Definição de Padrões Arquiteturais e seu Impacto em Atividades de Manutenção de Software

Ricardo Terra, Marco Túlio Valente

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
31.270-010 – Belo Horizonte – MG

{terra,mtov}@dcc.ufmg.br

Abstract. *Developers usually rely on patterns and best practices to increase the quality of their projects. However, as projects evolves, it is usual to observe deviations in the use of the patterns and best practices defined during the initial design of a system. This article aims to show the application of DCL – a static, declarative dependency constraint language – to define architectural patterns and best practices that can contribute to the maintainability of a system.*

Resumo. *A utilização de padrões e a adoção de boas práticas são recomendações sempre realizadas em projetos de desenvolvimento de software. Contudo, no decorrer do projeto, esses padrões tendem a se degradar, fazendo com que seus benefícios sejam anulados. Assim, este artigo tem como objetivo mostrar o uso de DCL – uma linguagem declarativa e estaticamente verificável – para expressar padrões e boas práticas arquiteturais que, uma vez preservados, contribuem para a manutenibilidade de um sistema.*

1. Introdução

Arquitetura de software é geralmente definida como um conjunto de decisões de projeto que tem impacto em cada aspecto da construção e evolução de sistemas de software. Isso inclui como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir [3, 1]. A definição de uma arquitetura engloba diversos padrões e boas práticas arquiteturais. Contudo, com o decorrer do projeto, esses padrões tendem a se degradar fazendo com que os benefícios proporcionados por um projeto arquitetural (manutenibilidade, escalabilidade, portabilidade etc) sejam anulados [4].

Diante disso, o objetivo deste artigo é demonstrar que a preservação de padrões e boas práticas arquiteturais contribui para o processo de manutenção. Para isso, é utilizado um sistema motivador cuja arquitetura envolve diversos padrões e boas práticas arquiteturais comumente encontrados no desenvolvimento de sistemas. A partir desse sistema, é demonstrado como preservá-los por meio da utilização da linguagem DCL (*Dependency Constraint Language*), que permite a arquitetos de software restringir o espectro de dependências que podem ser estabelecidas em um dado sistema [6]. Mais especificamente, DCL é uma linguagem que permite definir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de um sistema. Uma vez definidas, tais restrições são automaticamente verificadas por uma ferramenta de conformação integrada à IDE Eclipse.

O restante deste artigo está organizado conforme descrito a seguir. A Seção 2 introduz a linguagem DCL. A Seção 3 descreve o sistema motivador cuja arquitetura envolve diversos padrões e boas práticas arquiteturais. Em seguida, a Seção 4 apresenta como esses padrões podem ser definidos na linguagem DCL de forma que sejam preservados e contribuam para a manutenibilidade do sistema. Por fim, a Seção 5 apresenta as considerações finais.

2. A Linguagem DCL

DCL é uma linguagem de domínio específico, declarativa e estaticamente verificável que permite a definição de restrições de dependência entre módulos [5, 6]. É possível restringir 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. 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 1 resume a sintaxe para declaração de restrições de dependência em DCL. Essas restrições e os principais elementos da linguagem DCL são descritos a seguir:

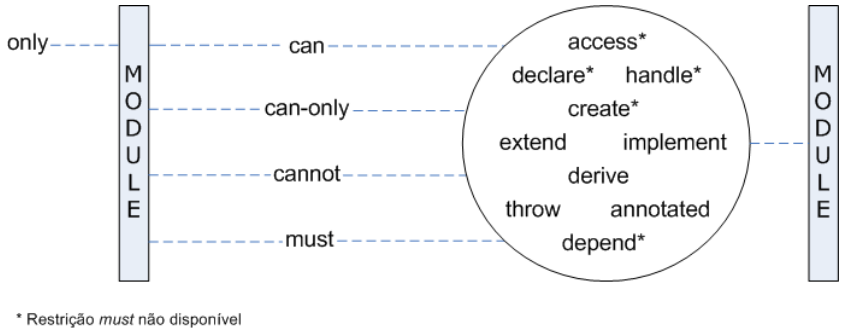

Figura 1. Restrições de dependência

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+
```

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 de seus subpacotes e também a classe `Tree`. Por fim, o módulo `Remote` denota todas as subclasses de `UnicastRemoteObject`.

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

- `only A can-`*x* `B`[1]: Somente as classes do módulo `A` podem depender dos tipos definidos no módulo `B`. Por exemplo, a restrição `only DAOFactory`

[1] 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`).

`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 em [6]. A linguagem DCL e a ferramenta `dclcheck` estão publicamente disponíveis em: `http://www.dcc.ufmg.br/~terra/dcl`.

3. Sistema Motivador

Para ilustrar padrões e boas práticas arquiteturais comumente aplicadas no desenvolvimento de sistemas Java, utiliza-se um sistema chamado `TerraMarket` que automatiza atividades comuns de uma mercearia, como cadastro de clientes, vendas, emissão de nota fiscal etc.

A arquitetura do sistema `TerraMarket` segue o padrão arquitetural *Model View Presenter* [1], conforme ilustrado na Figura 2. A camada de Modelo contém Objetos de Negócio (*Business Objects* ou BOs), Objetos de Transferência de Dados (*Data Transfer Objects* ou DTOs) e Objetos de Acesso a Dados (*Data Access Objects* ou DAOs). BOs encapsulam regras de negócio e comportamentos. DTOs representam entidades do domínio, tais como cliente, produto, venda, nota fiscal etc. DAOs proveem uma interface para acesso ao *framework* de persistência subjacente. Particularmente, na implementação desse sistema utiliza-se o *framework* Hibernate[2] para persistência objeto/relacional.

A camada de Apresentação contém classes que interceptam os eventos gerados na Visão, monitorando e adaptando entradas do usuário, manipulando o Modelo e atualizando a Visão. Nesse sistema, o *framework* Swing é utilizado pela camada de Visão

[2]http://www.hibernate.org

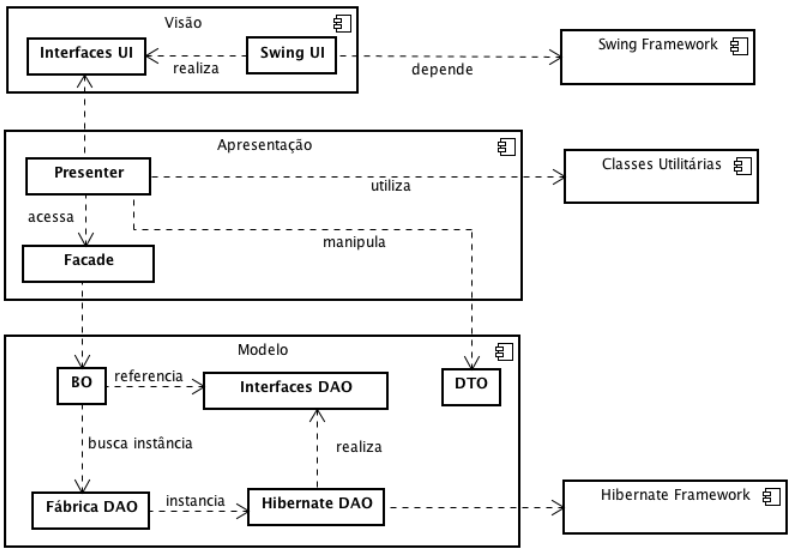

Figura 2. Arquitetura do Sistema `TerraMarket`

para a criação de janelas (*User Interface* ou UI). As requisições ativadas pela Visão são encaminhadas para um componente de Fachada, que provê um ponto de acesso único à camada de Modelo. Em resumo, a arquitetura do sistema se baseia em padrões (*MVP*, *Factory*, *Facade*, *Data Access Objects* etc) e em tecnologias (Swing, Hibernate etc) que atualmente são largamente utilizados no desenvolvimento de sistemas.

4. Padrões Arquiteturais e Manutenibilidade

Nesta seção, a arquitetura do sistema `TerraMarket` é utilizada para ilustrar diversos padrões e boas práticas arquiteturais. Inicialmente, é mostrado como esses padrões podem ser definidos em DCL[3]. Após a definição de cada padrão, são relacionados seus principais benefícios em termos de manutenibilidade.

4.1. Arquitetura em Camadas

Arquitetura em camadas provê um modelo de desenvolvimento em que componentes são organizados em camadas hierárquicas [1]. 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}, $i > 0$. Desse modo, o estabelecimento de qualquer dependência nesse sistema que viole essa regra está, de fato, violando sua arquitetura. Para especificar que o sistema `TerraMarket` adota uma arquitetura em camadas, foram definidas as seguintes restrições DCL:

```
1 view cannot-depend Model
2 only Presenter can-handle Facade
```

Na linha 1, uma restrição impede que a camada de Visão estabeleça qualquer forma de dependência com a camada de Modelo. Na linha 2, uma restrição garante o

[3]Para simplificar o entendimento das restrições DCL, serão omitidos as definições de módulos, visto que, no sistema `TerraMarket`, os módulos basicamente se referem às classes de um determinado pacote.

acesso à Fachada – único ponto de acesso à camada de Modelo – somente às classes de Apresentação.

Benefícios para a Manutenibilidade: As restrições mostradas garantem que nenhuma modificação será introduzida de forma a violar a arquitetura em camadas. Isso torna o sistema mais fácil de manter e de reutilizar. Por exemplo, no sistema `TerraMarket`, garante-se o acesso à Fachada somente aos componentes de Apresentação e impede-se que a camada de Visão estabeleça qualquer forma de dependência com a camada de Modelo, consequentemente preservando o padrão MVP.

Além disso, a aderência a essas restrições permite inferir que no caso de problemas de acesso a dados, a falha provavelmente estará localizada em uma classe de Apresentação ou na própria camada de Modelo. De forma similar, pode-se garantir que havendo a necessidade de manutenções em DAOs ou BOs, somente as classes de Apresentação poderão sofrer algum tipo de impacto. Ademais, caso a aplicação precise alterar toda a sua camada de Modelo, a camada de Visão não sofrerá impacto algum, devendo somente a Fachada ser ajustada.

4.2. Programação por Interface

Programação por Interface é um princípio de projeto que consiste em separar a interface de um componente de sua implementação. Isso cria uma camada de abstração entre o componente cliente e o componente servidor [1]. Para ilustrar esse princípio, as seguintes restrições DCL foram definidas para o sistema `TerraMarket`:

```
1 Presenter cannot-handle SwingUI
2 BO cannot-handle HibernateDAO
```

Na linha 1, uma restrição impede a camada de Apresentação de manipular diretamente as implementações UI. De forma semelhante, na linha 2, uma outra restrição impede que os objetos de negócio (BOs) manipulem diretamente as implementações dos objetos de acesso a dados (DAOs).

Benefícios para a Manutenibilidade: A preservação dessas restrições garante a separação entre componentes clientes e servidores, proporcionando a utilização de serviços sem conhecimento da implementação dos mesmos. Havendo a necessidade de uma alteração nas implementações UI ou DAO, o sistema não sofrerá impacto algum. Por exemplo, caso a interface gráfica tenha que migrar de Swing para SWT (*Standard Widget Toolkit*) ou que a implementação DAO deixe de utilizar Hibernate e passe a utilizar JDBC (*Java Database Connectivity*), isso não impactará a camada de Apresentação nem os BOs.

4.3. Padrões de Criação

O funcionamento de sistemas orientados a objetos se baseia na interação entre diversos objetos. Contudo, em certos casos, uma classe é instanciada diversas vezes mesmo sem necessidade, já que o objeto poderia ser criado uma única vez e o mesmo ser utilizado por todo o sistema [1].

Existem diversos padrões que proveem uma melhor forma de se realizar a criação única de objetos [2]. Por exemplo, o padrão *Singleton* faz com que uma classe somente

tenha uma única instância. Assim, quando se precisar da instância de uma classe *singleton*, basta invocar um método estático dessa classe cujo retorno é sua instância única. Na mesma linha, o padrão *abstract factory* consiste na existência de uma classe cujos métodos retornam instâncias de determinadas classes. Convém salientar que esses métodos usualmente possuem uma interface como tipo de retorno e o retorno é a instância (normalmente única) do objeto. Assim, toda vez que uma classe do sistema necessitar do objeto, basta solicitar à sua respectiva fábrica. Para ilustrar esses padrões, as seguintes restrições DCL foram definidas para o sistema `TerraMarket`:

```
1 only Facade can-create Facade
2 only com.tm.model.dao.DAOFactory can-create HibernateDAO
```

Na linha 1, uma restrição garante a criação de Fachadas somente a elas mesmas, de forma que o padrão *Singleton* seja preservado. Por outro lado, na linha 2, uma segunda restrição impede que qualquer classe do sistema que não seja a fábrica, crie objetos DAOs.

Benefícios para a Manutenibilidade: A preservação dessas restrições garante que a criação de objetos não ocorra irrestritamente. Define-se um local centralizado de criação de DAOs. Isso faz com que haja um local específico para qualquer manutenção relacionada a criação de objetos de acesso a dados. Por exemplo, manutenções relacionadas à forma como eles são criados, ao número máximo de instâncias permitidas, a existência de caches estarão centralizadas em uma única classe.

Por outro lado, o uso de fábrica abstratas proporciona a vantagem da abstração já citada na Seção 4.2. Um exemplo são as classes BO, as quais não podem estabelecer qualquer forma de dependência com implementações DAO. Diante disso, ao solicitarem determinada implementação DAO à fábrica, essa já será retornada como a respectiva interface. Isso faz com que classes BO se abstraiam de qual implementação utilizam e garantam o total desacoplamento com implementações DAO.

4.4. Reúso

O reúso deve ser um dos objetivos centrais de qualquer projeto de desenvolvimento de sistemas. Em essência, técnicas de reúso devem promover a criação de componentes – de forma mais genérica possível – para que possam ser reutilizados em outros projetos. Para isso, eles devem ser independentes de classes específicas do projeto. Para promover o reúso de componentes, as seguintes restrições DCL foram definidas para o sistema `TerraMarket`:

```
1 Util can-only-depend Util, ApacheUtils, $java
2 DTO can-only-depend DTO, $java
```

Na linha 1, uma restrição garante que classes utilitárias só possam estabelecer dependências entre elas mesmas, com classes da API de Java e com classes de um *framework* de classes utilitárias da Apache, comumente utilizados em projetos Java. De forma semelhante, na linha 2, uma restrição garante que objetos de transferência de dados só possam depender deles mesmos e de classes da API de Java.

Benefícios para a Manutenibilidade: A preservação dessas restrições garante que classes possam ser reutilizadas em outros projetos. Isso, em longo prazo, beneficia o processo

de manutenção, uma vez que, com a utilização de componentes reutilizáveis, tem-se uma qualidade já verificada e, portanto, menos susceptível a erros.

4.5. Controle de Dependências Externas

Sistemas reais tendem a se acoplar a diversos outros sistemas externos, como *frameworks*. Contudo, esse acoplamento deve ser controlado de forma que o *framework* seja acoplado somente às devidas classes. Por exemplo, um *framework* de persistência deve ser acoplado somente às classes de acesso a dados, isto é, devem ser impedidas dependências com quaisquer outras classes. Para ilustrar esse controle de dependência, as seguintes restrições DCL foram definidas para o sistema `TerraMarket`:

```
1 only SwingUI can-depend Swing
2 only HibernateDAO can-depend Hibernate
```

Essas restrições permitem que somente implementações UI e DAO estabeleçam qualquer forma de dependência com os *frameworks* Swing e Hibernate, respectivamente.

Benefícios para a Manutenibilidade: A definição dessas restrições garante que não serão estabelecidos acoplamentos indevidos com os *frameworks* utilizados. Esse controle de dependência proporciona que modificações de versões de *frameworks* ou, até mesmo, a alteração para um outro *framework* não impactem classes não relacionadas. Por exemplo, no sistema `TerraMarket`, caso haja alteração do *framework* de visualização ou de persistência, somente serão afetadas as implementações UI e DAO, respectivamente. Assim, percebe-se que o controle de dependências externas proporciona benefícios à manutenibilidade semelhantes àqueles proporcionados por um arquitetura em camadas, contudo voltado a utilização de classes externas.

4.6. Herança

No desenvolvimento de um sistema de software, é usual observar grupos de classes que compartilham propriedades e comportamentos comuns. Uma boa prática, normalmente adotada nessas situações, consiste na criação de uma classe base (também conhecida como classe padrão) na qual são inseridos os membros comuns. Para ilustrar essa prática, as seguintes restrições DCL foram definidas para o sistema `TerraMarket`:

```
1 Presenter must-extend com.tm.controller.presenter.BasePresenter
2 DTO must-derive com.tm.model.dto.BaseDTO, java.io.Serializable
3 IDAO must-extend com.tm.model.dao.IBaseDAO
4 HibernateDAO must-derive BaseHibernateDAO, IDAO
```

As restrições acima possuem a mesma intenção: obrigar determinado grupo de classes a herdar sua classe base. Por exemplo, as classes de Apresentação compartilham propriedades e comportamentos comuns. Por isso, na linha 1, uma restrição garante que objetos de Apresentação devem estender um classe base. O mesmo foi realizado com os objetos de transferência de dados, pois sabe-se que todos devem possuir os atributos `codigo`, `versao` e `dataAlteracao`, além de seus respectivos métodos acessores e anotações JPA (*Java Persistence API*) genéricas. Por isso, na linha 2, uma restrição garante que todos os DTOs estendam uma classe base e sejam serializáveis, uma vez que

trafegam via rede. Por fim, na linha 3, uma restrição garante que as interfaces DAO estendam uma interface base e, na linha 4, uma última restrição garante que implementações HibernateDAO estendam sua classe base e ainda implementem sua respectiva interface.

Benefícios para a Manutenibilidade: A preservação dessas restrições evita duplicidade de código, além de manter um padrão de desenvolvimento. Por exemplo, garantir que toda classe de Apresentação estenda uma classe base faz com que desenvolvedores não implementem classes de Apresentação fora do padrão ou operações que já foram implementadas. Além disso, uma alteração em todas as classes de Apresentação pode ser facilmente realizada por meio de sua classe base, pois é garantido que todas as classes de Apresentação a estendem. Por fim, a definição explícita dessas restrições evita erros triviais, como um DTO não serializável ou uma implementação DAO sem respectiva interface.

5. Considerações Finais

O uso de padrões e a adoção de boas práticas sempre foi recomendado no desenvolvimento de software. Contudo, no decorrer do projeto, esses padrões tendem a se degradar, fazendo com que seus benefícios sejam anulados. Desse modo, a tarefa de manutenção de software torna-se ainda mais árdua.

Diante disso, a partir de um sistema motivador cuja arquitetura engloba diversos padrões e boas práticas arquiteturais, foi demonstrado como utilizar DCL para expressá-los e ainda formou argumentos mostrando que a aderência a esses padrões contribui diretamente para a manutenibilidade de um sistema.

Como trabalho futuro, pretende-se desenvolver estudos de caso em sistemas reais para mensurar a melhoria proporcionada pela linguagem DCL no processo de manutenção desses sistemas. Esses novos estudos de caso poderão inclusive contribuir para agregar novas funcionalidades à linguagem DCL.

Agradecimentos: Este trabalho foi apoiado pela FAPEMIG, CAPES e CNPq.

Referências

[1] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[2] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[3] D. Garlan and M. Shaw. *Software Architecture Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[4] 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.

[5] 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*, pages 1–14, Porto Alegre, RS, Brasil, 2008.

[6] 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.