Conformação Arquitetural Utilizando Restrições de Dependência entre Módulos

Ricardo Terra[1]
Orientador: Marco Túlio Valente[1,2]

[1]Instituto de Informática, PUC Minas

[2]Departamento de Ciência da Computação, UFMG

{terra,mtov}@dcc.ufmg.br

Abstract. *In this master dissertation we have proposed an approach that allows software architects to restrict the spectrum of dependencies that can be established between the modules of object-oriented systems. The ultimate goal is to provide means to detect structural dependencies that are indicators of architectural erosion. The proposed approach has been successfully applied to a real-world human resource management system.*

Resumo. *Nesta dissertação de mestrado foi proposta uma solução que permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidas entre os módulos de sistemas orientados por objetos. O objetivo central é prover meios para detectar dependências estruturais que são indicadores de erosão arquitetural. A solução proposta foi aplicada com sucesso em um sistema real de gerenciamento de recursos humanos.*

1. Introdução

Arquitetura de software é geralmente definida como o conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução de sistemas de software. Essas decisões incluem como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir. Apesar de sua inquestionável importância, a arquitetura documentada de um sistema – se disponível – geralmente não reflete a sua implementação atual [4]. Na prática, desvios em relação à arquitetura planejada são comuns, devido a desconhecimento por parte dos desenvolvedores, requisitos conflitantes, dificuldades técnicas etc. Ainda mais importante, tais desvios geralmente não são capturados e resolvidos, levando ao problema conhecido como erosão arquitetural [8]. Recentemente, esse problema foi considerado como um dos problemas ainda em aberto na área de arquitetura de software [1].

Nesta dissertação de mestrado, foi proposta uma abordagem para explicitar e evitar os problemas de erosão arquitetural[1]. Mais especificamente, a dissertação foi centrada na observação de que dependências inter-modulares impróprias são uma fonte importante de violações arquiteturais e, portanto, contribuem para o processo de erosão arquitetural. Por exemplo, suponha um sistema organizado estritamente nas camadas $M_i, M_{i-1}, \ldots, M_0$ (onde M_0 representa o módulo de mais baixo nível na hierarquia), de tal forma que M_i somente pode utilizar serviços providos pelo módulo M_{i-1}, $i > 0$.

[1]A dissertação completa está em: http://www.dcc.ufmg.br/~mtov/2009_terra.pdf.

Assim, o estabelecimento de qualquer dependência nesse sistema que viole essa regra está, de fato, violando sua arquitetura. Um outro exemplo seria um sistema *Web* com um módulo de controle C e um módulo P que encapsula serviços de persistência. Claramente, nesse sistema, C é o único módulo que pode manipular requisições e respostas HTTP. Do mesmo modo, P é o único módulo que pode utilizar os serviços providos por um *framework* de persistência.

Por outro lado, linguagens de programação convencionais oferecem recursos limitados para restringir dependências inter-modulares. Por exemplo, linguagens como Java e C# permitem ocultamento de informação por meio de interfaces ou modificadores de visibilidade (tais como `public`, `private` e `protected`). No entanto, esses modificadores oferecem um modelo de controle de dependência de granularidade muito grossa. Na prática, qualquer serviço público provido por um módulo M pode ser utilizado por qualquer outro módulo do sistema. Diante disso, nesta dissertação foi projetada e implementada uma linguagem de restrição de dependências inter-modulares, denominada DCL (*Dependency Constraint Language*), que permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidas em um dado sistema. Basicamente, a linguagem DCL permite definir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema. Uma vez definidas, tais restrições são verificadas por uma ferramenta de conformação integrada à plataforma Eclipse (essa ferramenta também foi desenvolvida durante o trabalho de mestrado). Em resumo, o objetivo principal da solução proposta é prover conformação arquitetural por construção, por meio de uma linguagem de restrição de dependência declarativa e estaticamente verificável.

O restante deste resumo estendido está organizado conforme descrito a seguir. A Seção 2 apresenta uma visão geral da abordagem de conformação arquitetural proposta na dissertação. A Seção 3 apresenta a linguagem DCL, que constitui o componente central dessa abordagem. A Seção 4 ilustra a aplicação de DCL em diferentes versões de um sistema real de gerenciamento de recursos humanos, desenvolvido pelo SERPRO. A Seção 5 conclui com uma avaliação crítica da abordagem proposta, incluindo uma comparação resumida com trabalhos relacionados.

2. Abordagem Proposta

A abordagem de verificação arquitetural proposta utiliza técnicas de análise estática para detectar dependências estruturais que são indicadores de erosão arquitetural. Conforme ilustrado na Figura 1, inicialmente o arquiteto de software deve definir as restrições de dependência do sistema, utilizando para isso a linguagem DCL, descrita na Seção 3. Para definição dessas restrições, o arquiteto deve se basear em informações e modelos do código fonte (tais como diagramas de classes) e em documentos e modelos arquiteturais (tais como diagramas de pacotes e componentes).

A solução proposta inclui também uma ferramenta de verificação de conformação arquitetural, chamada `dclcheck`, que detecta violações das restrições de dependência especificadas. Pelo menos dois cenários podem ser vislumbrados para aplicação dessa ferramenta: como parte do processo de compilação ou durante os *builds* noturnos de um sistema. Ao se integrar `dclcheck` ao processo de compilação, tem-se a vantagem de garantir que as restrições arquiteturais serão continuamente verificadas à medida que o sistema de software é implementado. Por exemplo, a execução da ferramenta `dclcheck`

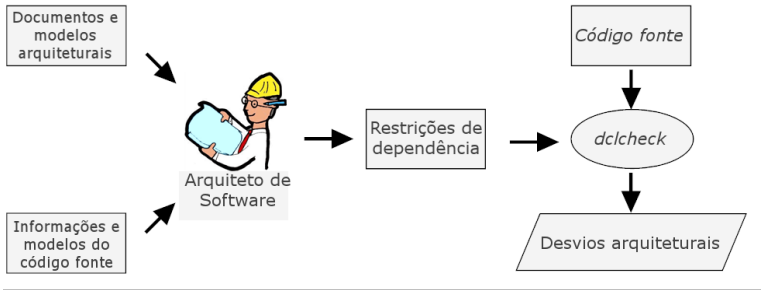

Figura 1. Abordagem proposta para conformação arquitetural

pode ser ativada nas fases iniciais da implementação ou apenas para desenvolvedores que não possuem familiaridade com a arquitetura (como é o caso de novatos incorporados ao time de desenvolvedores). Por outro lado, verificação de conformação de forma contínua pode ser intrusiva, detectando violações que foram estabelecidas somente para depuração, testes ou outros propósitos temporários. Assim, como uma segunda alternativa, pode-se executar a ferramenta `dclcheck` apenas durante os *builds* noturnos, como parte das atividades de controle de qualidade de um sistema.

A abordagem proposta detecta dois tipos de desvios arquiteturais [6]:

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

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

De forma resumida, a solução proposta possui as seguintes características: (1) ela é baseada em técnicas de análise estática, o que é importante para evitar qualquer *overhead* durante a execução dos sistemas; (2) ela é não-invasiva, uma vez que não realiza qualquer modificação no código fonte; (3) ela permite detectar violações de maneira incremental, isto é, os arquitetos podem começar aplicando DCL em uma pequena parte ou na parte mais crítica de seus sistemas; (4) ela provê conformação arquitetural por construção, isto é, violações na arquitetura planejada são detectadas logo que são implementadas no código fonte (similar ao que ocorre, por exemplo, com violações de modificadores de visibilidade, como `public`, `private` etc); (5) ela é compatível com linguagens atuais de programação orientada por objetos (embora a ferramenta `dclcheck` esteja atualmente disponível somente para a linguagem Java).

3. A Linguagem DCL

DCL é uma linguagem de domínio específico, declarativa e estaticamente verificável que permite a definição de restrições de dependência entre módulos. Assim, o objetivo principal da linguagem é restringir a organização modular de um sistema de software e não o seu comportamento. A principal contribuição da linguagem é a proposição de um modelo

de granularidade fina para especificação de dependências estruturais comuns em sistemas orientados por objetos. Esse modelo permite a definição de dependências originadas a partir do acesso a atributos e métodos, declaração de variáveis, criação de objetos, extensão de classes, implementação de interfaces, ativação de exceções e uso de anotações. Essencialmente, o modelo proposto cobre todos os relacionamentos entre classes que podem ser verificados de forma estática.

Linguagens orientadas por objetos permitem que módulos clientes referenciem quaisquer tipos públicos de outros módulos. Assim, a lógica por trás do projeto da linguagem DCL consistiu em prover meios para controlar tais dependências. Basicamente, para capturar divergências, arquitetos devem especificar que certas dependências podem (*only can* e *can-only*) ou não podem (*cannot*) ser estabelecidas entre módulos específicos. Além disso, para capturar ausências, DCL permite aos arquitetos especificar que certas dependências devem (*must*) ocorrer entre determinados módulos do sistema. Em resumo, o objetivo principal da linguagem é detectar violações estruturais que representam anomalias arquiteturais e que, portanto, contribuem para a erosão da arquitetura de um sistema.

A Figura 2 resume a sintaxe para declaração de restrições de dependência. Essas restrições e os principais elementos da linguagem DCL são descritos a seguir:

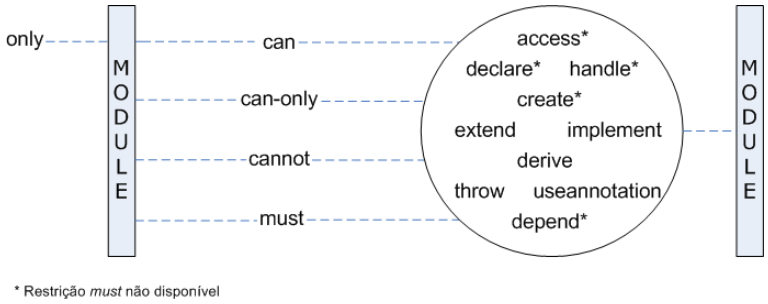

Figura 2. Restrições de dependência

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

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

O módulo `View` inclui todas as classes do pacote `org.foo.view`. O módulo `DataStructure` inclui todas as classes do pacote `org.foo.util` e a classe `Tree`. O módulo `Remote` denota todas as subclasses de `UnicastRemoteObject`. Por fim, o módulo `Frame` é definido por uma expressão regular que engloba todas as classes cujo nome qualificado inicia-se com `org.foo.` e termina com `Frame`.

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

- `only A can-x B`[2]: Somente as classes do módulo `A` podem depender dos tipos definidos no módulo `B`. Por exemplo, a restrição `only DAOFactory can-create DAO` define que somente uma classe de fábrica pode criar objetos de acesso a dados.
- `A can-only-x B`: Classes do módulo `A` somente podem depender dos tipos definidos no módulo `B`. Por exemplo, a restrição `Util can-only-depend Util, $java` define que classes utilitárias somente podem depender delas próprias ou de classes da API de Java.
- `A cannot-x B`: Classes do módulo `A` não podem depender dos tipos definidos no módulo `B`. Por exemplo, a restrição `Facade cannot-handle DTO` define que classes de fachada não podem manipular classes de entidade.

Essas restrições cobrem todas as formas de dependência típicas de linguagens orientadas por objetos, incluindo *access*, *declare*, *create*, *extend*, *implement*, *throw* e *use-annotation*. Uma vez que o objetivo dessas restrições é capturar divergências, elas definem dependências que não podem ser estabelecidas no código fonte. Restrições *only can* proíbem dependências originadas de classes não especificadas nos módulos de origem das restrições. Já restrições *can-only* proíbem dependências para classes não especificadas nos módulos de destino da restrição.

Ausências: Para capturar ausências, DCL possibilita a definição da seguinte restrição:

- `A must-x B`: Classes do módulo `A` devem depender de tipos definidos no módulo `B`. Por exemplo, a restrição `DTO must-implement java.io.Serializable` define que classes de entidade devem implementar a interface de serialização de Java.

Uma descrição detalhada das restrições disponíveis em DCL pode ser encontrada no texto completo da dissertação.

4. Estudo de Caso

A abordagem proposta para conformação arquitetural foi aplicada em um sistema de gerenciamento de recursos humanos de grande porte, chamado SGP (Sistema de Gestão de Pessoas), que atualmente é utilizado pelo SERPRO para gestão de seus mais de 12 mil empregados. A arquitetura do sistema SGP segue o padrão arquitetural MVC, conforme ilustrado na Figura 3. A camada de Modelo contém Objetos de Negócio (BOs), Objetos de Transferência de Dados (DTOs) e Objetos de Acesso a Dados (DAOs). BOs encapsulam regras de negócio e comportamentos. DTOs representam entidades de domínio, tais como empregados, planos de cargo, departamentos etc. DAOs proveem uma interface para acesso ao *framework* de persistência subjacente. Particularmente, na implementação do sistema SGP utiliza-se o *framework* Hibernate.

Metodologia: Foram consideradas três versões do sistema SGP, em diferentes estágios do seu desenvolvimento, conforme descrito na Tabela 1. A última versão escolhida possui mais de 240 KLOC e 2.300 classes e interfaces. A fim de avaliar a solução proposta, as seguintes atividades foram realizadas sobre cada uma das versões selecionadas:

[2]O literal *x* se refere ao tipo da dependência, que pode ser mais abrangente (`depend`) ou mais específico (`access`, `declare`, `create`, `extend`, `implement`, `throw` e `useannotation`).

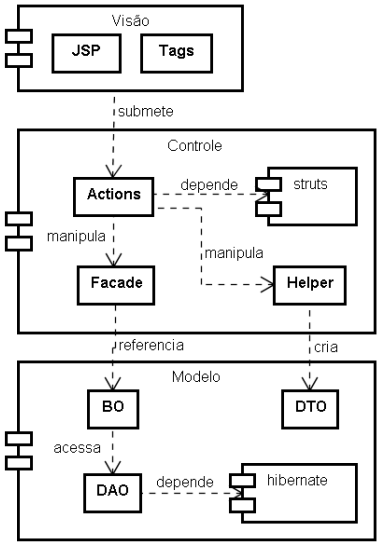

Figura 3. Arquitetura do Sistema SGP

Tabela 1. Versões analisadas no estudo de caso

	1ª versão	2ª versão	3ª versão
Data	Junho, 2006	Julho, 2007	Abril, 2008
LOC	18.062	181.306	239.589
Pacotes	26	49	83
Classes/Interfaces	308	1.923	2.329
Bibliotecas externas (JARs)	32	60	68

1. Com apoio dos arquitetos de software responsáveis pelo sistema, foram definidas restrições de dependência para cada uma das versões analisadas.
2. A ferramenta `dclcheck` foi aplicada sobre cada uma das versões para detectar divergências e ausências a partir das restrições definidas no passo anterior.
3. As relações divergentes e ausentes foram reportadas aos arquitetos de software com o intuito de confirmar se elas realmente representam violações arquiteturais.

Resultados: Como pode ser observado na Tabela 2, a abordagem proposta foi capaz de detectar diversas violações arquiteturais nas três versões analisadas do sistema SGP. Além disso, verificou-se que o número de classes com violações aumentou consideravelmente ao longo das versões analisadas: onze classes com violações foram detectadas na primeira versão (isto é, 3,5% das classes do sistema), 175 classes na segunda versão (9,1%) e 245 classes na terceira versão (10,5%). Essas violações, reportadas pela ferramenta `dclcheck`, foram apresentadas aos arquitetos que confirmaram que elas realmente representam violações arquiteturais, sem exceção. Assim, os arquitetos puderam notar que, após dois anos da data de entrega de sua primeira versão estável, a arquitetura do sistema SGP encontrava-se em um processo relevante de degeneração.

Como um exemplo de desvio arquitetural, pode ser citado o serviço de criação de DAOs previsto na arquitetura do sistema SGP. Enquanto nenhuma violação dessa restrição foi encontrada na primeira versão, foram encontradas seis classes na segunda e terceira versões que criam DAOs diretamente sem a utilização desse serviço de fábrica. Como

Tabela 2. Informações sobre as restrições de dependência definidas

	1ª versão	2ª versão	3ª versão
Número de módulos	22	39	44
Número de restrições *only can*	15	20	24
Número de restrições *can only*	5	5	5
Número de restrições *cannot*	3	1	1
Número de restrições *must*	15	22	25
Total de restrições definidas	38	48	55
Número de classes com violações	11	175	245

um outro exemplo, existem duas restrições definindo que BOs somente podem manipular interfaces de DAOs e nunca as respectivas implementações. Essas restrições são importantes para desacoplar a camada de Modelo do serviço de persistência subjacente. Porém, foram encontrados nove BOs na segunda versão e dez BOs na terceira versão que acessam diretamente implementações de DAOs.

Por fim, com a orientação do arquiteto responsável, as violações detectadas foram classificadas nas seguintes categorias: violações das camadas MVC (tal como acesso à camada de Modelo diretamente da camada de Visão), uso inapropriado de padrões de persistência (na maioria das vezes BO e DAO), uso não autorizado de *frameworks* (isto é, módulos acessando *frameworks* aos quais lhes foi negado o acesso), não uso de *frameworks* (isto é, módulos não estão utilizando *frameworks* que eles deveriam utilizar), comprometimento de reúso (geralmente acoplamentos com tipos específicos do sistema) e uso inapropriado de padrões de projeto (principalmente fábricas).

5. Conclusões

Conformação arquitetural é um problema relevante na área de arquitetura de software. Nesta dissertação, foi proposta uma abordagem para lidar com esse problema baseada em uma linguagem de domínio específico. A linguagem DCL é mais poderosa e baseada em um modelo de granularidade mais fina que modificadores de visibilidade típicos de linguagens orientadas por objetos. Por outro lado, conforme mostrado na Tabela 3, DCL apresenta diferenças importantes em relação a outras linguagens e/ou soluções para restrição de dependências, como SCL [3], LogEn [2] e IntensionalViews [5]. Fundamentalmente, ela tem um foco claro em violações arquiteturais introduzidas por meio de dependências inter-modulares impróprias. Esse foco permite que restrições arquiteturais sejam definidas em uma linguagem mais simples e de fácil entendimento.

Tabela 3. Linguagens e soluções para restrições de dependência

	SCL	LogEn	IntensionalViews	DCL
Foco	Decisões de projeto	Decisões arquiteturais e de projeto	Padrões de projeto e programação	Decisões arquiteturais
Entidades	Nodos da AST	Módulos	Nodos da AST	Módulos
Linguagem	Prolog-like	Datalog	Prolog-like	Domínio específico

Além do projeto e implementação da linguagem DCL, o trabalho incluiu sua

aplicação em um ambiente real de desenvolvimento de sistemas. Foram analisadas três versões de um sistema de recursos humanos de grande porte. Na terceira versão avaliada, foram detectadas 245 classes – ou 10,5% das classes do sistema – com alguma forma de dependência estrutural que representa uma violação da arquitetura planejada do sistema.

Os primeiros resultados do trabalho foram publicados em um *short paper* [9] e em um artigo em simpósio nacional [10]. Posteriormente, os resultados finais e completos da pesquisa foram publicados em um periódico [11]. O trabalho serviu ainda de inspiração para publicação de um artigo ilustrativo sobre conformação arquitetural em uma revista [7]. A ferramenta `dclcheck` está publicamente disponível em: `http://www.dcc.ufmg.br/~terra/dcl`.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq. Durante seu desenvolvimento, o orientador da dissertação estava vinculado à PUC Minas.

Referências

[1] P. Clements and M. Shaw. The golden age of software architecture revisited. *IEEE Software*, 26(4):70–72, 2009.

[2] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini. Defining and continuous checking of structural program dependencies. In *30th International Conference on Software Engineering (ICSE)*, pages 391–400, 2008.

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

[4] J. Knodel and D. Popescu. A comparison of static architecture compliance checking approaches. In *6th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, page 12, 2007.

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

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

[7] L. Passos, R. Terra, R. Diniz, M. T. Valente, and N. das Chagas Mendonca. Static architecture conformance checking – an illustrative overview. *IEEE Software*, 2010. To appear.

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

[9] R. Terra and M. T. Valente. Towards a dependency constraint language to manage software architectures. In *Second European Conference on Software Architecture (ECSA)*, volume 5292 of *Lecture Notes in Computer Science*, pages 256–263. Springer, 2008.

[10] R. Terra and M. T. Valente. Verificação estática de arquiteturas de software utilizando restrições de dependência. In *II Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS)*, pages 1–14, 2008.

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