1.0

```
1  architecture exemplo{
2      foo{
3          entidades{
4              Entidade{ matching: "Ent{?}";
5                  restrictions{
6                    can declare only platform.java.lang;
7                  }
8              }
9          }
10         constantes{
11             Constante{ matching: "Const{?}";
12                 restrictions{
13                   can declare only platform.java.lang;
14                 }
15             }
16         }
17         utils{
18             Util{ matching: "{?}Util";
19                 restrictions{
20                     can declare only platform.java.lang,
21                     requires entidades.Entidade;
22                 }
23             }
24         }
25     }
26     ignore "doc";
27 }
```

Listagem 3.2: Especificação DCL 2.0

Percebe-se que cada módulo é definido de forma isolada não havendo indicativo formal de que o mesmo possa fazer parte de outro módulo de mais alto nível e que também o mesmo módulo possa incluir módulos de níveis inferiores. Na Listagem 3.2, é possível perceber um alinhamento entre os modelos lógico e físico (Figura 3.1-a e 3.1-b) e a especificação DCL 2.0, uma vez que os três modelos: diagrama, código e especificação DCL 2.0 têm natureza hierárquica. Outra questão importante em relação a especificação de arquitetura é que DCL 2.0 exige mais formalidade ao se especificar arquiteturas. Como exemplo, pode-se observar a Linha 26 da Listagem 3.2, onde a palavra chave ***ignore*** é utilizada para informar que o componente *doc* não deve ser considerado como um artefato arquitetural; na prática artefatos relacionados a esse componente não serão verificados ou visualizados pelas ferramentas que leem a especificação DCL 2.0. Caso a palavra chave ***ignore*** não declarasse o componente *doc*, uma violação seria imediatamente gerada.

3.2.2 Modularidade

De maneira geral, modularidade em sistema de software é apresentada há anos como uma forma de tornar os sistemas mais compreensíveis e flexíveis impactando diretamente atividades de manutenção [Parnas, 1972]. No âmbito de arquitetura de software, modularidade também é considerada importante, como pode ser observado no modelo de documentação de arquitetura (4+1), onde visões como *Logical View* e *Implementation View* são dedicadas a capturar aspectos de modularização [Kruchten, 1995].

Ainda que sejam claros os benefícios trazidos por conceitos de modularização em sistemas de software, no contexto de conformidade arquitetural, sobretudo no que se refere à especificação de arquitetura, pouco se discute sobre modularidade. Isso pode ser constatado observando-se as abordagens relatadas no Capítulo 2, as quais, boa parte realizam a especificação de arquiteturas de forma monolítica.

DCL 1.0 está entre as abordagens cujo modelo de especificação é feito de forma monolítica. Em DCL 2.0, a linguagem passou a permitir a modularização de especificação de arquitetura por meio de um mecanismo de referência cruzada, onde um elemento é definido em um arquivo de especificação e referenciado em outro. Na Listagem 3.2 pode-se observar que os componentes (Linhas 6, 13 e 20) não foram definidos no arquivo corrente, mas apenas referenciados. A especificação para os componentes referenciados na Listagem 3.2 é apresentada na Listagem 3.3. É possível perceber que os componentes descritos na Listagem 3.3 referem-se a código de reúso. Por exemplo, o componente *lang*, refere-se ao pacote *java.lang* disponibilizado na plataforma Java e bastante utilizado em projetos dessa plataforma. Sendo assim, é possível perceber que uma das principais vantagens da modularização de especificações de arquitetura trazida por DCL 2.0 é tornar possível o reúso de componentes arquiteturais de forma similar ao reúso feito em nível de código. Ou seja, *frameworks*, APIs, bibliotecas podem ter seus componentes especificados em DCL 2.0 apenas uma vez e reusados em vários projetos de especificação de arquitetura.

```
1  architecture platform{
2      java{
3          lang{
4              reflect{}
5          }
6          util{}
7          text{}
8          io{}
9      }
10 }
```

Listagem 3.3: Especificação - Plataforma de reúso

3.3 Verificação de Novos Tipos de Violação

A extensão da linguagem DCL 1.0 para um modelo hierárquico possibilitou a verificação de novas violações arquiteturais, classificadas neste trabalho como *Violações de Estrutura*. *Violações de Estrutura* ocorrem quando algum elemento de código diverge do modelo de estrutura especificado pela hierarquia de componentes. O controle desse tipo de violação tem impacto direto no controle de complexidade do sistema, uma vez que novos componentes não podem ser criados sem que estejam formalmente declarados na arquitetura. Diferente de DCL 1.0, na nova versão da linguagem, as restrições que apontam *Violações de Estrutura* não são explicitamente declarados da forma *Module-Source* ***can-declare-only*** *Module-Target*, como pode ser visto na Linha 12 da Listagem 3.1. Em DCL 2.0, a própria especificação de hierarquia de componentes já define implicitamente restrições ao código fonte. Os tipos de *Violações de Estrutura* são detalhados a seguir:

3.3.1 Componente Desconhecido

Essa violação ocorre quando um artefato é encontrado no código fonte, mas o mesmo não se enquadra em nenhum dos componentes previstos no modelo da arquitetura planejada e, por consequência, não é especificado em DCL 2.0. A Figura 3.2 apresenta um exemplo de violação do tipo *Componente Desconhecido*. Na Figura 3.2-b, é possível observar a existência da classe *FooHelper* e do pacote *bar*, para os quais não existem componentes correspondentes no modelo de arquitetura apresentado na Figura 3.2-a. Considerando que a especificação de arquitetura apresentada na Listagem 3.2 reflete o modelo descrito na Figura 3.2-a, então duas violações do tipo *Componente Desconhecido* são reveladas.

3.3.2 Referência Desconhecida

Essa violação ocorre quando uma instância de um componente conhecido (artefato de código) referencia uma instância de um componente desconhecido. Em outras palavras, uma violação do tipo *Referência Desconhecida* será revelada sempre que um elemento de código estiver referenciando um segundo elemento que não está explicitamente especificado em DCL 2.0, seja como elemento interno ou externo ao sistema. A Figura 3.3 apresenta um exemplo de violação do tipo *Referência Desconhecida* onde a classe *PessoaEnt* referencia a classe *Remote* do pacote *java.rmi*. No entanto, se observarmos

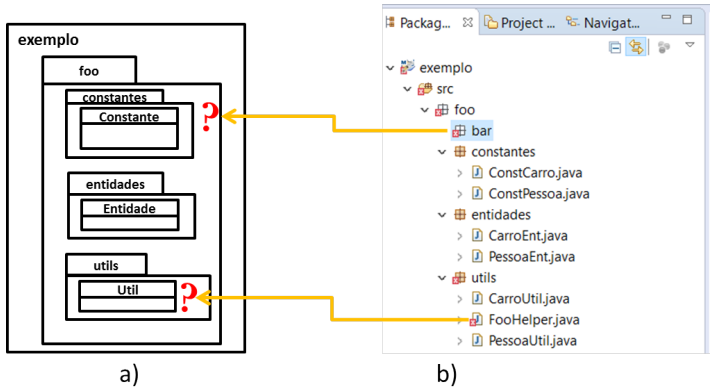

Figura 3.2: Violação do tipo *Componente Desconhecido*

os modelos apresentados nas Figuras 3.3-a e 3.3-c, nenhum deles apresenta um componente correspondente. Uma vez que esses modelos foram formalizados em DCL 2.0 por meio das Listagens 3.1 e 3.3, uma violação do tipo *Referência Desconhecida* é revelada.

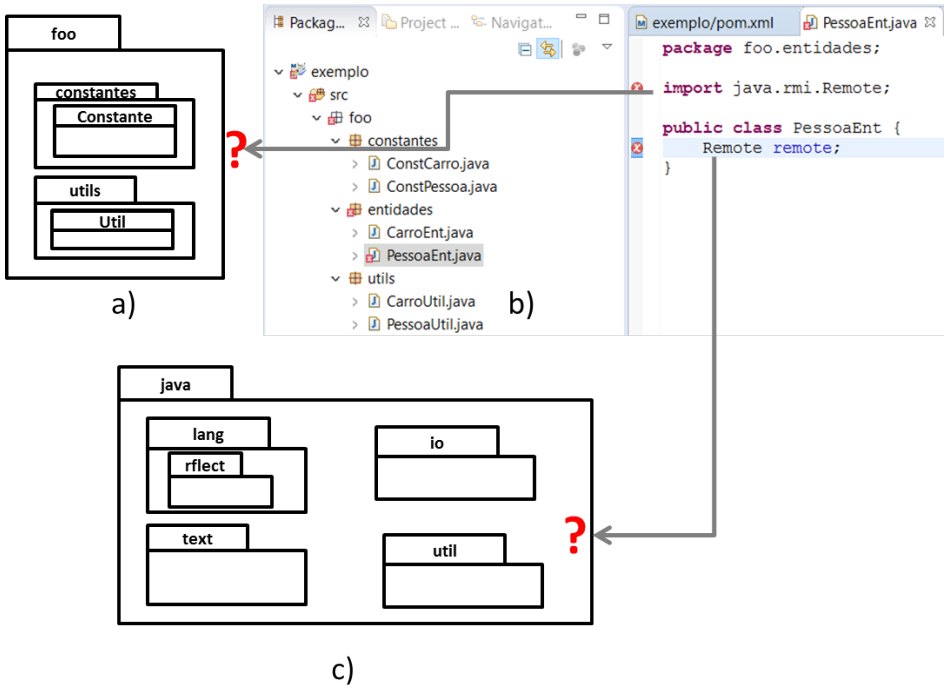

Figura 3.3: Violação do tipo *Referência Desconhecida*

3.3.3 Localização Incorreta

Essa violação ocorre quando um artefato de código é identificado como uma instância válida de um componente arquitetural, porém sua localização na arquitetura diverge

da localização onde o artefato foi implementado no código fonte. No exemplo da Figura 3.4, temos a ocorrência de uma violação do tipo *Localização Incorreta*. Nessa figura pode-se observar que a classe *PessoaEnt*, que é uma instância do componente *Entidade*, deveria estar localizada no componente *entidades* de acordo com o que prescreve a arquitetura do sistema, formalizada via Listagem 3.2. A seta laranja mostra o local onde o artefato se encontra no código e a seta verde indica onde a classe deveria estar localizada de acordo com o modelo de arquitetura.

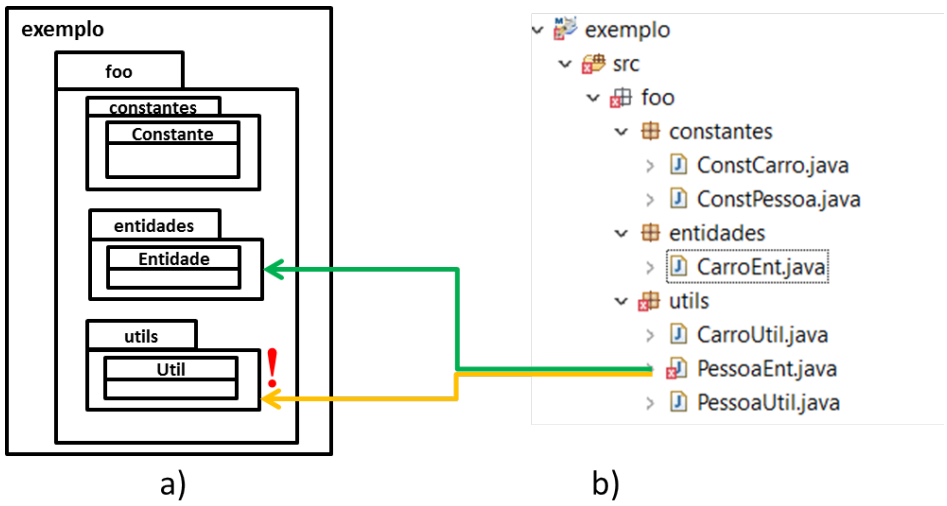

Figura 3.4: Violação do tipo *Localização Incorreta*

3.3.4 Ausência de Componente Dominante

As arquiteturas atuais precisam lidar com um tipo de dependência que vai além dos tipos já discutidos neste trabalho. Esse tipo de dependência ocorre entre os conceitos essenciais, que representam o modelo conceitual do sistema, e o código fonte. Em outras palavras, dependências dessa natureza ocorrem entre elementos relacionados a um mesmo conceito (i.e, requisito ou funcionalidade), porém em níveis de abstração ou contextos diferentes.

De maneira geral, é complexo o controle desse tipo de dependência, uma vez que os conceitos essenciais estão espalhados por vários artefatos no código fonte. Controlar a complexidade e o relacionamento entre os conceitos essenciais de um sistema é reconhecidamente uma tarefa difícil [Brooks, 1987]. Algumas técnicas de modelagem de sistema, tais como UML, classificam esse tipo de dependência como sendo *dependências do tipo abstração*[3]. No entanto, UML limita-se a documentar essa dependência e

[3]http://www.uml-diagrams.org/abstraction.html/

não fornece nenhuma proposta de validação.

Outro complicador relacionado a esse tipo de dependência é que, além de ocorrer entre elementos de níveis de abstração diferentes (modelo conceitual e código), ela também pode envolver elementos de natureza heterogênea, ou seja, artefatos de linguagens diferentes. Por exemplo, em aplicações Web é comum desenvolvedores utilizarem várias linguagens (e.g, Java, JavaScript, XML, CSS, HTML, etc.) para implementar um único requisito. Em termos arquiteturais, o desafio é garantir a rastreabilidade entre os elementos de forma a diminuir erros conceituais. Uma diferença crucial entre esse tipo de dependência e os demais tipos citados nesse trabalho é que, nesse tipo de dependência, não necessariamente existe uma relação física entre os elementos de código. Por essa razão, a maior parte das técnicas discutidas neste trabalho não detecta violações relacionadas a esse tipo de dependência.

DCL 2.0 propõe um novo tipo de restrição, representado pela palavra chave ***requires***, que tem como propósito controlar a rastreabilidade entre o modelo conceitual e elementos de código. Em DCL 2.0, a quebra dessa rastreabilidade é considerada uma violação do tipo *Ausência de Componente Dominante*. *Componente Dominante* é o componente que, dado um contexto, justifica a existência dos demais componentes, ou seja, domina os demais componentes. Em outras palavras, um componente dominado só pode existir se seu *Componente Dominante* existir. Na prática, a restrição ***requires*** cria uma dependência entre componente dominado e componente dominante, garantindo a rastreabilidade entre os dois elementos. A Listagem 3.4 mostra um exemplo de especificação DCL 2.0, onde observa-se o uso da restrição ***requires***.

```
1  architecture XYZ{
2      A{ }
3      B{
4          restrictions{
5            requires A;
6          }
7      }
8      C{
9          restrictions{
10           requires B;
11         }
12     }
13 }
```

Listagem 3.4: Uso da restrição ***requires***

Em DCL 2.0, a relação de dominância depende do contexto no qual os componentes estão inseridos, sendo assim, um componente dominante em um contexto pode ser um componente dominado em outro contexto. Em DCL 2.0, a relação de

dominância entre os componentes é transitiva, ou seja, se A domina B, e B domina C, logo, A domina C, e é justamente essa característica que garante a rastreabilidade entre elementos conceituais e elementos de implementação mais específicos. Diferente das demais violações propostas por DCL 2.0, violações do tipo *Ausência de Componente Dominante* necessitam que uma declaração explícita de restrição seja adicionada a uma especificação DCL 2.0, como pode ser visto na linha 21 da Listagem 3.2, onde a restrição é representada pela palavra chave ***requires***. A leitura dessa linha pode ser interpretada de forma abstrata da seguinte forma:

> *"Para toda instância do componente Util deve haver uma instância do componente Entidade relacionado ao mesmo conceito."*

No exemplo da Figura 3.5-b temos a ocorrência de uma violação do tipo *Ausência de Componente Dominante*. Nessa figura pode-se observar que a classe *PessoaUtil*, que é uma instância do componente *Util*, possui uma marcação de erro. Observando o pacote *entidades* percebe-se que a classe *PessoaEnt* foi removida do código fonte. Tal remoção gera uma violação do tipo *Ausência de Componente Dominante*, uma vez que a Linha 21 da Listagem 3.2 impõe uma restrição que é afetada diretamente pela remoção dessa classe. Em outras palavras, essa violação pode ser interpretada de forma concreta da seguinte forma:

> *"Para que exista a classe PessoaUtil deve existir a classe PessoaEnt".*

Na Figura 3.5-b pode-se observar que as partes sublinhadas dos nomes das classes representam os conceitos essenciais do sistema. Seguindo uma estratégia baseada em convenção de nomes, DCL 2.0 identifica essas partes essenciais e as utiliza como identificador para relacionar artefatos de um mesmo conceito. DCL 2.0 identifica os conceitos em dois passos. Primeiro, recupera-se o padrão de nome do componente por meio da palavra chave ***matching***. Segundo, retira-se todos os prefixos e sufixos restando apenas o radical, o qual passa a ser considerado um conceito essencial.

3.4 Novos conceitos

3.4.1 Grau de Cobertura Arquitetural

Conforme proposto por DCL 2.0, essa medida revela o quanto da arquitetura de um determinado sistema está formalizado na linguagem DCL 2.0. Na prática, essa métrica revela o quanto do código fonte pode ser verificado e visualizado. Em DCL 2.0, o grau de formalidade da arquitetura está diretamente ligado à palavra chave *ignore*, a

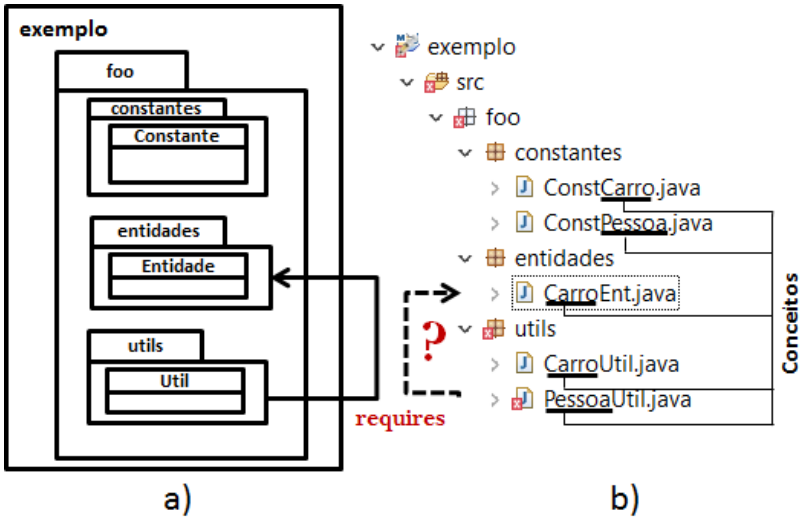

Figura 3.5: Violação do tipo *Ausência de Componente Dominante*

qual indica as partes do código que não devem ser consideradas durante a validação de arquitetura. Vale ressaltar que o julgamento do que é ou não parte da arquitetura do sistema é de responsabilidade dos arquitetos do sistema e depende de cada contexto. Em linhas gerais, quanto menos itens forem declarados como *ignore*, maior será o Grau de Cobertura Arquitetural da especificação. Em DCL 2.0, Grau de Cobertura Arquitetural é calculado sobre o número de artefatos ou sobre o número de linhas. O cálculo é feito da seguinte forma. (i) Somam-se todos os artefatos considerados instância de um componente, ou seja, aqueles que foram detectados por DCL 2.0, obtendo-se um número X; (ii) divide-se X pelo número total de artefatos do projeto. Por exemplo, um código que possui 10 artefatos de código (e.g, classes, XML, JavaScripts, etc.), mas somente 7 desses artefatos são considerados instância de componentes arquiteturais, diz-se que o *Grau de cobertura Arquitetural* é de 70%. Por padrão, arquivos binários são desconsiderados no cálculo.

3.4.2 Reusabilidade Arquitetural

Uma das principais mudanças observadas entre DCL 1.0 e 2.0 está na forma com que o projeto de especificação (arquitetura) e o projeto de código fonte (sistema) se relacionam. Enquanto em DCL 1.0, a especificação é parte integrante do sistema, em DCL 2.0 o projeto de arquitetura (especificação DCL 2.0) é totalmente desacoplado do código fonte do sistema alvo. Essa diferença possibilita que arquiteturas especificadas em DCL 2.0 possam ser reutilizadas. Além disso, essa mudança proporciona um maior controle da evolução das arquiteturas, uma vez que permite que as especificações possam ser geridas e distribuídas por ferramentas de gestão e configuração de software, de

forma semelhante ao que é feito com código fonte.

3.4.3 Visualização Arquitetural

O entendimento da arquitetura de um sistema é um importante fator para evitar a proliferação de violações arquiteturais [Murphy et al., 1995]. Nesse sentido, qualquer aspecto que facilite a visualização da arquitetura contribui indiretamente para conformidade arquitetural do sistema.

A decisão de tornar o modelo de especificação DCL 2.0 em um modelo hierárquico permitiu a implementação de uma API, a qual torna fácil diferentes visualizações de arquiteturas especificadas em DCL 2.0. Além disso, a API de DCL 2.0 fornece completo acesso a elementos básicos da gramática da linguagem, tais como, interpretadores, reconhecedores sintáticos, validadores, tornando fácil as tarefas de edição e visualização de arquiteturas.

Em seguida são apresentadas três visualizações para uma mesma arquitetura de sistema, obtidas a partir da API de DCL 2.0 e apresentadas na forma de *views* da plataforma *Eclipse*. Na Figura 3.6, é apresentada a visualização da arquitetura em forma textual; na Figura 3.7 observa-se a visualização da árvore de componentes arquiteturais; e na Figura 3.8 mostra-se os artefatos do sistema (classes Java) e o nome do respectivo componente arquitetural entre «». Como exemplo, pode-se observar as classes *ConstCarro* e *ConstPessoa* que são instâncias do componente *Constantes*. Nessa visualização também pode-se observar que novos ícones foram aplicados aos componentes *constantes, entidades* e *utils*, os quais deixaram de ser representados como pacotes genéricos da linguagem Java para terem uma visualização mais representativa da responsabilidade do componente.

Vale ressaltar que, assim como a especificação da hierarquia de componentes, a especificação de ícones customizados para cada componente é disponibilizada em DCL 2.0, sendo totalmente desacoplada das ferramentas de visualização. Essa funcionalidade permite que no momento da especificação de determinado componente possa-se definir também uma imagem que melhor defina tal componente em termos visuais. De maneira semelhante, a palavra chave ***description*** pode ser utilizada para especificar uma descrição textual do componente, como pode ser observado na Linha 4 da Figura 3.6.

3.4.4 Correspondências

Em DCL 1.0 a especificação de correspondência entre arquitetura e código é feita por meio de uma estratégia que combina convenção de nomes e expressão regular. A Lista-

Figura 3.6: Visualização textual da arquitetura

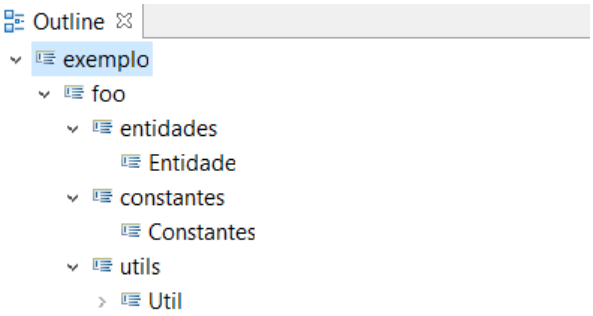

Figura 3.7: Visualização da árvore de componentes

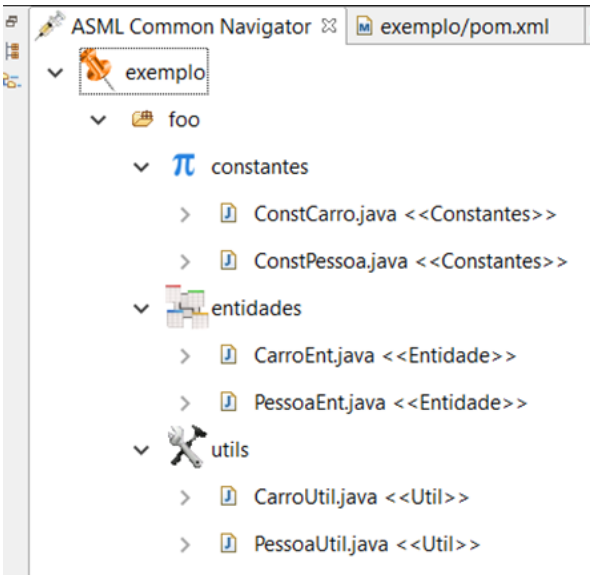

Figura 3.8: Visualização da arquitetura e código

gem 3.5 apresenta um exemplo de especificação de correspondência entre o componente Entidade e os artefatos de código feita em DCL 1.0. Na prática, a declaração pode ser entendida da seguinte forma: Toda classe Java contida no pacote *foo.constantes* cujo nome começa com *Ent*, é uma instância do componente *Entidade*.

```
1 module Entidade:   "foo.constantes.Ent[a-zA-Z0-9]"
```

Listagem 3.5: Correspondência DCL 1.0

Em DCL 2.0, a estratégia baseada em convenção de nomes também foi utilizada, porém algumas modificações foram feitas em função do modelo hierárquico de especificação. Em DCL 2.0 informações de localização do componente não necessitam ser informado como em DCL 1.0, uma vez que a própria estrutura arquitetural já informa a localização do componente. A Listagem 3.5 apresenta um exemplo de especificação de correspondência em DCL 2.0.

```
1 architecture exemplo{
2     foo{
3         entidades{
4             Entidade{ matching: "Ent{?}";
5             }
6         }
7     }
8 }
```

Listagem 3.6: Correspondência DCL 2.0

Além da estratégia baseada em convenção de nomes, DCL 2.0 permite outros dois tipos de estratégias para captura de instâncias de componentes: Por extensão de arquivo e por anotação. A Listagem 3.7 apresenta uma especificação de exemplo, onde ilustra a utilização das três estratégias de correspondência oferecidas por DCL 2.0. Na Linha 4 pode-se observar a palavra chave **matching** utilizando-se da estratégia de correspondência por anotação para capturar as instâncias do componente *Model*. Na Linha 5 observa-se o uso da estratégia de correspondência por extensão do arquivo para capturar as instâncias do componente *View*. Na Linha 6, observa-se a utilização da estratégia de convenção de nomes para capturar as instâncias do componente *Controller*. Vale ressaltar que a captura de artefatos em DCL 1.0 limita-se a classes Java, porém em DCL 2.0 a captura pode ser aplicada a qualquer tipo de artefato.

```
1 architecture exemplo{
2     foo{
3         mvc{
4             Model{ matching: "annotation=Entity";}
5             View{ matching: "extension=xhtml";}
6             Controller{ matching: "{?}Ctr";}
7         }
8     }
9 }
```

Listagem 3.7: Novas estratégias de correspondência

3.5 Considerações Finais

Esse capítulo apresentou a proposta de solução para as questões colocadas nessa dissertação de mestrado. Na Seção 3.1 foi apresentada uma visão geral da linguagem DCL 2.0, bem como as principais mudanças em relação a versão 1.0. Na Seção 3.2 foram apresentados os conceitos de especificação *Hierárquica e Modular* e os impactos esperados desses conceitos sobre a especificação de arquitetura, sobretudo em relação ao reúso e evolução das especificações arquiteturais. Na Seção 3.3 foram apresentados novos tipos de restrições trazidos por DCL 2.0, assim como alguns exemplos de uso dessas restrições. A Seção 3.4.1 apresentou o conceito de *Grau de cobertura arquitetural*, o qual é utilizado para informar o quanto do código do sistema alvo está sendo mapeado pela especificação de arquitetura. As características de *Reusabilidade* foram descritas na Seção 3.4.2, onde foi apresentada a forma desacoplada com que DCL 2.0 se relaciona com o sistema alvo, o que também favorece o reúso e evolução das especificações arquiteturais. A Seção 3.4.3 mostrou o utilização da API de DCL 2.0 em três modelos de visualização de arquitetural diferentes, salientando a importância das visualizações arquiteturais para a compreensão do sistema.

Capítulo 4

Ferramenta

Com o intuito de avaliar a aplicabilidade de DCL 2.0, este trabalho apresenta uma ferramenta que implementa a maior parte dos conceitos apresentados pela abordagem. Este capítulo está organizado como descrito a seguir. A Seção 4.1 apresenta uma visão geral da ferramenta proposta, bem como os principais componentes utilizados em sua construção. A Seção 4.2 apresenta o modelo conceitual da ferramenta, suas principais entidades e os relacionamentos entre elas. A Seção 4.3 apresenta as principais funcionalidades implementadas na ferramenta DCL 2.0. E, por fim, a Seção 4.4 apresenta as considerações finais.

4.1 Visão Geral

Os dois principais componentes utilizados na construção dessa ferramenta foram a plataforma *Eclipse* e *Xtext*[1]. Xtext é um *framework* para desenvolvimento de linguagens de domínio específico. O *framework* possui editores que facilitam a definição de gramáticas para vários tipos de necessidades. Além disso, *Xtext* possui aceleradores que permitem a rápida construção de elementos típicos de uma linguagem de programação como, analisador léxico e sintático, compilador, validadores, etc. No Apêndice B são apresentados os detalhes da gramática de DCL 2.0.

A ferramenta é disponibilizada na forma de um conjunto de *plug-ins* para a plataforma *Eclipse*. Para se utilizar a ferramenta é necessário criar um projeto específico na plataforma Eclipse o qual conterá a especificação DCL 2.0 da arquitetura. Uma vez que o projeto de especificação esteja criado, o mesmo deve ser exportado para o formato de *JAR (Java ARchive)*, de forma que a especificação possa ser aplicada a

[1]https://eclipse.org/Xtext/

um sistema alvo. Na prática, o arquivo JAR contendo a especificação DCL 2.0 é adicionado como dependência do projeto para o qual se deseja verificar a conformidade arquitetural. A ferramenta de verificação de DCL 2.0 identifica a especificação de arquitetura entre as dependências do projeto e então a utiliza para fazer as verificações de conformidade arquitetural do mesmo. Uma integração com o *framework Maven* foi implementada na ferramenta para facilitar o processo de distribuição da especificação. Com a integração, os arquitetos disponibilizam os projetos de especificação em repositórios remotos, os quais são acessados automaticamente pelo *framework Maven* para atualizar a especificação DCL 2.0 na máquina de cada desenvolvedor.

4.2 Arquitetura

A Figura 4.1 apresenta o modelo conceitual da ferramenta contendo suas principais entidades, as quais são em seguida descritas.

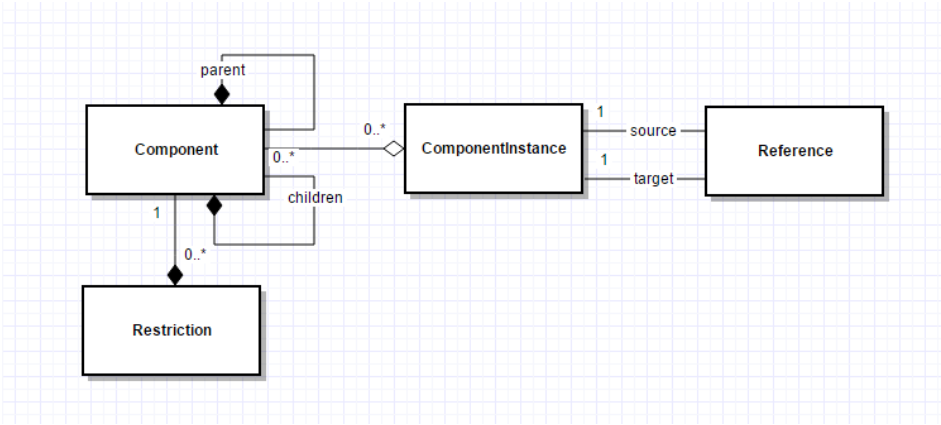

Figura 4.1: Modelo conceitual da ferramenta DCL 2.0

1 - Component - Principal entidade do modelo de DCL 2.0, a qual é utilizada para representar os elementos conceituais de um sistema. Essa entidade possui dois autorelacionamentos (*parent* e *children*) que permitem a navegação na árvore de componentes da arquitetura.

2 - ComponentInstance - Entidade que representa a parte do código fonte relativa a um determinado componente.

3 - Reference - Entidade que representa a relação entre duas instâncias de componentes.

4 - Restriction - Entidade que representa as regras aplicadas a cada instância de um determinado componente.

4.3 Funcionalidades

4.3.1 Edição de Arquitetura

A ferramenta apresenta um conjunto de funcionalidades que possibilita a edição de especificações de arquiteturas na linguagem DCL 2.0. Por exemplo, a ferramenta disponibiliza um editor de texto adaptado à sintaxe da linguagem. Esse editor dispõe de funcionalidades como referência cruzada, recursos de auto completar e validação automática de erros de sintaxe.

4.3.2 Verificação de Arquitetura

Em relação a verificação de conformidade arquitetural, a ferramenta oferece um *plug-in* específico para realização dessa tarefa. A verificação pode ser habilitada em cada projeto Eclipse por meio do menu *configuration*. Uma vez habilitada no projeto Eclipse, a verificação de arquitetura pode ocorrer de três formas diferentes: a) *Live feedback* - a validação ocorre toda vez que um determinado artefato é alterado; b) *Build feedback* - a validação ocorre durante o processo de compilação do projeto; e c) *Off-line feedback* - a validação ocorre por demanda, ou seja, no momento que o desenvolvedor deseja fazer, o mesmo aciona um comando específico no IDE Eclipse para dar início ao processo de validação. A escolha do tipo de validação a ser utilizada pode ser configurada na página de preferências da ferramenta no IDE Eclipse. Na Figura 4.2 pode se observar uma violação na classe *PessoaEnt*. A violação é apontada na linha de código do artefato onde ocorre a violação. Uma mensagem textual também é apresentada para facilitar o entendimento dos desenvolvedores a cerca do tipo de violação. Vale ressaltar que as mensagens de restrições também podem ser especificadas na linguagem DCL 2.0.

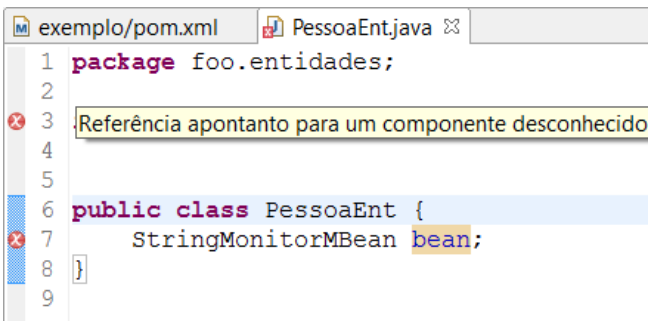

Figura 4.2: Violação arquitetural

4.3.3 Visualização de Arquitetura

Todas as visualizações discutidas na Seção 3.4.3 são implementadas pela ferramenta DCL 2.0. Além dessas visualizações, a ferramenta também utiliza as *views Console* e *Problems* do Eclipse para apresentar informações sobre violações arquiteturais. A ferramenta também disponibiliza uma visualização que apresenta o *Grau de cobertura arquitetural* de cada projeto Eclipse, a qual pode ser acessada via menu propriedades do projeto Eclipse.

4.4 Considerações Finais

Neste Capítulo foi apresentada a ferramenta que implementa os principais conceitos da linguagem DCL 2.0. Na Seção 4.1 foi apresentada uma visão geral da ferramenta proposta e os principais componentes utilizados em sua construção. Na Seção 4.2, o modelo conceitual da ferramenta e suas principais entidades foram apresentados. Na Seção 4.3, as principais funcionalidades foram apresentadas.

Capítulo 5

Avaliação da Solução

Com a intenção de avaliar a linguagem proposta neste trabalho, este capítulo apresenta um estudo de caso, fruto da parceria entre ASERG-UFMG (*Applied Software Engineering Research Group*) e a PRODEMGE (Empresa de Tecnologia da Informação do Estado de Minas Gerais), no qual um dos sistemas desenvolvidos pela empresa passou por um processo de conformidade arquitetural utilizando a linguagem DCL 2.0.

O estudo de caso concentrou-se em dois pontos principais: em primeiro lugar, entender o fenômeno de erosão arquitetural no contexto da empresa, e em segundo lugar, avaliar a linguagem DCL 2.0. As questões e os objetivos derivados desses dois pontos são apresentados na Tabelas 5.1 e 5.2.

Erosão arquitetural		
.	**Questão**	**Objetivo**
Q1.1	Por que as violações são geradas?	Entender em que medida fatores como pressões de projeto, requisitos, falta de comunicação, inexperiência da equipe e a falta definição de arquitetura podem causar violações arquiteturais. Em mais detalhes responder: Quais as causas das violações arquiteturais? Qual o perfil dos desenvolvedores que as provocaram? Quando ocorrem, na elaboração, na construção, na manutenção?
Q1.2	Como são tratadas?	Entender como a equipe lida com as violações, ou seja, as violações são percebidas? São removidas? Quais as formas usadas na remoção das violações? Utiliza-se alguma ferramenta? Tem algum planejamento ou processo?

Tabela 5.1: Questões relacionadas a erosão arquitetural

Este capítulo está organizado da seguinte forma: Seção 5.1, apresenta a metodologia usada no estudo de caso. Na Seção 5.2, a execução do estudo é apresentada em detalhes. Seção 5.3, mostra os dados obtidos no estudo. Na Seção 5.4 é feita uma análise sobre os resultados obtidos. Por fim, na Seção 5.5 são discutidas as principais percepções obtidas ao longo do estudo de caso.

Avaliar a linguagem DCL 2.0 de forma geral		
.	**Questão**	**Objetivo**
Q2.1	DCL 2.0 pode ser usada para evitar violações arquiteturais?	Trazer um entendimento em relação às dificuldades, os limites e os entraves encontradas ao se utilizar DCL 2.0 como ferramenta de combate a violação arquitetural em processos de desenvolvimento de sistemas de software reais.
Q2.2	Os novos conceitos propostos por DCL 2.0 podem melhorar o processo de verificação arquitetural?	Verificar se aspectos como *Modelagem hierárquica e modular, novos tipos de restrições, referência cruzada entre módulos, reusabilidade e gestão e configuração de arquitetura* podem melhorar o processo de verificação arquitetural.
Q2.3	A nova versão da ferramenta pode ser usada em processos reais de verificação de arquitetura?	Avaliar se aspectos como funcionalidade, robustez, usabilidade e desempenho podem impactar na adoção da abordagem sugerida.

Tabela 5.2: Questões relacionadas à DCL 2.0

5.1 Metodologia

Esta seção foi dividida em três partes: na Seção 5.1.1 são apresentados os critérios para escolha do sistema alvo. Em seguida, as métricas a serem utilizadas no estudo de caso são apresentadas na Seção 5.1.2. Por fim, na Seção 5.1.3 é mostrado em detalhes o processo de conformidade arquitetural proposto especificamente para o estudo de caso.

5.1.1 Escolha do Sistema Alvo

Alguns trabalhos têm relatado a necessidade de se avaliar novas abordagens em sistemas reais antes de oferecê-las à Indústria. Knodel et al. [2008a] questionam abordagens baseadas apenas em protótipos e validadas apenas em sistemas exemplo. Tonella et al. [2007] enfatiza o papel da avaliação empírica na apresentação de uma determinada abordagem. Portanto, a fim de avaliar a real aplicabilidade de DCL 2.0, alguns critérios foram pré-estabelecidos para escolha do sistema alvo, de forma que o cenário em torno do sistema escolhido representasse um contexto o mais apropriado para o tema conformidade arquitetural. São eles:

1. **Tamanho do Sistema - Médio a Grande**

   As causas de violação arquitetural envolvem vários fatores, como pressões relacionadas a prazo do projeto, requisitos conflitantes, falta de comunicação, falta de experiência dos desenvolvedores e a falta de uma explícita definição de arquitetura [Terra et al., 2012c]. Em sistemas maiores, a probabilidade de esses fatores ocorrerem é grande, criando um cenário propenso a proliferação de violações arquiteturais.

2. **Linguagem do Sistema - Java**

   Embora os principais conceitos de DCL possam ser aplicados a qualquer sistema orientado por objeto, a versão atual da ferramenta é mais adequada à plataforma Java. Muito embora suporte a artefatos "não Java" tenha sido adicionado em DCL 2.0, as restrições do tipo DCL 1.0 possuem implementação apenas para a plataforma Java. Por essa razão, definiu-se que o sistema alvo deveria ser implementado em Java.

3. **Nível de Atividade - Em Fase de Construção**

   Foram descartados sistemas muito estáveis e com pouca atividade de desenvolvimento, uma vez que é menos provável violações em sistemas sem desenvolvimento de novas funcionalidades ou sem manutenção de funcionalidades existentes.

4. **Número e Perfil de Desenvolvedores - Equipes médias**

   Sistemas muito personificados ou com equipes muito pequenas foram retirados da lista de escolha. Em sistemas com essas características é menor a probabilidade de haver violação arquitetural, pois quem define a arquitetura é também quem implementa e, em tese, tem consciência dos padrões. Em geral, são sistemas que trazem menor risco a para empresa e não exigem uma rigidez arquitetural muito grande. Em relação aos perfis dos desenvolvedores, foi dada prioridade a sistemas cujas equipes eram compostas por profissionais de várias faixas de experiência: júnior, pleno, sênior a fim de refletir cenários usuais, pois, dependendo do expertise, os desenvolvedores se comportam de maneira diferente, causando diferentes impactos na arquitetura. A classificação dos perfis seguiu critérios da empresa, embora tenham sido realizadas também reuniões com a gestão do projeto para garantir um ajuste fino dos perfis.

Após algumas reuniões com os arquitetos da empresa e análise de algumas dezenas de sistemas, o sistema alvo definido foi o *SSC-ADMIN*, que é um sistema de um ecossistema conhecido como SSC (Sistema de Segurança Corporativo), uma solução para gestão de identidade digital do Estado de Minas Gerais. O SSC-ADMIM é o módulo de administração da solução. Ele é desenvolvido na linguagem Java e suas principais funcionalidades são Gestão de Unidades (órgãos), Gestão de Sistemas, Gestão de Usuários, Gestão de Recursos, Gestão de Perfil e Auditoria. Seção 5.2.2 mostra algumas métricas referentes ao SSC-ADMIM e Seção 5.2.3 apresenta sua arquitetura.

5.1.2 Métricas

A fim de responder as questões definidas para o trabalho, foi estabelecido um conjunto de métricas, as quais deveriam ser registradas durante o período do estudo de caso. As métricas foram classificadas em quatro grupos: Métricas relacionadas ao sistema, métricas relacionadas a equipe, métricas relacionadas ao processo e métricas relacionadas às violações. As Tabelas 5.3, 5.4, 5.5 e 5.6 mostram o nome e a descrição de cada métrica.

Métricas relacionadas ao sistema		
.	**Nome**	**Descrição**
1	Número de classes	Número de classes existentes em todos os módulos analisados.
2	Número de arquivos	Número de arquivos existentes em todos os módulos analisados.
3	Tipos de artefatos	Número de tipos de arquivos classificados pela sua extensão. Por exemplo XML e Java.
4	Linhas de código	Número de linhas de código Java, descontados espaços e comentários.
5	Número de funcionalidades	Número de funcionalidades do sistema. Calculado por meio da regra: Soma de todas as entidades (VO) multiplicado por 5 (CRUD + Pesquisa) mais a soma de todos comandos (CmdVO). Ex.: PessoaVO = 5, RegistrarAcessoCmdVO = 1.
6	Número de mudanças	Número de mudanças (*commits*) feitas no código ao longo do projeto. Por exemplo adição, alteração, remoção, movimentação e renomeação de artefatos.
7	Data das entregas (*commits*)	Data em que o código do desenvolvedor foi adicionado ao ramo principal de desenvolvimento.

Tabela 5.3: Métricas relacionadas ao sistema

Métricas relacionadas à equipe		
.	**Nome**	**Descrição**
1	Número de desenvolvedores	Número de desenvolvedores que atuaram no projeto.
2	Grau de experiência	Grau de experiência de cada desenvolvedor (júnior, pleno, sênior).
3	Tempo de projeto	Tempo de atuação de cada desenvolvedor em cada projeto.

Tabela 5.4: Métricas relacionadas à equipe

Após a definição das métricas, foi criado um modelo de dados para representar as informações a serem extraídas do estudo de caso e foi escolhida uma base de dados relacional para o armazenamento das informações para possibilitar consultas futuras. Para armazenamento da base de dados foi utilizado o banco de dados *Postgres*.[1]

5.1.3 Definição do Processo

O processo de remoção de violações arquiteturais sugerido por Tvedt et al. [2002] serviu como base para a definição do processo utilizado no estudo de caso. Como pode ser visto no estudo feito por Rosik et al. [2011], normalmente processos de remoção de violações arquiteturais possuem os seguintes passos:

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

Métricas relacionadas ao processo		
.	**Nome**	**Descrição**
1	Número de módulos	Número de módulos especificados na linguagem DCL 2.0 durante o processo, ou seja, o número de projetos de especificação de arquitetura.
2	Número de componentes	Número de componentes especificados na linguagem DCL 2.0 durante o processo.
3	Número de referências	Número de referências entre módulos utilizadas durante o processo de especificação.
4	Número de evoluções	Número de versões das especificações DCL 2.0 criadas durante o processo.
5	Tempo de verificação	Tempo de processamento da verificação da arquitetura.
6	Número de violações removidas	Número de violações removidas durante o processo de conformidade arquitetural.
7	Número de horas	Número de horas gastas durante o processo de conformidade arquitetural.
8	Número de mudanças (*commits*)	Número de mudanças feitas em função das violações.
9	Grau de cobertura arquitetural	Grau de cobertura arquitetural do sistema após o processo de formalização.

Tabela 5.5: Métricas relacionadas ao processo

Métricas relacionadas às violações		
.	**Nome**	**Descrição**
1	Tipo de violação	Tipo de restrição violada. Ex.: must implement, declare only.
2	Tipo de dependência	De estrutura ou de relacionamento.
3	Causa	Motivo que levou à violação. Ex.: Falta de conhecimento, pressão externa, requisitos conflitantes.
4	Autor	Desenvolvedor que inclui a violação no ramo principal do projeto.
5	Complexidade	Dificuldade no processo de remoção (BAIXA, MÉDIA, ALTA) .
6	Data	Data da violação no ramo principal.
7	Localização	Local no código fonte onde a violação ocorreu.

Tabela 5.6: Métricas relacionadas às violações

1. Formalização da Arquitetura.

2. Recuperação do Código fonte.

3. Verificação da Arquitetura.

4. Análise de Inconsistências.

5. Correção das Violações (Repetir os passos de 3 a 5).

Para o contexto do estudo de caso, algumas adaptações no processo foram necessárias para torná-lo mais iterativo e incremental e, assim, evitar maiores impactos no desenvolvimento diário do sistema. A Figura 5.1 mostra a comparação entre o processo original e o processo adaptado para o estudo. As principais alterações no processo foram: 1) Criação de três novas etapas *(Preparação da equipe, Geração de versão e Acompanhamento e evolução)*; 2) Mudança na ordem de duas etapas *(Preparação do código fonte e Formalização da arquitetura)* e 3) Junção de duas etapas *(Verificação de arquitetura e Análise de violações)*.

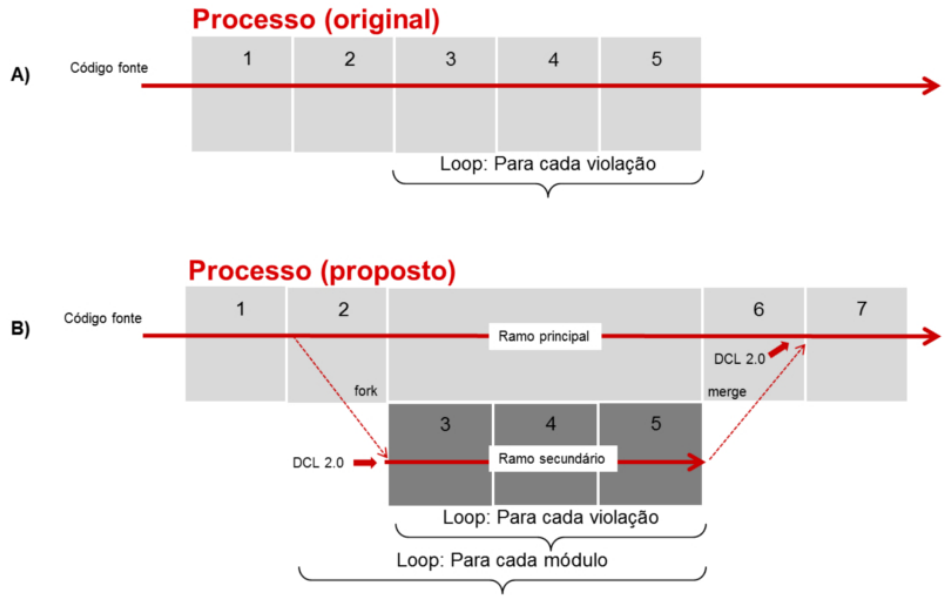

Figura 5.1: Processo proposto para o estudo de caso

Após as adaptações, o processo proposto para o estudo de caso inclui as seguintes etapas: *1. Preparação da equipe, 2. Preparação do código fonte, 3. Formalização da arquitetura, 4. Verificação e Análise, 5. Remoção das violações, 6. Geração de nova versão, 7. Acompanhamento e evolução. Na sequência segue o planejado para cada etapa do processo.*

1. **Preparação da Equipe**

   Com o intuito de facilitar a adoção de práticas de conformidade arquitetural por parte da equipe do projeto, no início dos trabalhos foram planejadas algumas reuniões com a equipe. As reuniões previam o alinhamento de conceitos de DCL 2.0 e explicações sobre o funcionamento da ferramenta. No planejamento definiu-se que a equipe do projeto deveria ser separada em dois grupos. O Grupo 1, composto por arquitetos, os quais seriam responsáveis por: formalizar a arquitetura no padrão DCL 2.0 e refatorar o código para remover as violações na arquitetura. Os demais desenvolvedores seriam o Grupo 2, responsável por continuar com o desenvolvimento normal do sistema.

2. **Recuperação do Código Fonte**

   Para essa fase, ficou planejada a recuperação da versão atual do código fonte do sistema, bem como as versões anteriores do mesmo. As informações deveriam ser

formatadas no modelo de dados mencionado anteriormente e armazenadas no banco de dados, de forma a possibilitar consultas futuras.

Como pode ser visto no processo original, Figura 5.1-A, a formalização da arquitetura (Fase 1) aparece antes da etapa de recuperação da arquitetura implementada (Fase 2). Essa ordem é comum na maioria dos processos de remoção de violação arquitetural, como pode ser observado no trabalho apresentado por [Rosik et al., 2011]. No entanto, para este estudo de caso, como mostra a Figura 5.1-B, a obtenção da arquitetura implementada ou código fonte (Fase 2) foi definida para ocorrer antes da formalização da arquitetura. A motivação para a alteração de ordem das etapas se deve ao fato de que o código fonte também poderia ser uma fonte de documentação para a fase de formalização, ou seja, o código fonte poderia ser consultado no momento de especificação da arquitetura em DCL 2.0. Para amenizar o impacto no processo de desenvolvimento da equipe, definiu-se a criação de um ramo de código secundário no qual seriam realizadas as correções de arquitetura, sendo assim, o Grupo 1 trabalharia no ramo secundário e Grupo 2 continuaria o desenvolvimento normal do sistema no ramo principal.

3. **Formalização da Arquitetura**

Nessa etapa do processo, após a instalação da ferramenta DCL 2.0, os arquitetos do Grupo 1 deveriam utilizar o editor da linguagem DCL para criar os projetos de definição da arquitetura do sistema. Uma vez formalizada a arquitetura, o editor de DCL deveria ser utilizado para mapear os componentes arquiteturais para componentes de código. Para implementar o mapeamento, os arquitetos deveriam se basear no modelo de estereótipos usados na arquitetura da empresa, que é feito basicamente via convenção de nomes, ou seja, um estereótipo pode ser identificado por um sufixo ou prefixo adicionados ao nome do artefato. Por exemplo, uma classe cujo arquivo tem o nome *PessoaCtr* (sufixo=Ctr), significa que essa classe deve ser classificada como um componente de «controle».

4. **Verificação da Arquitetura e Análise**

Após a etapa de *formalização da arquitetura*, o módulo de verificação da ferramenta DCL deveria ser habilitado nas máquinas dos arquitetos do Grupo 1. Os arquitetos deveriam utilizar novamente o editor de DCL 2.0 para implementar as restrições arquiteturais em linguagem DCL. Após a implementação das restrições, deveria-se confrontar a arquitetura planejada e código implementado para que as violações

arquiteturais fossem reveladas. Devido ao tamanho do sistema e também devido a interdependência entre os módulos, foi definido que a verificação, bem como as demais etapas do processo, deveriam ser executadas de forma incremental. A estratégia para validar a arquitetura deveria seguir o seguinte critério: *"Habilitar a verificação do sistema módulo a módulo, e em ordem do módulo menos dependente para o módulo mais dependente"*. Com a lista de violações, os arquitetos do Grupo 1 deveriam avaliar cada violação e planejar mudanças no código, mudanças na arquitetura ou em ambos.

5. **Remoção das Violações**

   As violações deveriam ser corrigidas, documentadas e entregues no repositório remoto. O Grupo 1 foi instruído a repetir para cada módulo individualmente, os Passos 2, 3, 4 e 5 até que todas as violações fossem corrigidas em todos os módulos.

6. **Geração de Nova Versão**

   Após todas as remoções de violação terem sido feitas no ramo secundário, todas as alterações deveriam ser levadas para o ramo principal e uma nova versão do sistema gerada. No entanto, as violações ocorridas no ramo principal durante o período de refatoração no ramo secundário, deveriam ser corrigidas antes da geração da nova versão. Durante o período de geração da nova versão nenhuma nova funcionalidade deveria ser criada e nenhuma manutenção poderia ser feita no ramo principal. Portanto, essa etapa não deveria ser longa. Planejamos um período de uma semana para esta etapa. Após a junção dos dois ramos, a ferramenta DCL deveria ser instalada nas máquinas do segundo grupo, a fim de coibir violações.

7. **Acompanhamento e Evolução**

Após a geração da nova versão do sistema, o planejamento previa a realização de uma reunião com toda a equipe do projeto para apresentar a arquitetura resultante e os resultados obtidos no estudo de caso. Nessa reunião deveria ser apresentada a definição da arquitetura do sistema a partir da funcionalidade de visualização de arquitetura da ferramenta DCL 2.0. Os arquitetos deveriam acompanhar a arquitetura a fim de identificar possíveis evoluções sempre que um novo requisito arquitetural aparecesse. Caso alguma alteração fosse necessária na arquitetura, uma nova versão da definição da arquitetura na linguagem DCL 2.0 deveria ser criada e publicada no repositório. Os arquitetos deveriam também auxiliar os desenvolvedores no entendimento e interpretação das mensagens exibidas pela ferramenta.

5.2 Execução do Estudo de Caso

5.2.1 Preparação da Equipe

Conforme planejamento, a equipe foi dividida em dois grupos: o Grupo 1, formado por dois arquitetos e o Grupo 2, formado por oito desenvolvedores. Os principais tópicos relacionados à conformidade arquitetural foram apresentados à equipe e um pequeno exemplo foi utilizado para demostrar o funcionamento da ferramenta. As reuniões tiveram início na primeira quinzena de Julho de 2015.

5.2.2 Preparação do Código Fonte

Para o estudo, as ferramentas *IBM-RTC*[2] *(Rational Team Concert)* e *Maven*[3] foram adotadas para controle de versão e gestão de dependência, respectivamente. As ferramentas foram utilizadas tanto para controle dos projetos de código (e.g., .java, .js e .xml), quanto para os projetos de especificação de arquitetura (e.g,.asml). A escolha se deu devido ao fato de o *RTC* e *Maven* serem as ferramentas oficiais da empresa para controle de versão e gestão de dependências. Com o código disponível, foi utilizada a API do *RTC* para coleta de algumas informações iniciais relacionadas ao sistema.

Após alguns testes iniciais, verificou-se a necessidade de alteração das políticas de entrega no repositório do *RTC* em função das regras de gestão e configuração da empresa, as quais todos os códigos são submetidos antes de serem entregues no repositório do *RTC*. Em especial, uma dessas regras afetaria diretamente o processo de remoção de violações. Segue a descrição da regra em questão: *"Nenhum código pode ser entregue no repositório contendo erros."*, ou seja, códigos com erros são automaticamente impedidos de entrar no repositório central. Uma vez que violações em DCL são apresentadas como erros no código, isto implicaria que nenhum desenvolvedor estaria apto a entregar seu código no repositório até que todos os erros existentes (violações) fossem resolvidos. Para evitar esse impacto, foi criado um ramo secundário de código exclusivo para o estudo de caso. Para esse ramo secundário, as regras de validação de entregas foram retiradas. Na Tabela 5.7 é possível ver algumas métricas relacionadas ao sistema após a criação do ramo secundário.

[2]https://jazz.net/products/rational-team-concert/
[3]https://maven.apache.org/index.html

Métrica	Valor
Tempo de desenvolvimento	31 meses
Linhas de código	30.000 (aprox.)
Classes	187
Pacotes	39
Camadas	6
Mudanças - *Commits*	9.368
Conjunto de mudanças	1.454
Tipos de artefatos	18
Número de artefatos	602
Desenvolvedores	21

Tabela 5.7: Métricas do sistema SSC-ADMIN

5.2.3 Formalização da Arquitetura

O *SSC-ADMIN* é um sistema decomposto em camadas, módulos fisicamente independentes, onde cada camada exerce uma função pré-estabelecida pela arquitetura. A especificação do sistema na linguagem DCL seguiu o mesmo critério de decomposição, ou seja, foi criada uma especificação DCL para cada camada do *SSC-ADMIN*. O sistema possui seis camadas. São elas: *ssc-admin-comum, ssc-admin-dominio, ssc-admin-interfaces, ssc-admin-infraestrutura, ssc-admin-negocio* e *ssc-admin-web*. Cada camada lógica (visão lógica) do sistema corresponde a um projeto Eclipse específico (visão de implementação). A Figura 5.2 mostra um diagrama contendo todas as camadas, bem como o respectivo projeto no IDE Eclipse. No diagrama da Figura 5.2-a é possível perceber também a relação de dependência entre as camadas. Essa informação foi fundamental para definir por onde começar a especificação da arquitetura.

Seguindo o planejamento estabelecido na metodologia, a primeira camada a ter sua arquitetura formalizada foi a camada *ssc-admin-comum*, por ser a menos dependente de todas as camadas. Consequentemente, a camada *ssc-admin-web* foi a última a ser formalizada na linguagem DCL, por ser a camada mais dependente das outras.

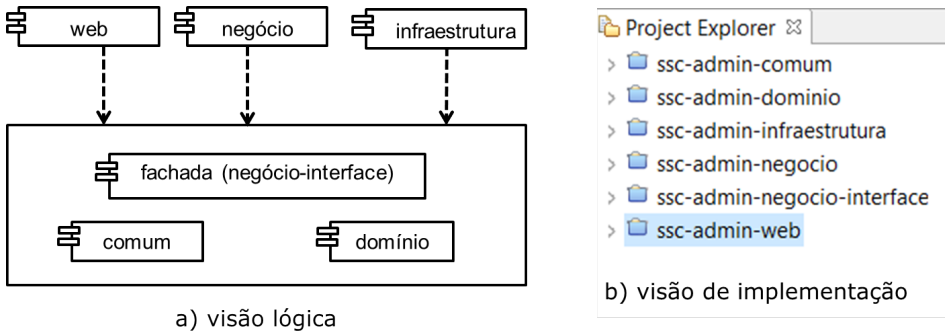

Figura 5.2: Componentes por camada.

Internamente, cada camada possui estrutura específica contento seus próprios componentes arquiteturais. Cada componente pode, ou não, ser composto de outros

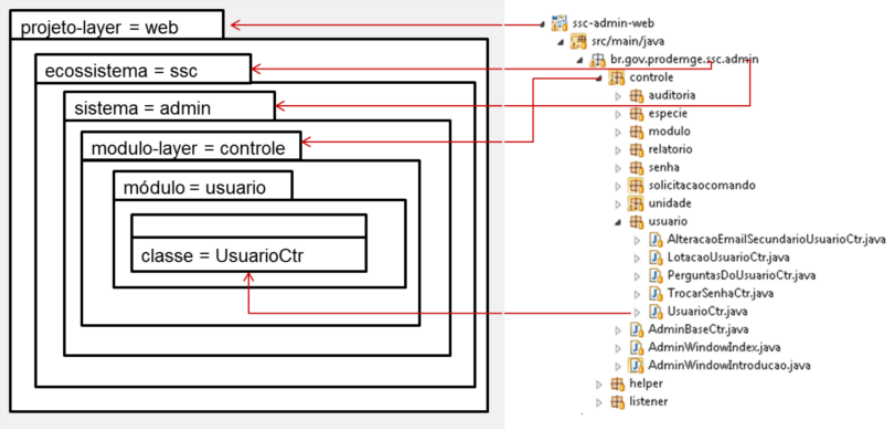

Figura 5.3: Visão lógica e Visão de implementação - *ssc-admin-web*.

componentes menores. Componentes maiores representam conceitos arquiteturais de mais alto nível (sistemas, módulos, etc.), enquanto que componentes menores representam conceitos de mais baixo nível (classes, enumerações, scripts, XMLs, etc.). A Figura 5.3 mostra parte da organização lógica do componente *ssc-admin-web*, bem como os seus respectivos componentes de código fontes. Embora com responsabilidades diferentes, internamente, as camada do sistema seguem modelos de estrutura semelhantes. Sendo assim, observando-se a visão lógica da camada *ssc-admin-web*, é possível ter uma ideia da estrutura lógica das demais camadas.

Como previsto na metodologia, após o entendimento dos modelos arquiteturais, a arquitetura de cada módulo foi definida na linguagem DCL 2.0. De acordo com a abordagem de DCL 2.0, cada artefato de código associado a um componente arquitetural é considerado uma instância desse componente. No entanto, para que um artefato seja associado a um determinado componente, a correspondência entre artefato de código e componente arquitetural precisa ser formalizada em DCL 2.0. Sendo assim, após a definição da hierarquia de componentes arquiteturais, os arquitetos definiram o mapeamento entre arquitetura e os artefatos de código. Como apresentado no Capítulo 3, em DCL 2.0 é possível fazer o mapeamento entre arquitetura e código, por meio de várias estratégias, sendo que a estratégia mais indicada pela abordagem é a estratégia de mapeamento por *convenção de nomes*. A escolha da estratégia de mapeamento por *convenção de nomes* se deu por dois motivos. Primeiro, reforçar o próprio uso das convenções, o que segundo os arquitetos, ajuda na compreensão do sistema. O segundo motivo foi em razão da estratégia ser pouco intrusiva, uma vez que não é necessário alterar o código para que o artefato seja reconhecido. Diferente da estratégia de mapeamento por *anotações*, por exemplo, a qual necessita de intervenção no código, além de só poder ser utilizada em artefatos Java. Uma vez

definida a estratégia de mapeamento, os arquitetos se basearam na documentação de nomenclatura da empresa para realizar o mapeamento entre arquitetura e código. O padrão de convenção de nomes da empresa utiliza sufixos ou prefixos para classificar os componentes arquiteturais. No entanto, algumas convenções encontradas no código, não estavam formalizadas na documentação, ou seja, a arquitetura evoluiu mas a documentação não acompanhou a evolução. A Listagem 5.1 mostra a especificação final em linguagem DCL 2.0 para a camada de *domínio*, onde é possível observar o mapeamento de cada componente definido por meio da palavra reservada *matching*. Pode-se observar nas *Linha 6 e 8*, os mapeamentos definidos para os componentes *Entity* e *AudityEntity*, respectivamente. De fato, essas declarações significam que qualquer artefato cujo nome termine com *VO* é uma instância do componente *Entity* e qualquer artefato cujo nome termine com *AudVO* é uma instância do componente *AudityEntity*. Por exemplo, a classe *UsuarioVO* é uma instância do componente *Entity* e a classe *AuditoriaAudVO* é um instância do componente *AudityEntity*. As especificações relativas às demais camadas podem ser observadas no Apêndice A desta dissertação.

```
1  architecture dominio{
2        ecosistema{matching: "xx.yyy.zzzzzzzz.{?}";
3              sistema{matching: "{?}";
4                    entidades{matching: "entidades";
5                          modulo{matching: "{?}";
6                                Entity{matching: "{?}VO";
7                                }
8                                AudityEntity{matching: "{?}AudVO";
9                                }
10                         }
11                         BaseEntity{matching: "{?}BaseVO";
12                         }
13                   }
14                   enums{matching: "enums";
15                         modulo{matching: "{?}";
16                               Enum{matching: "{?}Enum";
17                               }
18                         }
19                   }
20             }
21       }
22       ignore "xx", "yyy", "zzzzzzzz"...;
23 }
```

Listagem 5.1: Especificação da camada de domínio usando DCL 2.0

De maneira geral, quase todos os componentes de todas as camadas foram mapeados utilizando *convenção de nome* como estratégia. No entanto, em alguns poucos casos, foi utilizada a estratégia de mapeamento por *extensão de arquivo*. A Tabela 5.8

mostra o número de componentes por camada, formalizados na linguagem DCL 2.0. No estudo foi observado que o *Grau de cobertura* pode variar de módulo para módulo e a decisão de tornar uma arquitetura mais ou menos formal depende, principalmente, das características do módulo. A Tabela 5.9 mostra os dados computados pela ferramenta DCL 2.0 relativos ao *Grau de cobertura arquitetural* para todos os módulos calculados de acordo com o apresentado no Capítulo 3.

Componentes internos	
Projeto (camada)	Nº de componentes
web	25
negocio	21
infraestrutura	11
negocio-interface	14
dominio	10
comum	7
Total do sistema	**88**

Tabela 5.8: Métricas - Componentes internos por camada

Grau de cobertura		
Projeto (camada)	Por artefatos(%)	Por linhas(%)
web	82	97
negocio	63	94
infraestrutura	50	77
negocio-interface	53	77
dominio	67	93
comum	33	47

Tabela 5.9: Métricas - Grau de cobertura arquitetural

5.2.4 Verificação Arquitetural e Análise de Violações

Embora no planejamento, as definições de restrição tenham sido previstas para serem adicionadas nessa etapa, na etapa anterior —Formalização de arquitetura—, de certa forma, restrições de hierarquia já haviam sido definidas, uma vez que a própria especificação da hierarquia impõe, implicitamente, restrições à inserção de componentes à sua estrutura. Sendo assim, qualquer artefato de código que não se enquadre no modelo hierárquico gerará violação. Tendo essa questão em mente, os arquitetos iniciaram a adição das restrições do tipo relacionamento para cada módulo (camada). A Listagem A.1 mostra a especificação completa para a camada *ssc-admin-comum* incluindo as restrições DCL 1.0. Durante a especificação das restrições DCL 2.0, a equipe observou que as referências a componentes externos se repetiam muito. Por esse motivo, foi criado um projeto específico, onde os componentes externos foram especificados em DCL 2.0, desacoplando esses componentes das demais especificações. O módulo externo recebeu o nome de *platform*, o qual foi usado para especificar todos os

componentes externos relativos a *frameworks*, APIs, e a própria plataforma Java. A Tabela 5.10 reporta o número de componentes externos formalizados em DCL 2.0.

Componentes externos	
Projeto (camada)	Nº de componentes
platform	136
functional-test	11
TOTAL	**147**

Tabela 5.10: Métricas - Componentes externos

Ao analisar as primeiras violações, observou-se um número alto de *violações estruturais* (91%), se comparado com o número de violações do tipo relacionamento (9%). Após uma análise mais aprofundada do código fonte, os arquitetos concluíram que um bom número dessas violações estava relacionado a dois pontos especificamente: (1) a falta de atualização da arquitetura, ou seja, os conceitos arquiteturais evoluíram no código, mas não foram formalizados na arquitetura, tendo em vista que não foram encontrados na documentação. Por exemplo, a classe *EspecieFunctionalTest* foi considerada como *Componente Desconhecido* na primeira validação, uma vez que não havia correspondência com nenhum estereótipo existente na documentação. A solução foi formalizar o componente *FunctionalTest* em DCL 2.0; e (2) artefatos não se enquadravam em nenhum dos componentes arquiteturais pré-estabelecidos na arquitetura, embora sua estrutura e comportamento estivessem de acordo com algum componente já existente. Essa situação, segundo análise dos arquitetos, era causada pela não observância à convenção de nomes por parte do desenvolvedor. Como exemplo, observou-se que a classe *MailService* tinha as características de uma classe de acesso a dados e, portanto, foi renomeada para *NotificacaoDAO*.

Em função desses dois pontos, inicialmente, a maioria das recomendações para refatoração foram no sentido de formalizar a arquitetura, Ponto 1, e ajustar os artefatos de código de acordo com sua responsabilidade, Ponto 2. Em termos práticos, as ações objetivavam resolver violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*. Diante do volume de violações na hierarquia de componentes, resolveu-se dividir o processo de conformidade arquitetural em duas etapas, antes e depois da remoção dessas violações.

A Tabelas 5.11 mostra as camadas, o número de restrições e o número de referências formalizadas na linguagem DCL 2.0.

5.2.5 Remoção das Violações

Devido ao alto número de violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*, inicialmente, as ações dessa etapa seguiram em dois sentidos: 1) ajustar

Componentes internos		
Projeto (camada)	Nº de restrições	Nº de referências
web	8	42
negocio	11	72
infraestrutura	11	31
negocio-interface	8	20
dominio	8	22
comum	2	6
TOTAL	**45**	**193**

Tabela 5.11: Restrições - Referências

a especificação da arquitetura, alterando a especificação em DCL 2.0; e 2) ajustar os artefatos de código fonte, refatorando o código para que os artefatos se enquadrassem no modelo de componentes da arquitetura. Para facilitar o processo de refatoração, a Tabela 5.12 foi disponibilizada aos arquitetos para servir como guia nas tarefas de remoção de violação. A tabela inclui informações sobre o tipo de violação, o cenário onde ocorre e as ações a serem executadas para resolvê-las.

Componentes internos		
Tipo de violação	**Cenário**	**Solução**
Componente Desconhecido	Essa violação ocorre quando uma instância de componente é encontrada no código fonte do sistema, mas não há uma especificação formalizada em DCL 2.0.	1 - Formalizar o componente em DCL 2.0. 2 - Refatorar a instância do componente (renomear) no código fonte. 3 - Remover a instância do componente do código fonte.
Referência Desconhecida	Essa violação ocorre quando uma instância de componente aponta para uma instância de componente desconhecido.	1 - Remover a referência. 2 - Apontar para outra instância de componente. 3 - Formalizar o componente.
Localização Incorreta	Essa violação ocorre quando uma instância de componente é identificada pela arquitetura, porém sua localização dentro da arquitetura está errada.	1 - Remover a instância do componente. 2 - Mover a instância do componente para o local correto. 3 - Alterar a definição em DCL 2.0 para comportar esse novo componente no local.
Ausência de Componente Dominante	Essa violação ocorre quando uma instância de componente é identificada pela arquitetura, porém uma instância de componente do dominante relativo a esse componente não existe.	1 - Adicionar uma instância do componente dominante. 2 - Remover a instância do componente.

Tabela 5.12: Violação vs. Solução

Resolvidas as violações do tipo *Componente Desconhecido* e *Referência Desconhecida*, iniciou-se o processo de remoção das demais violações, até que a maioria das violações fossem removidas. O motivo de algumas violações não serem removidas é detalhado no fim dessa seção. Para identificar e documentar as violações foi utilizada

a API do RTC para recuperar as informações descritas no planejamento. No entanto, informações como autoria e data da violação necessitaram de uma estratégia combinada entre API e inspeção manual. A ferramenta RTC possui o recurso *annotate*, o qual revela o autor e data de cada linha de código de um arquivo. A Figura 5.4 mostra o recurso *annotate* sendo utilizado. No entanto, esse recurso mostra apenas a última alteração no artefato. Diante dessa limitação, foi verificado, para cada violação encontrada, o histórico do sistema para identificar quando cada linha havia sido inserida no código e qual seria o desenvolvedor responsável por essa inserção.

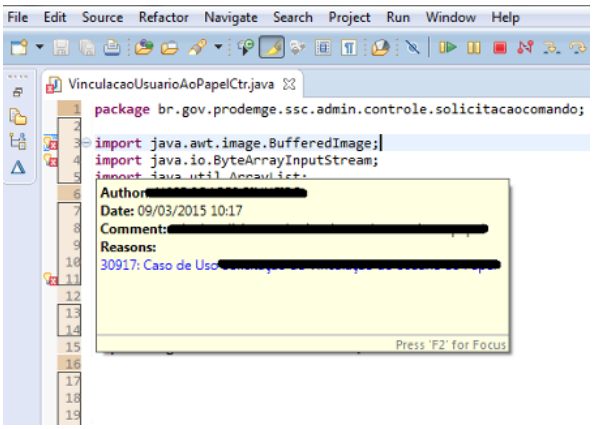

Figura 5.4: Recurso annotate da ferramenta - RTC

Uma parte das violações não pôde ser removida devido ao risco que essas violações apresentavam para o projeto, contudo, elas foram documentadas e apresentadas à gestão do projeto. A gestão ficou de fazer um planejamento para remoção das mesmas. Curiosamente, a maior parte dessas violações eram *violações estruturais* e ocorreram em artefatos não Java, o que atesta a hipótese de que não basta controlar apenas uma linguagem, tendo em vista a quantidade de tecnologia envolvida em um sistema de informação.

5.2.6 Geração de nova versão

Ao término das atividades de refatoração,—Remoção de violações—, foi feita a junção dos dois ramos de código. Após a junção, todos os módulo passaram por uma nova validação, a fim de capturar as possíveis violações ocorridas no período. Foram removidas 11 violações no período. Após a refatoração, tanto arquitetura quanto código ganharam novas versões e a ferramenta DCL 2.0 foi então instalada nas máquinas da equipe do Grupo 2 a fim de garantir a conformidade arquitetural desse ponto em diante.

5.2.7 Acompanhamento e Evolução

Logo nas primeiras semanas dessa fase, os arquitetos começaram a identificar violações no código por parte de alguns desenvolvedores, algo que deveria ser evitado, uma vez que a ferramenta tinha sido instalada na máquina desses desenvolvedores. Ao verificar o ocorrido, os arquitetos perceberam que esses desenvolvedores estavam desabilitando a ferramenta de validação no IDE Eclipse. A justificativa para tal situação, segundo os desenvolvedores, é que a validação instantânea da ferramenta estava impactando no desempenho da máquina. Devido a esse motivo, ajustes foram feitos na ferramenta a fim de possibilitar três tipos de validação: *Live feedback, Build feedback, Off-line feedback*, os quais estão descritos no Capítulo 3. Com os novos tipos de validação os desenvolvedores puderam escolher o tipo de validação mais adequado ao seu caso.

Para continuar garantindo a integridade da arquitetura, a tarefa de verificação arquitetural com a ferramenta DCL 2.0 passou a ser obrigatória para o *builder* (membro da equipe responsável por gerar a versão), uma vez que não havia garantia de que os desenvolvedores aplicariam a ferramenta no código entregue no repositório.

Como relatado na fase de refatoração, nem todas as violações foram removidas. Em função dessa questão, definiu-se que o *builder* do projeto não aceitaria nenhuma violação além dessas, quando da geração de uma nova versão de código. Essa fase durou 8 meses e 39 violações foram evitadas.

5.3 Resultados

Os resultados apresentados nessa seção tiveram como base o histórico de desenvolvimento do sistema, período compreendido entre Junho de 2013 e Abril de 2016, 35 meses. O processo de especificação da arquitetura e remoção das violações do sistema teve início em 2 de Julho de 2015 e durou até 27 de Agosto de 2015, 58 dias. Nesse período, DCL 2.0 foi utilizada para revelar as violações ocorridas desde o início do sistema. Após a remoção das violações a ferramenta de verificação de DCL 2.0 foi instalada nas máquinas da equipe. A etapa de acompanhamento teve inicio no mês de Setembro de 2015 e se estendeu até Abriu de 2016, 8 meses. Como discutido nas seções anteriores, o processo de conformidade arquitetural do sistema foi dividido em duas etapas em função do número de violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*. A Figura 5.5 mostra o número de violações encontradas antes (Etapa 1) e depois (Etapa 2) da remoção da maior parte dessas violações. A Figura 5.6 mostra as violações divididas em três grupos: Grupo 1, composto por *Componente Desconhecido* e *Referência Desconhecida*; Grupo 2, contendo as demais violações de DCL 2.0; e

Grupo 3, contendo as violações DCL 1.0.

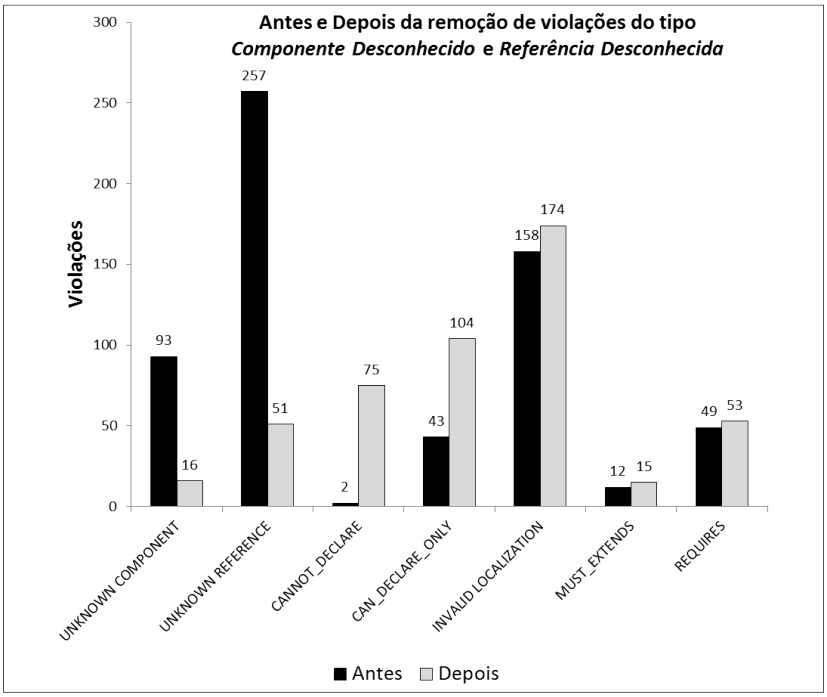

Figura 5.5: Antes e depois da remoção de violações

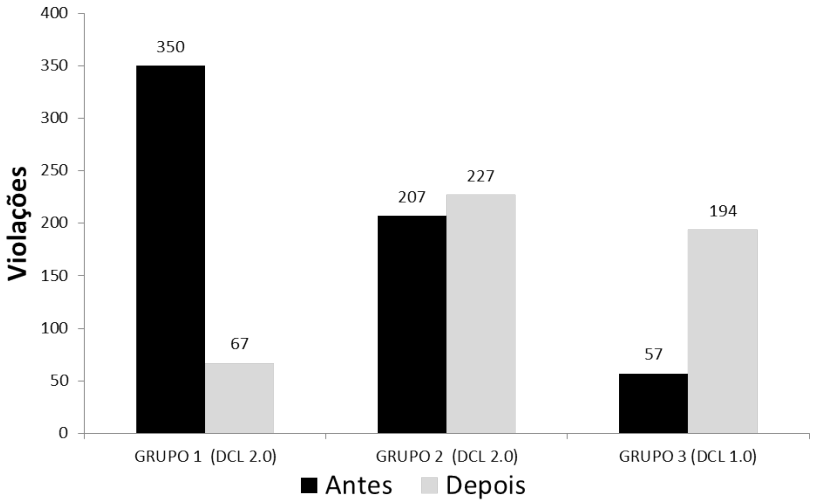

Figura 5.6: Número de violações agrupadas

A Figura 5.7 mostra a quantidade de mudanças (*commits*) feitas no período, bem como a quantidade de mudanças com violações no período. Vale ressaltar que o gráfico não revela todas as violações ocorridas no período, mas as violações ocorridas no período que não foram removidas do código até a data de início do estudo de caso. A

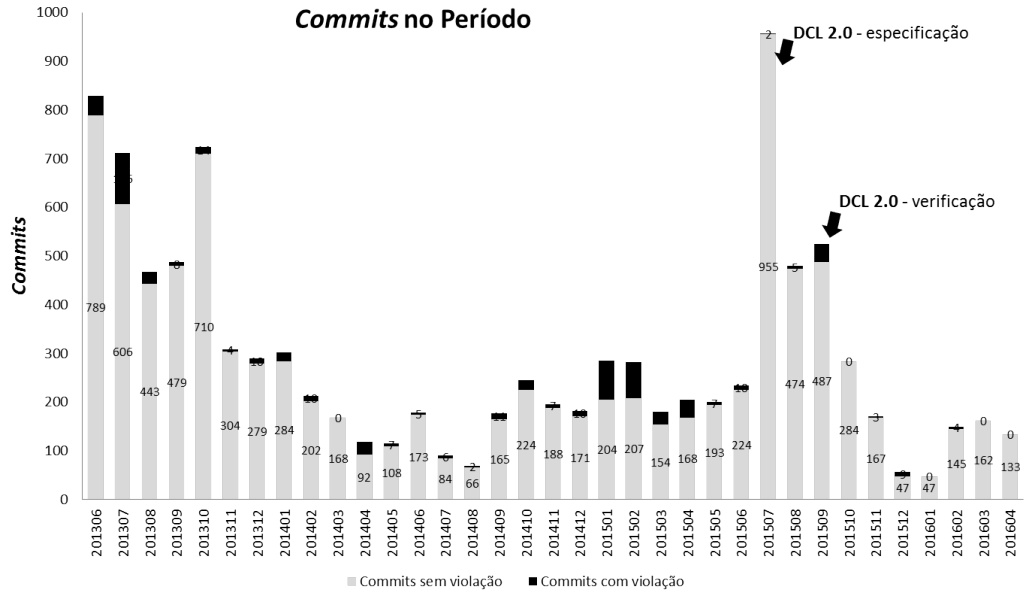

Figura 5.7: Quantidade de violações/mudanças no período

Figura 5.8 mostra a quantidade de funcionalidades adicionadas ao projeto no período, bem como a quantidade de violações inseridas no projeto no mesmo período.

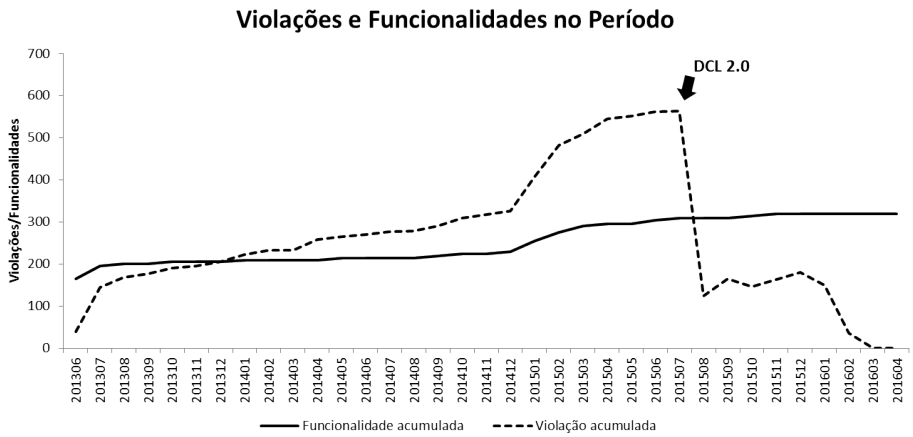

Figura 5.8: Quantidade de violações/funcionalidades no período

A Figura 5.9-a mostra a quantidade de violações por perfil de desenvolvedor. No mesmo gráfico também é apresentada a quantidade de violações de acordo com a complexidade, dificuldade de remoção da violação, para cada perfil. Na Figura 5.9-b é apresentada a quantidade de conjuntos de mudanças, conjunto de *commits*, de acordo com o perfil dos desenvolvedores.

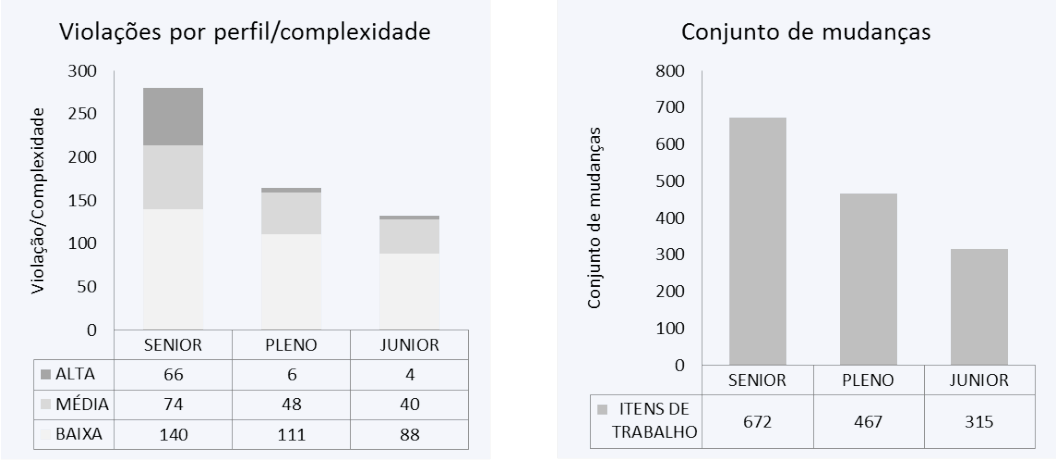

Figura 5.9: Violações por perfil e mudanças por perfil

5.4 Análise dos Resultados

A Figura 5.5 mostra o alto número de violações dos tipos *Componente Desconhecido* e *Referência Desconhecida* em relação aos demais tipos de violação. No entanto, no gráfico é possível observar o aumento nos demais tipos de violações depois da remoção das violações dos tipos *Componente Desconhecido* e *Referência Desconhecida*, o que reforça a hipótese de que essas violações encobrem violações dos demais tipos. Essa constatação fica mais evidente no gráfico da Figura 5.6, onde também é possível perceber que violações do tipo relacionamento são mais afetadas por violações do tipo *Componente Desconhecido* e *Referência Desconhecida*. Ou seja, a remoção de violações do tipo *Componente Desconhecido* e *Referência Desconhecida* aumenta a possibilidade de mais violações do tipo relacionamento serem reveladas.

A Figura 5.7 mostra a evolução das violações durante o período de desenvolvimento do sistema. O gráfico mostra que não há uma relação direta entre quantidade de mudanças (*commits*) e a quantidade de violações adicionadas, uma vez que é possível observar períodos com muitas mudanças e poucas violações e vice-versa. Com exceção de dois períodos, observa-se um padrão constante de adição de violação ao longo do desenvolvimento. Ao observar em detalhes os dois períodos, percebeu-se que o motivo do número de violações ser maior nos meses *06/2013* e *07/2013* se deve ao fato de que nesse período foi adicionado a maior parte das classes Java no projeto, o que consequentemente tende a gerar mais violações, uma vez que a maioria das restrições arquiteturais tem como alvo classes Java. No outro período fora dos padrões, correspondentes aos meses de janeiro a abril de 2015, dois fatores contribuíram para o aumento das violações: 1) adição de novas funcionalidades para permitir a funcionalidades de solicitação

e aprovação de alteração de dados no sistema; e 2) adição de código para extensão de um *framework* JSON. Observando-se o gráfico da Figura 5.8, nota-se certa relação entre adição de novas funcionalidades e adição de violações. No entanto, fica evidente que o crescimento das duas métricas seguem proporções diferentes. Logo, pode-se concluir que períodos com adição de novas funcionalidades geraram mais violações do que em períodos onde houve apenas manutenções de funcionalidades já existentes.

No gráfico da Figura 5.8, pode-se observar o impacto do processo de conformidade arquitetural a partir do mês de agosto de 2015, uma vez que nesse período o número de violações cai a níveis de início do projeto.

Na Figura 5.9-a, observa-se que de maneira geral, todos os perfis geraram violações no sistema. Observando-se a Figura 5.9-b em relação à Figura 5.9-a pode-se notar uma correlação entre conjunto de mudanças e violações, ou seja, quem muda mais o sistema, incorpora mais violações. No entanto, observando-se apenas as violações classificadas como de complexidade ALTA, percebe-se que, o perfil SÊNIOR, o mais experiente, inseriu aproximadamente dez vezes mais violações que os demais perfis. Essa questão se deve ao fato de que a esse perfil são delegadas as tarefas mais complexas e consequentemente essas tarefas tem maior probabilidade de gerar impacto na arquitetura.

5.5 Discussão

Com a intenção de atender as questões colocadas para esse estudo de caso, esta seção tenta responder tais questões tendo como base os resultados obtidos. Seguem as considerações sobre cada questão.

5.5.1 Caracterização das Violações

Q1.1 - Porque as violações são geradas?

Após a análise dos dados, percebe-se que as violações de arquitetura apareceram bem mais cedo no processo de desenvolvimento do sistema. Ou seja, o processo de degeneração do sistema não obedece um padrão linear. Observando-se os resultados, fica evidente a relação entre geração de violação e criação de novas funcionalidades, ou seja, mudanças que adicionam funcionalidades no sistema são mais propensas a incorporar violações no sistema do que mudanças que tenham por objetivo a manutenção de funcionalidade. Outra questão interessante diz respeito ao perfil dos autores das violações. De maneira geral, todos os perfis adicionam violações no sistema, no entanto, violações mais complexas são adicionadas por desenvolvedores

mais experientes. Segundo os desenvolvedores e arquitetos, as principais causas de geração de violação foram: 1) o desconhecimento das convenções arquiteturais, uma vez que a documentação existente era muito extensa, fazendo com que os desenvolvedores se sentissem desestimulados a consultá-la; e 2) a necessidade de evolução da arquitetura, pois, ao se depararem com uma limitação da arquitetura, os desenvolvedores construíam suas próprias soluções a revelia da equipe de arquitetura da empresa. Pressões de projeto foram citadas como fatores secundários, uma vez que os desenvolvedores disseram que nunca fizeram uma escolha contrária às convenções arquiteturais quando a convenção era conhecida. No entanto, os desenvolvedores ressaltaram que a pressão de projeto pode ter dificultado a busca por mais informações sobre a arquitetura quando determinada convenção não era conhecida.

Q1.2 - Como as violações são tratadas?

Observando-se o histórico do código fonte do sistema, percebe-se que algumas violações foram removidas, onde se conclui que a equipe tinha consciência das mesmas, ainda que essas correções tenham sido feitas de maneira espontânea e sem nenhum planejamento ou processo. No entanto, fica evidente ao analisar os dados do histórico do sistema que a maioria das violações removidas antes do estudo de caso com DCL 2.0, era de baixa complexidade, ou seja, as violações de alta complexidade tendem a permanecer no sistema. Indagados sobre essa questão, os desenvolvedores relataram que o risco de introduzir *bugs* no sistema foi o fator principal para a não remoção das violações de alta complexidade. Os desenvolvedores também disseram que o fato de não terem tido uma ferramenta que apontasse as violações arquiteturais desde o início do projeto, também pode ser considerado um fator que dificulta a remoção das mesmas, uma vez que a inspeção manual é bastante difícil e custosa.

5.5.2 DCL 2.0

Q2.1 - DCL 2.0 pode ser usada para evitar violações arquiteturais?

Passados dez meses do início do processo de conformidade arquitetural, ao analisar os resultados é possível observar que o número de violações se manteve controlado. Mesmo que tenha havido a adição de funcionalidades e manutenção nos meses avaliados, o sistema se manteve íntegro em relação aos conceitos arquiteturais.

Q2.2 - Os novos conceitos propostos por DCL 2.0 podem melhorar o processo de verificação arquitetural?

1) Modelagem hierárquica e modular se mostrou bastante efetiva, uma vez que o

conceito permitiu que a arquitetura dos sistemas pudesse ser especificada de maneira fiel à estrutura dos mesmos, tanto do ponto de vista de decomposição, permitindo especificar vários módulos independentes, quanto do ponto de vista de níveis de abstração, onde foi possível especificar aspectos desde o nível de sistema até o nível de arquivo e classes. *2) Novos tipos de restrições* para *Componente Desconhecido* e *Referência Desconhecida* foram importantes para explicitar o nível de informalidade no qual o sistema se encontrava, ou seja, esses dois tipos de violações foram essenciais para ajustar o *grau de cobertura arquitetural* ao nível desejado pela equipe. Outra evidência da eficácia desses dois tipos de restrições pode ser observada por meio dos resultados, os quais mostram que a revelação das demais violações aumentou após a remoção das violações do tipo *Componente Desconhecido* e *Referência Desconhecida*. *3) Referência cruzada* entre módulos foi importante para a prevenção de erros e para a celeridade no processo de especificação da arquitetura do sistema, uma vez que um componente era definido em um módulo e apenas referenciado em outro. *4) A reusabilidade* da abordagem apresenta um grande potencial, uma vez que as definições de arquitetura são completamente independentes dos sistemas alvo. No entanto, ainda é preciso que a técnica seja testada em mais sistemas com características similares para melhor avaliar esse aspecto. *5)* O desacoplamento entre arquitetura e sistemas alvo trazida por DCL 2.0 possibilitou que conceitos e práticas de *Gestão e Configuração* fossem aplicadas à especificação de arquitetura da mesma forma que se faz com o código fonte. Essa característica foi importante para controlar o processo de evolução da arquitetura, uma vez que cada módulo de arquitetura teve em média 8 versões ao longo do estudo de caso.

Q2.3 - A nova versão da ferramenta pode ser usada em processos reais de verificação de arquitetura?

Embora os aspectos relativos a essa questão possam ser considerados apenas de carácter técnico, entende-se que esses aspectos podem ser importantes para a adoção ou não da técnica. Ao longo do estudo várias funcionalidades foram melhoradas e adicionadas a fim de atender um cenário real de desenvolvimento. Por exemplo, os novos tipos de validação: *Build feedback, Off-line feedback* permitiram que máquinas com menos poder de processamento pudessem realizar a verificação arquitetural. As funcionalidades de *visualização de arquitetura, visualização de logs e validação parcial* foram essenciais durante o processo de especificação da arquitetura do sistema. As funcionalidade de referência cruzada, verificação de erros e auto-completar fornecidas pelo editor da linguagem DCL 2.0 facilitaram bastante os trabalhos, uma vez que erros de especificação eram detectados instantaneamente. A integração com outras ferramentas como Maven, foi importante para garantir a distribuição das versões

das especificações arquiteturais. Em relação ao desempenho, é necessário que sejam feitas melhorias para que o tempo de resposta de validação de arquitetura melhore, sobretudo para que a validação instantânea (*Live feedback*) possa ser utilizada de forma efetiva.

5.6 Considerações Finais

Este capítulo apresentou um estudo de caso como forma de avaliação da solução proposta para esta dissertação de mestrado, onde um sistema de grande porte passou por um processo de conformidade arquitetural utilizando a linguagem DCL 2.0 como principal ferramenta de apoio. Na Seção 5.1 foi apresentada a metodologia usada no estudo, bem como uma proposta de um processo para servir como guia em projetos de refatoração com foco em conformidade arquitetural. Cada etapa do processo teve seu objetivo definido, assim como as atividades a serem realizadas em cada uma delas.

Na Seção 5.2 foram narradas todas as ações realizadas durante o estudo, bem como os desafios e dificuldades encontradas durante a execução dos trabalhos. O sistema teve sua arquitetura especificada em DCL 2.0, totalizando 45 restrições e 235 componentes arquiteturais (internos e externos) formalizados, sendo esses componentes correspondentes a vários níveis de abstração diferentes, e.g., projetos, módulos, classes, arquivos, etc. Por fim, o sistema teve todas as suas violações removidas no fim do período de acompanhamento e evolução do sistema, um total de 771 violações, sendo 74% dessas violações detectadas por restrições contidas apenas em DCL 2.0.

A Seção 5.3, apresentou o resultado da verificação de conformidade arquitetural aplicada ao sistema alvo após 26 meses de seu desenvolvimento, nesse momento 771 violações foram detectadas. Nessa seção outros resultados obtidos após o processo de verificação de conformidade (oito meses) também foram apresentados, nesse período 39 violações teria sido adicionadas ao sistema se DCL 2.0 não estivesse sendo utilizada. Em detalhes a seção apresentou o número de funcionalidades, mudanças e violações ocorridas durante todo o ciclo de desenvolvimento do sistema, compreendendo 35 meses de desenvolvimento.

Na Seção 5.4, os resultados obtidos foram analisados e pontos específicos foram discutidos, com destaque para o número de violações dos tipos *Componente Desconhecido* e *Referência Desconhecida* e o impacto causado por elas. Outro ponto discutido na seção diz respeito ao perfil dos desenvolvedores, onde se constatou que, no contexto

do estudo de caso, desenvolvedores mais experientes inseriram cerca de dez vezes mais violações complexas no sistema do que os demais perfis. Por fim, na Seção 5.5, as principais percepções obtidas ao longo do estudo foram apresentadas tendo como ponto de partida as questões de pesquisa colocadas no início do estudo.

Capítulo 6

Conclusão

No Capítulo 1 desta dissertação de mestrado, foram destacados os impactos positivos na qualidade de um sistema de software quando a arquitetura planejada e a arquitetura concreta de um sistema convergem. Ressaltou-se, no entanto, que sistemas de software perdem características importantes de suas arquiteturas ao longo de sua evolução, dando origem à erosão arquitetural. No Capítulo 2, técnicas propostas para enfrentar esse problema foram apresentadas, dentre elas a linguagem DCL *(Dependency Constraint Language)*. No Capítulo 3, como contribuição central dessa dissertação de mestrado, foi apresentada uma nova versão de DCL, denominada DCL 2.0. Essa versão introduz como principal novidade o suporte à especificação arquitetural de forma modular, reutilizável e hierárquica. No Capítulo 4, a ferramenta que implementa o principais conceitos de DCL 2.0 foi apresentada. No Capítulo 5, apresentou-se uma avaliação, onde DCL 2.0 foi utilizada para auxiliar em um processo de conformidade arquitetural, tendo como alvo um sistema de grande porte. Os resultados comprovaram a aplicabilidade da extensão proposta, uma vez que 74% das violações somente puderam ser detectados devido às novas funcionalidades propostas em DCL 2.0. Além de bons resultados na detecção de violações existentes, os resultados também mostraram que DCL 2.0 foi importante no que se refere ao controle permanente de violações arquiteturais, pois, passados dez meses de implantação de DCL 2.0 como ferramenta de verificação arquitetural no sistema alvo, o número de violações se manteve estável por um período, sendo zerado no fim do processo. Diante dos resultados apresentados, conclui-se que DCL 2.0 pode desempenhar um papel positivo no contexto de conformidade arquitetural.

As próximas seções desse capítulo estão organizadas da seguinte forma. Na Seção 6.1 são apresentados os principais resultados obtidos no experimento. Na Seção 6.2 ressalta-se as principais contribuições desse trabalho. Por fim, na Seção 6.3, algumas

propostas para trabalhos futuros são apresentadas.

6.1 Principais Resultados

Os seguintes resultados importantes relacionados ao fenômeno de erosão arquitetural foram observados durante o experimento:

1. Violações complexas de arquitetura são realizadas principalmente por desenvolvedores mais experientes e tendem a permanecer no sistema. Violações estruturais podem esconder outros tipos de violações, uma vez que essas violações dificultam a identificação dos componentes. Violações ocorrem já nas fases iniciais do sistema e de maneira geral ocorrem com maior frequência em manutenções evolutivas do que em manutenções corretivas.

2. O primeiro passo para se validar uma arquitetura é identificar os seus componentes. Se não se conhece tais componentes, não se pode dizer nada a respeito de suas características. O risco das violações arquiteturais é menor em componentes conhecidos, pois nesses componentes, ainda que exista alguma violação, ela é controlada e sua remoção pode ser planejada. Já No entanto, os componentes desconhecidos e referências desconhecidas (componentes externos) são um problema silencioso, e pode vir a ser descoberto apenas quando o sistema já estiver enfrentando um sério problema.

3. A percepção após a execução da avaliação reportada nessa dissertação é que a evolução da arquitetura é constante, embora ocorra de maneira informal gerando, principalmente, violações estruturais. Embora violações não sejam necessariamente um código ruim, elas estão em desacordo com as normas arquiteturais. Nesse sentido, são tão prejudiciais para o sistema quanto um código ruim, dado o prejuízo que o mesmo traz para a compreensão do sistema e, consequentemente, para os demais atributos de qualidade de software.

4. Não há uma definição universalmente aceita do que é arquitetura de software e fica a cargo de cada organização julgar o que deve ser considerado ou não como um componente arquitetural [Kruchten, 2008]. No experimento, DCL 2.0 mostrou-se bastante flexível, possibilitando aos arquitetos definir a arquitetura de cada componente de acordo com cada contexto. Foi importante para a equipe envolvida no experimento saber qual o real escopo de suas especificações, o que de certa forma os incentivou a evoluir as especificações arquiteturais a fim de alcançar o maior número

de componentes possíveis. Nesse sentido, o grau de cobertura arquitetural fornecido pela ferramenta DCL 2.0 foi muito importante.

6.2 Contribuições

As principais contribuições desta dissertação de mestrado estão divididas em três partes: (i) novos conceitos e funcionalidades adicionados à linguagem DCL, (ii) um processo detalhado de refatoração de sistemas com foco em conformidade arquitetural e (iii) uma ferramenta que implementa os principais conceitos de DCL 2.0.

1. Linguagem DCL 2.0 - A modelagem modular e desacoplada do projeto alvo facilitou o reúso e a evolução da especificação arquitetural, viabilizando a realização do experimento, uma vez que seria bem mais custoso realizar a especificação de forma monolítica. A modelagem hierárquica teve sua efetividade demonstrada no experimento realizado, uma vez que usando a linguagem foi possível especificar a arquitetura em vários níveis de abstração diferentes: projeto, módulos, classes, arquivos de maneira fiel a arquitetura real do sistema. As novas restrições estruturais adicionadas à linguagem permitiram que violações, não apenas relacionadas ao controle de dependência, mas também aquelas relacionadas à estrutura do sistema, pudessem ser reveladas no código fonte. Portanto, a extensão proposta da linguagem DCL, com adição e melhoria das funcionalidades citadas, representa uma contribuição para os interessados em fazer uso dessa linguagem em cenários reais de desenvolvimento de software.

2. Processo - O processo de remoção de violações arquiteturais sugerido por [Tvedt et al., 2002] serviu como base para a definição do processo utilizado no experimento. No entanto, algumas adaptações no processo foram necessárias para torná-lo mais iterativo e incremental e, assim, evitar maiores impactos no desenvolvimento do sistema alvo. As principais alterações no processo foram: 1) Criação de três novas etapas: *Preparação da equipe, Geração de versão, e Acompanhamento e evolução*; 2) Mudança na ordem de duas etapas, *Preparação do código fonte* e *Formalização da arquitetura*; e 3) Junção de duas etapas, *Verificação de arquitetura* e *Análise de violações*. Algumas premissas definidas no início do processo também facilitaram a execução dos trabalhos. Por exemplo, em processos de conformidade arquitetural envolvendo sistemas altamente modularizados é ideal iniciar-se pelos módulos menos dependentes. Outra lição importante aprendida durante o processo é que violações estruturais devem ser resolvidas antes da verificação dos demais tipos de violações,

pois elas, de certo modo, ajudam no processo de identificação dos componentes. Outro importante aspecto a ser considerado em processos de conformidade arquitetural é o grau de cobertura da especificação, ou seja, arquitetos precisam saber o quanto do código fonte de um sistema está sendo alcançado pela especificação de arquitetura. Definições de arquitetura podem ter mais ou menos formalidade, ficando a cargo dos arquitetos definir para cada arquitetura o grau de cobertura desejado em um dado momento. Como contribuição, essa dissertação de mestrado apresenta esse processo de refatoração de sistemas de software com foco em conformidade arquitetural que pode servir como guia para desenvolvedores e arquitetos.

3. Ferramenta - Outra contribuição do trabalho é a ferramenta DCL 2.0, a qual foi projetada tendo como requisitos as principais questões colocadas nesta dissertação de mestrado. Funcionalidades como especificação de arquitetura foram aprimoradas e outras como visualização e medição de arquitetura foram adicionadas. A funcionalidade de integração com o *framework Maven* foi fundamental no processo de distribuição da especificação nas máquinas dos desenvolvedores. Distribuir manualmente a especificação a cada evolução da arquitetura impactaria enormemente o dia-a-dia de desenvolvimento do sistema, inviabilizando o experimento. A ferramenta foi implementada e disponibilizada na forma de um conjunto de *plug-ins* para a plataforma Eclipse, de forma que desenvolvedores possam utilizá-la em seus processos de desenvolvimento. Uma API também foi disponibilizada, a qual pode auxiliar outros pesquisadores a projetarem novas soluções usando a linguagem DCL 2.0.

6.3 Trabalhos Futuros

A avaliação de DCL 2.0 em um cenário real de desenvolvimento de software possibilitou a percepção de pontos importantes de melhorias em seu projeto e implementação. Algumas destas melhorias foram implementadas ainda durante o período do mestrado. As demais oportunidades percebidas durante o experimento podem ser objeto de trabalhos futuros. Em seguida essas oportunidades são apresentadas.

1. *Novos Tipos de Visualização* - Com DCL 2.0 alguns tipos de visualização foram adicionados na ferramenta para auxiliar nas tarefas de manutenção e evolução, uma vez que as visualizações ajudam na compreensão do sistema. No entanto, novas soluções podem fazer uso da API de DCL 2.0 para recuperar a arquitetura de um sistema e projetá-la em outros formatos de visualização, como exemplo, visualizações *3D* [Viana et al., 2015; Wettel & Lanza, 2008].

2. *Níveis de abstração* - DCL 2.0 limita-se a reconhecer artefatos no nível de classe, arquivos e pacotes. Em trabalhos futuros, seria importante que a especificação de arquitetura suportasse o mapeamento de elementos de granularidade mais fina como métodos, atributos, *tags* XML ou funções JavaScript.

3. *Métricas de qualidade* - Uma vez que DCL traz um modelo detalhado de identificação dos componentes arquiteturais, métricas de qualidade poderiam ser atribuídas a esses componentes de acordo com suas características. Uma vez atribuídas, tais métricas poderiam ser verificadas por ferramentas automatizadas.

4. *Novas visões arquiteturais* - DCL 2.0 concentra-se basicamente na validação de aspectos estruturais da arquitetura. No entanto, outros aspectos da arquitetura também são importantes, [Kruchten, 2008]. Nesse sentido, novos aspectos arquiteturais poderiam ser adicionados à linguagem. Por exemplo, a linguagem poderia adicionar restrições para controlar aspectos relacionados a visão de *deployment*, ou seja, verificar se o sistema está devidamente implantado.

5. *Catálogo de arquiteturas* - Como em DCL 2.0 as definições de arquitetura são reutilizáveis, um repositório de especificações poderia ser criado para ser compartilhado entre desenvolvedores, arquitetos e pesquisadores.

6. *Instanciação de arquiteturas* - Como DCL 2.0 possibilita especificar a estrutura arquitetural com bastante detalhe, trabalhos futuros podem explorar essa característica para automatizar a criação de sistemas baseados na especificação DCL 2.0.

7. *Rastreabilidade conceitual* - Como a nova restrição ***requires***, DCL 2.0 evita violações do tipo *Ausência de Componente Dominante*, o que na prática garante a rastreabilidade entre elementos essenciais do domínio e o código fonte. Novos trabalhos podem explorar essa característica de DCL 2.0 e propor visualizações que tenham como raiz os elementos essenciais, funcionalidade, casos de uso, entidades, de forma próxima à especificação de *feature models* no contexto de linhas de produto de software. Algumas métricas interessantes poderiam ser obtidas tendo como origem tais elementos essenciais. Por exemplo, quantos artefatos estão associados à funcionalidade *Registrar Venda*, ou qual valor máximo de complexidade ciclomática associado ao conceito *Nota Fiscal*.

Referências Bibliográficas

Antkiewicz, M. & Czarnecki, K. (2006). Framework-specific modeling languages with round-trip engineering. Em *Model Driven Engineering Languages and Systems*, pp. 692–706. Springer.

Baldwin, C. Y. & Clark, K. B. (1999). *Design rules: The power of modularity*. MIT Press.

Brooks, F. P. J. (1987). No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19.

Brunet, J.; Guerrero, D. & Figueiredo, J. (2009). Design tests: An approach to programmatically check your code against design rules. Em *31st International Conference on Software Engineering-Companion (ICSE)*, pp. 255–258.

Brunet, J.; Murphy, G. C.; Serey, D. & Figueiredo, J. (2015). Five years of software architecture checking: A case study of Eclipse. *IEEE Software*, 32(5):30–36.

De Moor, O.; Verbaere, M.; Hajiyev, E.; Avgustinov, P.; Ekman, T.; Ongkingco, N.; Sereni, D. & Tibble, J. (2007). Keynote address: .ql for source code analysis. Em *7th IEEE International Working Conference on Source code Analysis and Manipulation (SCAM)*, pp. 3–16.

Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley.

Knodel, J. (2002). *Process models for the reconstruction of software architecture views*. Tese de doutorado, Universität Stuttgart.

Knodel, J.; Lindvall, M.; Muthig, D. & Naab, M. (2006). Static evaluation of software architectures. Em *10th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 294–304.

Knodel, J.; Muthig, D.; Haury, U. & Meier, G. (2008a). Architecture compliance checking-experiences from successful technology transfer to industry. Em *12th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 43–52.

Knodel, J.; Muthig, D. & Rost, D. (2008b). Constructive architecture compliance checking-an experiment on support by live feedback. Em *IEEE International Conference on Software Maintenance (ICSM)*, pp. 287–296.

Knodel, J. & Popescu, D. (2007). A comparison of static architecture compliance checking approaches. Em *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pp. 12–12.

Koschke, R. & Simon, D. (2003). Hierarchical reflexion models. Em *10th Working Conference on Reverse Engineering (WCRE)*, pp. 36–45.

Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50.

Kruchten, P. (2008). What do software architects really do? *Journal of Systems and Software*, 81(12):2413–2416.

Lee, H.; Antkiewicz, M. & Czarnecki, K. (2008). Towards a generic infrastructure for framework-specific integrated development environment extensions. Em *2nd International Workshop on Domain-Specific Program Development*, pp. 1–6.

Maffort, C.; Valente, M. T.; Anquetil, N.; Hora, A. & Bigonha, M. (2013a). Heuristics for discovering architectural violations. Em *20th Working Conference on Reverse Engineering (WCRE)*, pp. 222–231.

Maffort, C.; Valente, M. T.; Bigonha, M.; Silva, L. H. & Aparecido, G. (2013b). Archlint: Uma ferramenta para detecção de violações arquiteturais usando histórico de versões. Em *4rd Brazilian Conference on Software: Theory and Practice, Tools Demonstration Track (CBSoft)*, pp. 1–6.

Maffort, C.; Valente, M. T.; Terra, R.; Bigonha, M.; Anquetil, N. & Hora, A. (2016). Mining architectural violations from version history. *Empirical Software Engineering*, 21(3):854–895.

Miodonski, P.; Forster, T.; Knodel, J.; Lindvall, M. & Muthig, D. (2004). Evaluation of software architectures with Eclipse. *Institute for Empirical Software Engineering (IESE)*, 107.

Miranda, S.; Rodrigues, E.; Valente, M. T. & Terra, R. (2016). Architecture conformance checking in dynamically typed languages. *Journal of Object Technology*, 1(1):1–35.

Miranda, S.; Valente, M. T. & Terra., R. (2015a). Archruby: conformidade e visualização arquitetural em linguagens dinâmicas.1. Em *6th Brazilian Conference on Software: Theory and Practice (Tools Track)*, pp. 1–8.

Miranda, S.; Valente, M. T. & Terra, R. (2015b). Conformidade e visualização arquitetural em linguagens dinâmicas. Em *18th Congresso Ibero-americano de Engenharia de Software (CIbSE)*, pp. 1–14.

Mitschke, R.; Eichberg, M.; Mezini, M.; Garcia, A. & Macia, I. (2013). Modular specification and checking of structural dependencies. Em *12th Annual International Conference on Aspect-oriented Software Development(AOSD)*, pp. 85–96.

Murphy, G. C.; Notkin, D. & Sullivan, K. (1995). Software reflexion models: Bridging the gap between source and high-level models. Em *ACM Software Engineering Notes*, pp. 18–28.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058.

Passos, L.; Terra, R.; Valente, M. T.; Diniz, R. & Mendonca, N. (2010). Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89.

Roberts, D. & Johnson, R. (1997). Patterns for evolving frameworks. Em *Pattern Languages of Program Design 3*, pp. 471–486. Addison-Wesley.

Rosik, J.; Le Gear, A.; Buckley, J.; Babar, M. A. & Connolly, D. (2011). Assessing architectural drift in commercial software development: a case study. *Software: Practice and Experience*, 41(1):63–86.

Sangal, N.; Jordan, E.; Sinha, V. & Jackson, D. (2005). Using dependency models to manage complex software architecture. Em *ACM Sigplan Notices*, pp. 167–176.

Shaw, M. & Clements, P. (2006). The golden age of software architecture. *IEEE Software*, 23(2):31–39.

Sudkamp, T. A. (2005). *Languages and Machines: An introduction to the theory of computer science*. Addison-Wesley.

Sullivan, K. J.; Griswold, W. G.; Cai, Y. & Hallen, B. (2001). The structure and value of modularity in software design. *ACM SIGSOFT Software Engineering Notes*, 26(5):99–108.

Terra, R. & Valente, M. T. (2008). Towards a dependency constraint language to manage software architectures. Em *2nd European Conference on Software Architecture (ECSA)*, pp. 256–263.

Terra, R. & Valente, M. T. (2009). A dependency constraint language to manage object-oriented software architectures. *Software: Practice and Experience*, 39(12):1073–1094.

Terra, R. & Valente, M. T. (2010). Definição de padrões arquiteturais e seu impacto em atividades de manutenção de software. Em *7th Workshop de Manutenção de Software Moderna (WMSWM)*, pp. 1–8.

Terra, R.; Valente, M. T.; Bigonha, R. & Czarnecki, K. (2012a). DCLfix: A recommendation system for repairing architectural violations. Em *3rd Brazilian Conference on Software: Theory and Practice (CBSoft), Tools Demonstration Track*, pp. 63–68.

Terra, R.; Valente, M. T.; Czarnecki, K. & Bigonha, R. S. (2012b). Recommending refactorings to reverse software architecture erosion. Em *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 335–340.

Terra, R.; Valente, M. T.; Czarnecki, K. & Bigonha, R. S. (2012c). Recommending refactorings to reverse software architecture erosion. Em *16th European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 335–340.

Terra, R.; Valente, M. T.; Czarnecki, K. & Bigonha, R. S. (2015). A recommendation system for repairing violations detected by static architecture conformance checking. *Software: Practice and Experience*, 45(3):315–342.

Tonella, P.; Torchiano, M.; Du Bois, B. & Systä, T. (2007). Empirical studies in reverse engineering: State of the art and future trends. *Empirical Software Engineering*, 12(5):551–571.

Tvedt, R.; Lindvall, M. & Costa, P. (2002). A process for software architecture evaluation using metrics. Em *27th Annual NASA Goddard/IEEE - Software Engineering Workshop*, pp. 191–196.

Viana, M.; Valente, M. T.; Barbosa, G.; Hora, A. & Moraes, E. (2015). Visualizing javascript source code. Em *3rd Workshop on Software Visualization, Evolution, and Maintenance (VEM)*, pp. 73–80.

Wettel, R. & Lanza, M. (2008). Codecity: 3d visualization of large-scale software. Em *Companion of the 30th International Conference on Software Engineering*, pp. 921–922.

Zapalowski, V.; Nunes, I. & Nunes, D. J. (2014). Revealing the relationship between architectural elements and source code characteristics. Em *22th International Conference on Program Comprehension (ICPC)*, pp. 14–25.

Apêndice A

Especificações DCL 2.0

Para avaliar a linguagem proposta neste trabalho, o Capítulo 5 apresenta um experimento, onde um sistema real teve sua arquitetura especificada na linguagem DCL 2.0. Baseado no conceito de camadas, o sistema é composto de seis camadas arquiteturais, onde cada uma delas tem sua responsabilidade definida de acordo com as convenções arquiteturais. Neste capítulo são apresentadas em detalhes as especificações DCL 2.0 para cada uma das seis camadas.

A.1 Camada Comum

Como o próprio nome diz, a camada comum tem como responsabilidade conter todos os artefatos que serão de uso geral da aplicação. Por exemplo, classes utilitárias e constantes. Por esta razão, o nível de acoplamento dessa camada dever ser baixo, pois qualquer aumento no acoplamento é propagado para todas as demais camadas, uma vez que todas outras camadas são dependentes dessa. Pode-se observar na especificação que são poucos os pacotes declarados nessa camada.

```
1  architecture comum{
2    ecosistema{matching: "xx.yyy.zzzzzzz.{?}";
3      sistema{matching: "{?}";
4        comuns{matching: "comuns";
5          constantes{
6            matching: "constantes";
7            Constante{
8              matching: "Constantes{?}";
9              restrictions{
10               can declare only platform.java.lang;
11             }
12           }
```

```
13          }
14          utils{matching: "utils";
15            Util{matching: "{?}Util";
16              restrictions{
17                can declare only platform.java.lang,
18                platform.java.util,
19                platform.java.text,
20                platform.wwwwwwww.comuns.utils;
21              }
22            }
23          }
24        }
25      }
26    }
27    ignore "xx", "yyy", "zzzzzzz", "java", ".classpath",
28    ".jazzignore", "main", "test", "classes", "resources",
29    ".project", ".settings*", "bin*", "target*", "bin",
30    "pom.xml", "src";
31  }
```

A.2 Camada Domínio

Os componentes dessa camada são utilizados para transporte de dados, persistência e enumerações. Os componentes *VOs* são utilizados para garantir a rastreabilidade entre código e os conceitos essenciais do sistema, uma vez que eles representam os principais conceitos do domínio.

```
1  architecture dominio{
2      ecosistema{matching: "xx.yyy.zzzzzzz.{?}";
3          sistema{matching: "{?}";
4              entidades{matching: "entidades";
5                  modulo{matching: "{?}";
6                      VO{matching: "{?}VO";
7                          restrictions{
8                                  must implement platform.java.io.Serializable;
9                                  must extend BaseVO message "";
10                             }
11                         }
12                         AudVO{matching: "{?}AudVO"; description: "";
13                             restrictions{
14                                 must extend
15                                 platform.wwwwwwww.comuns.auditoria.ProAuditoriaVO;
16                             }
17                         }
18                     restrictions{
19                     requires VO,AudVO;
```

```
20                      }
21                  }
22                  BaseVO{matching: "{?}BaseVO"; description: "";
23                      restrictions{
24                          must extend platform.wwwwwwww.entidades.ProVO;
25                      }
26                  }
27                  restrictions{
28                      can declare only enums .**, entidades .**,
29                      comum.ecosistema.sistema.comuns .**,
30                      platform.wwwwwwww.comuns .**,
31                      platform.wwwwwwww.entidades .**,
32                      platform.org.hibernate.annotation,
33                      platform.org.hibernate.envers,
34                      platform.org.hibernate.envers.RevisionType,
35                      platform.java.lang,
36                      platform.java.util, platform.java.text,
37                      platform.javax.persistence, platform.ssc-interface .**;
38                  }
39              }
40              enums{matching: "enums";
41              modulo{matching: "{?}";
42                  Enum{matching: "{?}Enum";}
43                      restrictions{
44                          requires entidades.modulo.VO;
45                      }
46                  }
47              }
48              restrictions{
49                  requires entidades.BaseVO;
50              }
51          }
52      }
53  ignore "xx", "yyy", "zzzzzzz", "java", ".classpath",
54  ".jazzignore", "main", "test",
55  "classes", "resources*", ".project", ".settings*",
56  "bin*", "target*", "bin", "pom.xml", "src";
57  }
```

A.3 Camada Interface de Negócio

A camada de interface de negócio tem a função de criar um contrato entre as camadas *Web*, *mobile* e legado de um lado, e camada de negócio de outro. A camada é subdividida em três módulos: interfaces, exceções e comuns. Esta também é uma camada bastante restrita, suas dependências são mínimas, limitando-se a algumas anotações da plataforma JEE.

```
architecture negocio-interface{
  ecosistema{matching: "xx.yyy.zzzzzzz.{?}";
    sistema{matching: "{?}";
      interfaces{matching: "interfaces";
        modulo{
          matching: "{?}";
          IFacade{matching: "I{?}Facade";
            restrictions{
              requires dominio.ecosistema.sistema.entidades.modulo.VO;
            }}
          IFacadeWS{
            matching: "I{?}FacadeWS";
            restrictions{
              requires dominio.ecosistema.sistema.entidades.modulo.VO;
            }}
          restrictions{
            requires dominio.ecosistema.sistema.entidades.modulo.VO;
          }}
        IBaseFacade{
          matching: "I{?}BaseFacade";
          restrictions{
            requires dominio.ecosistema.sistema.entidades.BaseVO;
          }}
        IBaseFacadeWS{matching: "I{?}BaseFacadeWS";
          restrictions{
            requires dominio.ecosistema.sistema.entidades.BaseVO;
          }}}
      comuns{matching: "comuns";
        modulo{}}
      excecoes{
        matching: "excecoes";
        modulo-excecoes{matching: "{?}";
          Excecao{matching: "{?}Exception";
          }
          restrictions{
            requires dominio.ecosistema.sistema.entidades.modulo.VO;
          }}
        BaseExcecao{matching: "{?}BaseException";
          restrictions{
            requires dominio.ecosistema.sistema.entidades.BaseVO;
          }}}
      restrictions{
        can declare only platform.org.hibernate.annotation,
        platform.java.lang, platform.java.util, platform.java.text,
        platform.javax.persistence, platform.javax.ejb.Remote,
        platform.javax.ejb, platform.javax.jws,platform.javax.ws,
        platform.javax.xml.bind.annotation,
        platform.wwwwwwww.entidades .**,
        platform.wwwwwwww.comuns.**, dominio.ecosistema.sistema.entidades .**,
        platform.ssc-interface .**;}}}
  ignore "java", ".classpath",".jazzignore", "main", "test", "classes", "
    resources", ".project", ".settings*", "bin*", "target*", "bin", "pom.xml", "
    src";
}
```

A.4 Camada Negócio

A camada de negócio inclui componentes responsáveis por implementar padrões de projeto. A sua responsabilidade principal é executar regras de negócios específicas via padrão de projeto *Especification* [Evans, 2004]. Outro padrão de projeto contido nesta camada é o padrão conhecido como *SessionFacade*, muito comum em arquiteturas JEE . A camada de negócio tem acesso às camadas infra-estrutura e domínio e deve implementar *interfaces* da camada *negocio-interface*.

```
1  architecture negocio{
2    ecosistema{
3      matching: "xx.yyy.zzzzzzz.{?}";
4      sistema{
5        matching: "{?}";
6        negocio{
7          matching: "negocio";
8          modulo{
9            matching: "{?}";
10           IFacadeImpl{
11             matching: "I{?}FacadeImp";
12             restrictions{
13               requires dominio.ecosistema.sistema.entidades.modulo.VO;
14               can declare only platform.wwwwwwww,
15               platform.org.hibernate.annotation,
16               platform.java.lang,
17               platform.java.util,
18               platform.java.text,
19               platform.javax.persistence,
20               dominio.ecosistema.sistema.entidades,
21               platform.ssc-interface;
22             }
23           }
24           Especification{
25             matching: "{?}RN";
26             restrictions{
27               requires dominio.ecosistema.sistema.entidades.modulo.VO;
28               can declare only Especification,
29               BaseEspecification,
30               dominio.ecosistema.sistema.entidades.**,
31               dominio.ecosistema.sistema.enums.**,
32               comum.ecosistema.sistema.comuns.**,
33               negocio-interface.ecosistema.sistema.excecoes.**,
34               infraestrutura.ecosistema.sistema.**,
35               platform.ssc-interface.enumerations.**,
36               platform.ssc-interface.interfaces.**,
37               platform.ssc-interface.interfaces.dto.**,
38               platform.wwwwwwww.negocio.**,
39               platform.wwwwwwww.comum.**,
40               platform.wwwwwwww.entidades.**,
41               platform.wwwwwwww.comuns.**,
42               platform.wwwwwwww.persistencia.**,
```

```
43                platform.wwwwwwww.criptografia.utils,
44                platform.wwwwwwww.json,
45                platform.org.hibernate.annotation,
46                platform.org.slf4j,
47                platform.org.apache.commons,
48                platform.org.apache.commons.lang,
49                platform.org.apache.commons.collections,
50                platform.org.apache.commons.beanutils,
51                platform.org.joda.time.format,
52                platform.org.joda.time,
53                platform.java.io,
54                platform.java.io.Serializable,
55                platform.java.lang,
56                platform.java.util,
57                platform.java.text,
58                platform.javax.persistence,
59                platform.javax.jws,
60                platform.javax.xml.bind.annotation;
61              }
62            }
63            EspecificationHelper{
64              matching: "{?}HelperRN";
65              restrictions{
66                requires dominio.ecosistema.sistema.entidades.modulo.VO;
67                can declare only platform.org.hibernate.annotation,
68                platform.java.lang platform.java.util,
69                platform.java.text,
70                platform.javax.persistence,
71                platform.javax.jws,
72                dominio.ecosistema.sistema.entidades,
73                platform.javax.xml.bind.annotation,
74                platform.ssc-interface;
75              }
76            }
77          }
78          BaseEspecification{
79            matching: "{?}BaseRN";
80          }
81          IBaseFacadeImpl{
82            matching: "{?}BaseFacadeImp";
83            restrictions{
84              requires dominio.ecosistema.sistema.entidades.BaseVO;
85              can declare only negocio.**,
86              dominio.ecosistema.sistema.entidades.**,
87              negocio-interface.ecosistema.**,
88              platform.ssc-interface.**,
89              platform.wwwwwwww.**,
90              platform.org.hibernate.annotation,
91              platform.java.lang platform.java.util,
92              platform.java.text,
93              platform.javax.persistence,
94              platform.javax.ejb,
95              platform.org.slf4j,
96              platform.org.apache.commons.lang,
```

```
97              platform.ssc-interface;
98            }
99          }
100       }
101       rest{
102         matching: "rest";
103         modulo{
104           matching: "{?}";
105         }
106         RestActivator{
107           matching: "{?}RestActivator";
108         }
109         RestFacadeImp{
110           matching: "{?}RestFacadeImp";
111         }
112       }
113       soap{
114         matching: "soap";
115         modulo{
116           matching: "{?}";
117           WSFacadeImp{
118             matching: "{?}WSFacadeImp";
119           }
120         }
121         BaseWSFacadeImp{
122           matching: "{?}BaseWSFacadeImp";
123         }
124       }
125       schedule{
126         matching: "schedule";
127         modulo{
128           matching: "{?}";
129           Schedule{
130             matching: "{?}Schedule";
131             restrictions{
132               requires dominio.ecosistema.sistema.entidades.modulo.VO;
133             }
134           }
135           restrictions{
136             requires dominio.ecosistema.sistema.entidades.modulo.VO;
137           }
138         }
139         BaseSchedule{
140                   matching: "{?}BaseSchedule";
141         }
142       }
143     }
144     restrictions{
145       cannot declare platform.java.lang.reflect;
146     }
147   }
148   ignore "xx", "yyy", "zzzzzzz", "java", ".classpath", ".jazzignore",
149   "bin*", "target*", "bin", "pom.xml", "src";
150 }
```

A.5 Camada Infraestrutura

A camada de infraestrutura é responsável por fornecer vários serviços de persistência e acesso a dados por meio de componentes que implementam o padrão *DAO*. Por esta razão, esta camada é acoplada a *frameworks* específicos para lidar com persistência, tais como, *Hibernate* e *JPA*. O padrão de projeto *DTO* também está presente nesta camada e sua responsabilidade é prover objetos de transporte de dados dentro da camada de infraestrutura.

```
1  architecture infraestrutura{
2    ecosistema{
3      matching: "xx.yyy.zzzzzzz.{?}";
4      sistema{
5        matching: "{?}";
6        persistencia{
7          matching: "persistencia";
8          modulo{
9            matching: "{?}";
10           DAO{
11             matching: "{?}DAO";
12             restrictions{
13               must extend BaseDAO;
14               requires dominio.ecosistema.sistema.entidades.modulo.VO;
15             }
16           }
17           AudDAO{
18             matching: "{?}AudDAO";
19             restrictions{
20               requires dominio.ecosistema.sistema.entidades.modulo.AudVO;
21             }
22           }
23           DAOHelper{
24             matching: "{?}DAOHelper";
25             restrictions{
26               requires DAO;
27             }
28           }
29         }
30         BaseDAO{
31           matching: "{?}BaseDAO";
32           restrictions{
33             must extend platform.wwwwwwww.persistencia.ProBaseDAO;
34             requires dominio.ecosistema.sistema.entidades.BaseVO;
35           }
36         }
37       }
38       entidades{
39         matching: "entidades";
40         modulo{
41           matching: "{?}";
42           DTO{
```

```
43              matching: "{?}DTO";
44            }
45            restrictions{
46              requires dominio.ecosistema.sistema.entidades.modulo.VO;
47            }
48          }
49        }
50        restrictions{
51          can declare only entidades .**,
52          persistencia .**,
53          dominio.ecosistema .**,
54          comum.ecosistema.sistema .**,
55          negocio-interface.ecosistema.sistema .**,
56          platform.ssc-interface .**,
57          platform.wwwwwwww .**,
58          platform.org.hibernate .**,
59          platform.org.springframework.ldap .**,
60          platform.org.springframework.context .**,
61          platform.org.springframework.context.support .**,
62          platform.org.apache.commons.lang,
63          platform.org.apache.commons.collections,
64          platform.org.slf4j,
65          platform.java.io,
66          platform.java.io.Serializable,
67          platform.java.util,
68          platform.java.lang,
69          platform.javax.persistence,
70          platform.javax.mail,
71          platform.javax.mail.internet,
72          platform.javax.naming,
73          platform.javax.naming.directory;
74        }
75      }
76    }
77    ignore "xx", "yyy", "zzzzzzz", "java", ".classpath", ".jazzignore", "main";
78  }
```

A.6 Camada Web

A camada web possui a responsabilidade de responder a eventos acionados pelo usuário. Basicamente, camada funciona da seguinte forma. (i) a camada fornece a interface de usuário no formato *HTML*; (ii) recebe as informação que vêm do usuário por meio do componente *Page*; (iii) transforma essas informações em objetos de domínio (*VOs*) utilizando o componente *Binder*; (iv) envia a informação para as camadas inferiores; (v) e exibe a resposta ao usuário por meio do componente *MessageHelper*. Esse conjunto de operações e a colaboração entre objetos é orquestrado pelo componente *Controller*.

```
architecture web{
  ecosistema{
    matching: "xx.yyy.zzzzzzz.{?}";
    sistema{
      matching: "{?}";
      listener{
        matching: "listener";
        AdminBaseListener{
          matching: "{?}AdminBaseListener";
        }
      }
      controle{
        matching: "controle";
        modulo{
          matching: "{?}";
          Controller{
            matching: "{?}Ctr";
            restrictions{
              requires dominio.ecosistema.sistema.entidades.modulo.VO;
            }
          }
        }
        BaseController{
          matching: "{?}BaseCtr";
          restrictions{
            requires dominio.ecosistema.sistema.entidades.BaseVO;
          }
        }
        WindowIntroducao{
          matching: "{?}WindowIntroducao";
        }
        WindowIndex{
          matching: "{?}WindowIndex";
        }
        restrictions{
          can declare only platform.org.hibernate.annotation,
          platform.java.lang,
          platform.java.util,
          platform.java.text,
          platform.javax.persistence,
          platform.javax.naming,
          platform.javax.servlet.**, //Rever
          platform.org.zkoss.zk.**,
          platform.org.zkoss.zul.**,
          platform.org.zkoss.image.**,
          platform.org.zkoss.zkplus.databind.BindingListModelSet,
          platform.org.apache,
          platform.org.joda,
          platform.org.hibernate.envers.RevisionType,
          platform.org.slf4j,
          platform.ssc-interface.**,
          platform.wwwwwww.entidades.**,
          platform.wwwwwww.criptografia.utils.**,
          platform.wwwwwww.json.**,
```

```
55            platform.wwwwwwww.controle .**,
56            platform.wwwwwwww.controle.componente .**,
57            platform.wwwwwwww.comuns .**,
58            platform.wwwwwwww.comuns.utils .**,
59            platform.wwwwwwww.visao.visao-heper .**,
60            negocio-interface.ecosistema.sistema.interfaces .**,
61            dominio.ecosistema.sistema.enums .**,
62            dominio.ecosistema.sistema.entidades .**,
63            comum.ecosistema.sistema.comuns .**;
64          }
65        }
66        helper{
67          matching: "helper";
68          modulo{
69            matching: "{?}";
70            HelperView{
71              matching: "{?}HelperView";
72              restrictions{
73                requires controle.modulo.Controller;
74              }
75            }
76            MessageHelper{
77              matching: "{?}MessageHelper";
78              restrictions{
79                requires controle.modulo.Controller;
80              }
81            }
82            BinderHelper{
83              matching: "{?}BinderHelper";
84              restrictions{
85                requires controle.modulo.Controller;
86              }
87            }
88          }
89          BaseHelperView{
90            matching: "{?}BaseHelperView";
91            restrictions{
92              requires controle.BaseController;
93            }
94          }
95        }
96        visao{
97          matching: "visao";
98          modulo{
99            matching: "{?}";
100           Page{
101             matching: "{?}{extension=zul}";
102             restrictions{
103               requires controle.modulo.Controller;
104             }
105           }
106         }
107       }
108       js{
```

```
109           matching: "js";
110           modulo{
111             matching: "{?}";
112             Script{
113               matching: "{?}{extension=js}";
114             }
115           }
116         }
117         css{
118           matching: "css";
119           modulo{
120             matching: "{?}";
121             Style{
122               matching: "{?}{extension=css}";
123             }
124           }
125         } test-funcionais{
126           matching: "test";
127           modulo{
128             matching: "{?}";
129             FunctionalTest{
130               matching: "{?}FunctionalTest";
131               restrictions{
132                 requires controle.modulo.Controller;
133               }
134             }
135           }
136           BaseFunctionalTest{
137             matching: "{?}BaseFunctionalTest";
138             restrictions{
139               requires controle.BaseController;
140             }
141           }
142           restrictions{
143             can declare only funtional-test.wwwwwwww.test,
144             funtional-test.com.thoughtworks.**,
145             funtional-test.org.openqa.selenium.**,
146             platform.java.lang;
147           }
148         }
149       }
150     }
151     ignore "xx", "yyy", "zzzzzzz", "java", ".classpath",
152     ".jazzignore", "main", "classes", "resources*",
153     ".project", "webapp*", "versao.txt",
154     "mensagem.properties", ".settings*",
155     "bin*", "target*", "bin", "pom.xml", "src",
156     "imagens*", "WEB-INF*", "META-INF*";
157   }
```

A.7 Plataforma de Reúso

Como relatado no Capítulo 3, DCL 2.0 permite especificar tanto componentes internos como componentes externos ao sistema. Uma das principais vantagens dessa funcionalidade é permitir o reúso entre especificações DCL 2.0. No experimento, uma especificação foi criada para conter componentes relacionados a *frameworks*, *APIs*, utilitários e plataforma *Java*. Após a especificação dessa camada, a equipe de arquitetos percebeu a necessidade de dividir a especificação em mais partes devido a quantidade de componentes envolvidos. No entanto, essa atividade foi planejada para outro projeto.

```
1  architecture platform{
2    ssc-interface{
3      matching: "xx.yyy.zzzzzzz.ssc.*";
4      interfaces{
5        matching: "interfaces.*";
6        dto{
7          matching: "dto.*";
8        }
9        base{
10         matching: "base.*";
11       }
12     }
13     enumerations{
14       matching: "enumerations.*";
15     }
16     comuns{
17       matching: "comuns.*";
18       constantes{
19         matching: "constantes.*";
20       }
21     }
22   }
23   org{
24     matching: "org.*";
25     fest{
26             matching: "fest.*";
27       assertions{
28         matching: "assertions.*";
29       }
30     }
31     zkoss{
32       matching: "zkoss.*";
33       zk{
34         matching: "zk.*";
35         ui{
36           matching: "ui.*";
37           util{
38             matching: "util.*";
39           }
```

```
40            event{
41              matching: "event.*";
42            }
43          }
44        }
45        zul{
46          matching: "zul.*";
47          ext{
48            matching: "ext.*";
49          }
50        }
51        zkplus{
52          matching: "zkplus.*";
53          databind{
54            matching: "databind.*";
55            BindingListModelSet{
56              matching: "BindingListModelSet";
57            }
58          }
59        }
60        image{
61          matching: "image.*";
62        }
63        util{
64          matching: "util.*";
65          media{
66            matching: "media.*";
67          }
68        }
69      }
70      joda{
71        matching: "joda.*";
72        time{
73          matching: "time.*";
74          format{
75            matching: "format.*";
76          }
77      }
78      slf4j{
79        matching: "slf4j.*";
80      }
81      springframework{
82        matching: "springframework.*";
83        context{
84          matching: "context.*"; //org.springframework.context.support
85          support{
86            matching: "support.*";
87          }
88        }
89        ldap{
90          matching: "ldap.*";
91          filter{
92            matching: "filter.*";
93          }
```

```
94          core{
95            matching: "core.*";
96            support{
97              matching: "support.*";
98            }
99          }
100         odm{
101           matching: "odm.*";
102           core{
103             matching: "core.*";
104           }
105           annotations{
106             matching: "annotations.*";
107           }
108         }
109       }
110     }
111     apache{
112       matching: "apache.*";
113       commons{
114         matching: "commons.*";
115         collections{
116           matching: "collections.*";
117         }
118         lang{
119           matching: "lang.*";
120         }
121         beanutils{
122           matching: "beanutils.*";
123         }
124       }
125     }
126     hibernate{
127       matching: "org.hibernate.*";
128       transform{
129         matching: "transform.*";
130       }
131       criterion{
132                 matching: "criterion.*";
133       }
134       annotation{
135         matching: "annotations.*";
136       }
137       core{
138         matching: "org.hibernate.*";
139       }
140       envers{
141         matching: "envers.*";
142         query{
143           matching: "query.*";
144           property{
145             matching: "property.*";
146           }
147           criteria{
```

```
148             matching: "criteria.*";
149           }
150         }
151         RevisionType{
152           matching: "RevisionType";
153         }
154         entities{
155           matching: "entities.*";
156           mapper{
157             matching: "mapper.*";
158             relation{
159               matching: "relation.*";
160               query{
161                 matching: "query.*";
162               }
163             }
164           }
165         }
166       }
167       entities{
168         matching: "entities.*";
169         mapper{
170           matching: "mapper.*";
171           relation{
172             matching: "relation.*";
173             query{
174               matching: "query.*";
175             }
176           }
177         }
178       }
179     }
180     jboss{
181       matching: "jboss.*";
182       security{
183         matching: "security.*";
184         annotation{
185           matching: "annotation.*";
186         }
187       }
188       ws{
189         matching: "ws.*";
190         api{
191           matching: "api.*";
192           annotation{
193             matching: "annotation.*";
194           }
195         }
196       }
197     }
198     junit{
199       matching: "junit.*";
200       runner{
201         matching: "runner.*";
```

```
202         }
203       }
204     }
205     javax{
206       matching: "javax.*";
207       annotation{
208         matching: "annotation.*";
209         security{
210           matching: "security.*";
211         }
212       }
213       ejb{
214         matching: "ejb.*";
215         Remote{
216           matching: "Remote";
217         }
218       }
219       servlet{
220         matching: "servlet.*";
221         http{
222           matching: "http.*";
223         }
224         annotation{
225           matching: "annotation.*";
226         }
227       }
228       naming{
229         matching: "naming.*";
230         directory{
231           matching: "directory.*";
232         }
233       }
234       persistence{
235         matching: "persistence.*";
236       }
237       jws{
238         matching: "jws.*";
239       }
240       ws{
241         matching: "ws.*";
242         rest{
243           matching: "rs.*";
244           core{
245             matching: "core.*";
246           }
247         }
248       }
249       xml{
250         matching: "xml.*";
251         bind{
252           matching: "bind.*";
253           annotation{
254             matching: "annotation.*";
255           }
```

```
256         }
257         ws{
258           matching: "ws.*";
259           handler{
260             matching: "handler.*";
261           }
262         }
263       }
264       imageio{
265         matching: "imageio.*";
266         stream{
267           matching: "stream.*";
268         }
269       }
270       mail{
271         matching: "mail.*";
272         internet{
273           matching: "internet.*";
274         }
275       }
276       accessibility{
277         matching: "accessibility.*";
278       }
279     }
280     java{
281       matching: "java.*";
282       awt{
283         matching: "awt.*";
284         image{
285           matching: "image.*";
286         }
287       }
288       io{
289         matching: "io.*";
290         Serializable{
291           matching: "Serializable";
292         }
293       }
294       lang{
295         matching: "lang.*";
296         Cloneable{
297           matching: "Cloneable";
298         }
299         reflect{
300           matching: "reflect.*";
301         }
302       }
303       sql{
304         matching: "sql.*";
305       }
306       util{
307         matching: "util.*";
308         concurrent{
309           matching: "concurrent.*";
```

```
310         }
311       }
312       text{
313         matching: "text.*";
314       }
315       net{
316         matching: "net.*";
317       }
318       security{
319         matching: "security.*";
320       }
321       math{
322         matching: "math.*";
323       }
324       applet{
325         matching: "applet.*";
326       }
327     }
328     wwwwwwww{
329       matching: "br.yyy.wwwwwwww.*";
330       hibernate{
331         matching: "hibernate.*";
332       }
333       entidades{
334         matching: "entidades.*";
335         ProVO{
336           matching: "ProVO";
337         }
338         ProBaseVO{
339           matching: "ProBaseVO";
340         }
341       }
342       comum{
343         matching: "comum.*";
344       }
345       comuns{
346         matching: "comuns.*";
347         utils{
348           matching: "utils.*";
349           imagem{
350             matching: "imagem.*";
351           }
352         }
353         constantes{
354           matching: "constantes.*";
355         }
356         auditoria{
357           matching: "auditoria.*";
358           ProAuditoriaVO{
359             matching: "ProAuditoriaVO";
360           }
361         }
362         anotacoes{
363           matching: "anotacoes.*";
```

```
364           enumeracao{
365             matching: "enumeracao.*";
366           }
367         }
368         exception{
369           matching: "exception.*";
370         }
371         excecoes{
372           matching: "excecoes.*";
373         }
374         validacoes{
375           matching: "validacoes.*";
376         }
377         IProFacade{
378           matching: "IProFacade";
379         }
380       }
381       visao{
382         matching: "visao.*";
383         visao-heper{
384           matching: "helper.*";
385         }
386       }
387       controle{
388         matching: "controle.*";
389         ProCtr{
390           matching: "ProCtr";
391         }
392         componente{
393           matching: "componente.*";
394         }
395       }
396       negocio{
397         matching: "negocio.*";
398         ProBaseRN{
399           matching: "ProBaseRN";
400         }
401       }
402       persistencia{
403         matching: "persistencia.*";
404         ProBaseDAO{
405           matching: "ProBaseDAO";
406         }
407       }
408       json{
409         matching: "json.*";
410       }
411       criptografia{
412         matching: "criptografia.*";
413         utils{
414           matching: "utils.*";
415         }
416       }
417       test{
```

```
418         matching: "test.*";
419       }
420     }
421     netscape{
422       matching: "netscape.*";
423       javascript{
424         matching: "javascript.*";
425       }
426     }
427   }
```

A.8 Plataforma de Reúso - Testes de Integração

Nesta seção, apresenta-se a especificação para os componentes externos correspondentes a *frameworks* e APIs relacionados a testes de integração. O testes de integração são utilizados para testar a interface de negócio do sistema. De maneira geral, essa camada é responsável por verificar se todas as regras de negócio estão funcionando adequadamente.

```
1   architecture integration-test{
2     org{
3       matching: "org.*";
4       jboss{
5         matching: "jboss.*";
6         arquillian{
7           matching: "arquillian.*";
8           container{
9             matching: "container.*";
10            test{
11              matching: "test.*";
12              api{
13                matching: "api.*";
14              }
15            }
16          }
17          junit{
18            matching: "junit.*";
19          }
20        }
21        shrinkwrap{
22          matching: "shrinkwrap.*";
23          api{
24            matching: "api.*";
25            spec{
26              matching: "spec.*";
27            }
28          }
```

```
29          }
30        }
31      }
32    }
```

A.9 Plataforma de Reúso - Testes Funcionais

Nesta seção, apresenta-se a especificação para os componentes externos correspondentes a *frameworks* e APIs relacionados a testes funcionais. O testes funcionais são utilizados para simular a ação do usuário ao utilizar o sistema. De maneira geral, essa camada é responsável por verificar se todas os comandos de interface de usuário estão funcionando adequadamente.

```
1  architecture funtional-test{
2    org{
3      matching: "org.*";
4      openqa{
5        matching: "openqa.*";
6        selenium{
7          matching: "selenium.*";
8          firefox{
9            matching: "firefox.*";
10         }
11       }
12     }
13   }
14   com{
15     matching: "com.*";
16     thoughtworks{
17       matching: "thoughtworks.*";
18       selenium{
19         matching: "selenium.*";
20         webdriven{
21           matching: "webdriven.*";
22         }
23       }
24     }
25   }
26 }
```

Apêndice B

Gramática DCL 2.0

Este apêndice apresenta a gramática completa da linguagem DCL 2.0 na notação BNF (Backus-Nahur Form) [Sudkamp, 2005]. Na versão BNF utilizada, símbolos terminais são grafados em negrito e entre aspas. Símbolos não-terminais iniciam-se com maiúsculas. Além disso, (A)* indica zero ou mais repetições de A e (A)? indica que A é opcional.

```
1  DCLModel: architecture ID_DCL "{"
2    (AbstractComponent)*
3    (((ignore STRING) (",")?)* (";"))?
4  "}";
5
6  ID_DCL: ('a'..'z' | 'A'..'Z' | '_' | '.')
7        ('a'..'z' | 'A'..'Z' | '_' | '-' | '.' | '0'..'9')*;
8
9  AbstractComponent: MetaModule;
10
11 MetaModule: ID_DCL "{"
12   (matching STRING ";")?
13   (description STRING ";")?
14   (AbstractComponent)*
15   (restrictions "{" (Restriction)* "}")?
16 "}";
17
18 Restriction: restriction (GroupClause)? (PermissionClause)?
19               RelationType (GroupClause)?
20           ((ComponentsBinRestrictionDeclaration)(",")?)*
21           (message STRING)? ";";
22
23 AbstractNameConvention: STRING;
24
25 QualifiedName: ID_DCL ('.' ID_DCL)*;
26
27 ComponentsBinRestrictionDeclaration: ([AbstractComponent|QualifiedName])
28                                      (WildCard)?;
```

```
29  GroupClause: only | only this;
30
31  PermissionClause: must |  can |  cannot;
32
33  RelationType: access | declare | handle |
34  extend | implement | create | throw | use annotation | depend | requires;
35
36  WildCard: ".*" | ".**";
```