Lambda
Como melhorar seu código com funções Lambda

Conformação
Use o DCLcheck para verificar regras arquiteturais

JBoss Infinispan
Dados concorrentes em memória de rápido acesso e alta performance

MUNDOJ

NÚMERO 55 | ANO X

SETEMBRO & OUTUBRO DE 2012 | R$ 15,00

EDITORA mundo

ISSN 1679-3978

Integração com Redes Sociais

Aplicações Java Integradas dentro do Facebook
Integrando suas aplicações Android com o Twitter e Facebook

ainda: Tópicos mais quentes do GUJ.com.br | Neo4j Cloud Deployment com Spring Data
Desenvolvimento Java EE 6 com Framework Demoiselle 2

MUNDOJ

NÚMERO 55 | ANO X | SETEMBRO E OUTUBRO DE 2012
www.mundoj.com.br

artigos >

colunas >

Conformação Arquitetural com DCLcheck

Defina as dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de seu sistema

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 a forma como sistemas são estruturados em componentes e restrições sobre como tais componentes devem interagir. Apesar de sua inquestionável importância, a arquitetura documentada de um sistema – se disponível – geralmente não reflete a sua implementação atual. Na prática, desvios em relação à arquitetura planejada são comuns, devido a desconhecimento por parte dos desenvolvedores, requisitos conflitantes, dificuldades técnicas, pressões para cumprir prazos de entregas etc. Ainda mais importante, tais desvios geralmente não são capturados e resolvidos, levando ao fenômeno conhecido como erosão arquitetural.

Projetado como um plug-in para a IDE Eclipse, DCLcheck detecta tais violações arquiteturais para que esse processo não se acumule ao longo dos anos e acabe por reduzir a arquitetura do sistema a um pequeno conjunto de componentes pouco coesos e fortemente acoplados, cuja manutenção e evolução se torna cada vez mais difícil e custosa. Basicamente, uma vez que o arquiteto defina o conjunto de restrições arquiteturais, a verificação passa a acontecer de forma constante e automática. Por exemplo, se nenhuma violação arquitetural for detectada, a ferramenta se mantém invisível. Mas, logo que uma violação arquitetural for inserida, DCLcheck a reporta no mesmo local em que erros tradicionais são exibidos. Enfim, o objetivo deste artigo é mostrar como DCLcheck pode ser utilizada por desenvolvedores e arquitetos para preservar a arquitetura de seus sistemas ao longo de sua evolução.

Linguagem DCL

Dependency Constraint Language (DCL) é uma linguagem de domínio específico, declarativa e estaticamente verificável que permite a definição de restrições estruturais entre módulos. A linguagem suporta a definição de dependências originadas a partir do acesso a atributos e métodos (access), declaração de variáveis (declare), criação de objetos (create), extensão de classes (extend), implementação de interfaces (implement), lançamento de exceções (throw) e uso de anotações (useannotation). Essencialmente, o objetivo principal da linguagem DCL é indicar a presença de violações arquiteturais.

A figura 1 resume a sintaxe usada para expressar restrições arquiteturais em DCL. Essas restrições e os principais elementos da linguagem DCL são descritos a seguir.

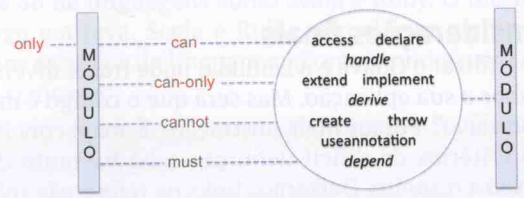

Figura 1. *Sintaxe DCL.*

Módulos

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

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

De acordo com essa definição, o módulo View contém todas as classes do pacote org.foo.view. O

Ricardo Terra | terra@dcc.ufmg.br

Doutorando em Ciência da Computação pela UFMG/UWaterloo. Trabalhou cinco anos como arquiteto Java em empresas como DBA Engenharia de Sistemas e Stefanini IT Solutions. Atualmente, atua também como professor na Universidade FUMEC.

Marco Túlio Valente | mtov@dcc.ufmg.br

Professor adjunto do Departamento de Ciência da Computação da UFMG. Seus interesses de pesquisa incluem arquitetura de software, linguagens de programação e engenharia de software. Marco Túlio possui doutorado em Ciência da Computação pela UFMG.

Luis Fernando Miranda | luisfmiranda@dcc.ufmg.br

Bacharelando em Ciência da Computação pela UFMG. Seus interesses de pesquisa incluem arquitetura de software e linguagens de programação.

A arquitetura planejada de um sistema, uma vez definida, deve ser seguida e mantida ao longo de sua evolução. No entanto, desvios em relação à arquitetura planejada são comuns, devido a desconhecimento por parte dos desenvolvedores, dificuldades técnicas etc. Neste artigo, apresenta-se a ferramenta DCLcheck – um plug-in para a IDE Eclipse – que permite a arquitetos de software restringir dependências aceitáveis e inaceitáveis de acordo com a arquitetura planejada de seus sistemas.

módulo DataStructure contém todas as classes do pacote org.foo.util, de seus subpacotes e também a classe org.foo.view.Tree. O módulo Frame inclui todas as classes cujo nome qualificado inicia-se com org.foo e termina com Frame. Por fim, o módulo Remote contém todas as subclasses de java.rmi. UnicastRemoteObject.

Divergências

Uma divergência ocorre quando uma dependência existente no código-fonte viola uma restrição arquitetural. Para capturar divergências, DCL permite a definição das seguintes restrições (onde dep se refere ao tipo de restrição):

- only MA can-dep MB: somente classes do módulo MA podem depender dos tipos definidos no módulo MB. Por exemplo, a restrição only DAOFactory can-create DAO define que somente uma classe de fábrica pode criar objetos de acesso a dados.
- MA can-dep-only MB: classes do módulo MA somente podem depender dos tipos definidos no módulo MB. 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.
- MA cannot-dep MB: classes do módulo MA não podem depender dos tipos definidos no módulo MB. 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 a objetos, incluindo access, declare, create, extend, implement, throw e useannotation. Uma vez que o objetivo dessas restrições é capturar divergências, elas alertam para dependências que não podem aparecer no código-fonte. Basicamente, restrições only can proíbem dependências originadas de classes não especificadas nos módulos de origem das regras, enquanto restrições can-only proíbem dependências de classes não especificadas nos módulos de destino das regras.

Ausências

Uma ausência ocorre quando o sistema não estabelece uma dependência prevista na arquitetura planejada. Para capturar ausências, DCL permite a definição da seguinte restrição:

- MA must-dep MB: classes do módulo MA precisam depender dos tipos definidos no módulo

MB. Por exemplo, a restrição DTO must-implement java.io.Serializable define que classes de entidade devem implementar a interface de serialização de Java.

DCLcheck

DCLcheck é um plug-in para a IDE Eclipse que reporta as violações arquiteturais detectadas no código-fonte. Tais violações são exibidas na mesma janela de erros e alertas de compilação e, mais importante, são reportadas logo que uma violação é inserida (isto é, DCL constitui uma solução para conformação arquitetural por construção). Isso força desenvolvedores a sempre seguirem a arquitetura planejada dos seus sistemas. Além disso, a ferramenta pode ser inicialmente aplicada em uma pequena parte ou na parte mais crítica dos sistemas. Em outras palavras, não é necessário a definição completa da arquitetura para utilizar DCLcheck.

A figura 2 ilustra o processo de verificação utilizando DCL. Basta os arquitetos de software especificarem um conjunto de restrições DCL para um determinado sistema-alvo que DCLcheck verifica se o código-fonte respeita tais restrições.

DCLcheck pode ser facilmente baixado pelo nosso Eclipse Update Site (http://www.dclsuite.org/update). A ferramenta permanece invisível aos usuários do Eclipse até que seja ativada. Para ativá-la, basta clicar com o botão direito sobre o nome do projeto e selecionar a opção Enable DCLcheck. Além disso, DCLcheck não requer execução do sistema (análise estática) e é completamente não invasiva, ou seja, nenhum código existente precisa ser modificado.

Quando ativado, DCLcheck cria um novo arquivo (architecture.dcl) na raiz do projeto. É nesse arquivo que o arquiteto deve definir os módulos e restrições de dependência. Como pode ser observado na figura 3, assuma que o arquiteto tenha definido uma restrição da forma only DAOFactory can-create DAO. Essa restrição indica que apenas a fábrica de Objetos de Acesso a Dados (Data Access Objects ou DAOs) pode criar tais objetos. Assuma ainda uma classe Client na qual o desenvolvedor tenha instanciado diretamente uma classe DAO. Em tal caso, DCLcheck detecta a violação arquitetural tão logo ela seja inserida.

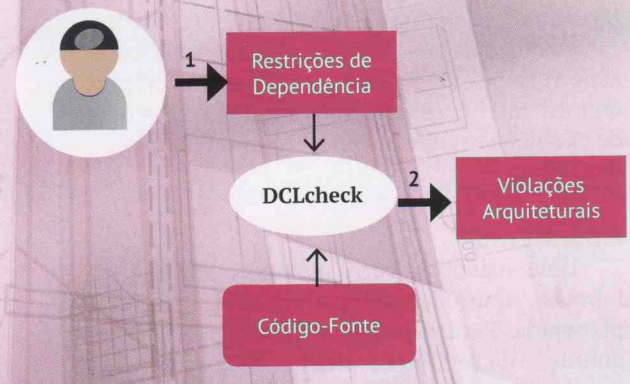

Figura 2. *Processo de verificação utilizando DCL.*

Normalmente, uma violação arquitetural deve ser analisada e corrigida pelo próprio desenvolvedor que a cometeu. Com o intuito de auxiliar tal desenvolvedor, DCLcheck provê informações adicionais sobre uma violação detectada, por exemplo, qual foi a restrição violada, o porquê da violação etc. Para isso, basta o desenvolvedor clicar com o botão direito sobre a mensagem de erro e selecionar Properties.

Aplicações práticas

DCLcheck permite definir restrições que são comumente utilizadas durante a fase de definição da arquitetura de sistemas. Nesta seção, são mostradas algumas de suas aplicações práticas, por exemplo, em arquiteturas de camadas, na programação por interface, em padrões de criação, na melhoria do reúso, no controle de dependências externas e no incentivo a herança de classes.

Arquitetura em camadas

A arquitetura em camadas provê um modelo de desenvolvimento em que componentes são organizados em níveis hierárquicos. Como um exemplo, suponha uma organização de componentes no padrão MVC (Model-View-Controller). Como pode ser observado na figura 4, a camada de Visão (View) está associada a componentes GUI (Frames, Buttons, TextField etc). A camada de Modelo (Model) está associada à persistência (DAOs, BOs etc). Por fim, a mediação entre as camadas de Modelo e Visão é feita por classes da camada de Controle (Controller). Particularmente, nesse exemplo, MVC foi implementado de tal forma que a camada de Visão não pode depender da camada de Modelo, nem o contrário.

Figura 4. *Organização de componentes no padrão MVC.*

Diante disso, as seguintes restrições DCL podem ser definidas para especificar tal implementação MVC:

```
module View:  org.foo.view.**
module Model: org.foo.model.**
View cannot-depend Model
Model cannot-depend View
```

Com os módulos de Visão e Modelo definidos (linhas 1-2), duas restrições garantem que a camada Visão

não dependa da camada de Modelo (linha 3) e vice-versa (linha 4). Em outras palavras, o acesso às camadas de Visão e Modelo terá que ser intermediado pela camada de Controle. Caso contrário, DCLcheck irá reportar uma violação arquitetural.

Além disso, a adesão a essas restrições garante que nenhuma modificação possa ser introduzida de forma a violar a organização dos componentes. Isso torna o sistema mais fácil de manter, reutilizar e adaptar. Por exemplo, nessa implementação MVC, se toda a camada de Modelo tiver que ser substituída, a camada de Visão não será impactada.

Programação por interface

Programação por interface é um princípio de projeto que consiste em separar a interface de um componente de sua implementação. Como um exemplo, objetos de acesso a dados (DAOs) contêm consultas a base de dados, como buscar clientes, atualizar empregados etc. Por outro lado, objetos de negócio (BOs) implementam regras de negócio do sistema e normalmente fazem uso de diversas consultas a bases de dados. Uma mesma regra de negócio pode ter que buscar departamentos, fazer verificações (por meio de blocos condicionais e laços de repetição) e ainda atualizar alguns empregados e clientes. A figura 5 ilustra um exemplo em que interfaces para objeto de acesso a dados (IDAO) são empregadas para desacoplar objetos de negócio (BO) das implementações de objetos de acesso a dados (HibernateDAO).

Para garantir conformidade arquitetural, podemos definir as seguintes restrições:

```
module BO: org.foo.model.bo.*
module HibernateDAO: org.foo.model.dao.hibernate.*
BO cannot-depend HibernateDAO
```

Figura 5. *Camada de abstração entre BOs e implementações DAO.*

Após definir os módulos BO e HibernateDAO (linhas 1 e 2), uma restrição impede BOs de manipular diretamente as implementações dos objetos de acesso a dados (linha 3). Caso isso ocorra, uma violação é reportada por DCLcheck.

A definição dessa restrição garante a separação entre componentes clientes e servidores, proporcionando a utilização de serviços sem conhecimento da implementação dos mesmos. Isso permite que migrações ou modificações no framework não impactem as regras de negócio. Por exemplo, caso a implementação DAO passe a utilizar JDBC (Java Database Connectivity), ao invés de Hibernate, garante-se que tal mudança não afetará os objetos de negócio.

Padrões de criação

Em sistemas orientados a objetos podem ocorrer situações em que se deseja abstrair os clientes da complexidade de criação dos objetos. Uma solução elegante para esse problema consiste na utilização do padrão Abstract Factory que consiste em uma classe cujos métodos retornam instâncias de certas classes. Como exemplo, a figura 6 ilustra uma fábrica de objetos de acesso a dados. Basicamente, quando qualquer módulo precisar de um objeto DAO, ele deve chamar um método fábrica (Factory).

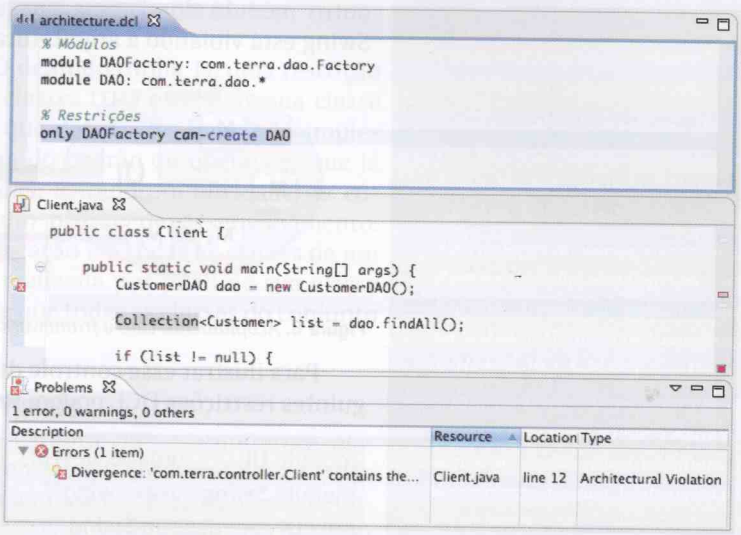

Figura 3. *Exemplo de DCLcheck reportando uma violação arquitetural.*

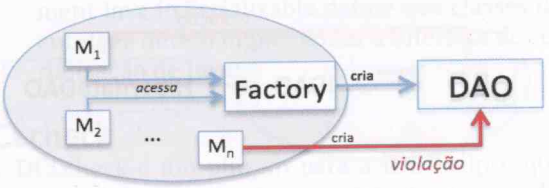

Figura 6. *Fábrica de objetos DAO.*

As seguintes restrições DCL podem ser definidas para especificar uma fábrica de objetos DAO:

```
module DAO:     org.foo.model.dao.*
module Factory: org.foo.model.dao.DAOFactory
only Factory can-create DAO
```

Com o módulo DAO e a classe Factory definidos (linhas 1 e 2), uma restrição impede que qualquer classe do sistema, que não seja a fábrica, crie objetos DAO (linha 3). A preservação dessa restrição garante que a criação de objetos não ocorra irrestritamente, definindo-se um local centralizado para criação de objetos DAO. Em consequência, isso beneficia a manutenibilidade do sistema uma vez que manutenções relacionadas à forma como tais objetos são criados, ao número máximo de instâncias permitidas, à existência de caches etc., estarão centralizadas em uma única classe.

É importante mencionar que além de abstrair os clientes da complexidade de criação dos objetos, o padrão Abstract Factory também provê todos os benefícios proporcionados pelo princípio da programação por interface. Isso ocorre, pois um objeto é normalmente retornado pela fábrica como seu tipo abstrato. Como um exemplo, ao solicitar o DAO da entidade Cliente, a fábrica retorna uma implementação Hibernate (ClienteHibernateDAO) como sua interface IClienteDAO, favorecendo assim a programação por interface.

Reúso

Reúso consiste na criação de componentes – da forma mais genérica possível – de modo que possam ser reutilizados em diversos projetos. Portanto, tais componentes devem depender apenas de frameworks e bibliotecas comuns a todos os sistemas em que serão utilizados. Por exemplo, a figura 7 ilustra dois sistemas que compartilham as mesmas classes utilitárias (módulo Util). Nesse caso, deve-se garantir que o módulo Util depende apenas da API de Java (módulo JavaAPI) e da biblioteca de classes utilitárias provida pela Apache (módulo ApacheUtils).

Para garantir a reusabilidade das classes utilitárias, as seguintes restrições DCL podem ser definidas:

```
module Util:        org.foo.util.*
module ApacheUtils: org.apache.commons.**
Util can-depend-only Util, ApacheUtils, $java
```

Após a definição dos módulos Util e ApacheUtils (linhas 1 e 2), uma restrição garante que classes utilitárias estabeleçam dependências com apenas certas classes do sistema (linha 3). Isso garante que classes utilitárias 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 sujeita a erros.

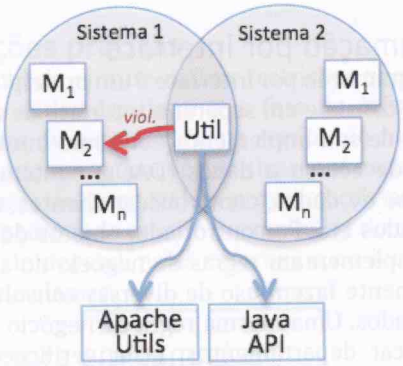

Figura 7. *Promovendo reúso de classes utilitárias.*

Controle de dependências externas

Sistemas de software normalmente se baseiam em frameworks existentes para uma série de atividades, como persistência, interface gráfica, relatórios etc. No entanto, esse acoplamento deve ser controlado de forma que o framework seja acoplado somente às devidas classes. Por exemplo, conforme ilustrado na figura 8, somente o módulo de interface gráfica do sistema (módulo UI) deve utilizar serviços providos pelo framework Swing. Em outras palavras, qualquer outro módulo do sistema que utilize o framework Swing está violando a arquitetura planejada.

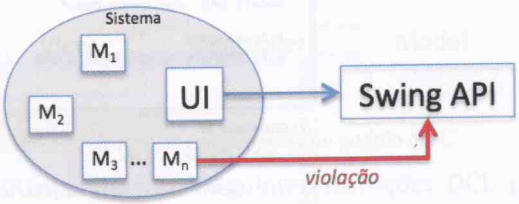

Figura 8. *Acoplamento com o framework Swing.*

Para ilustrar esse controle de dependência, as seguintes restrições DCL podem ser:

```
module UI:    org.foo.view.ui.*
module Swing: javax.swing.**
only UI can-depend Swing
```

Após a definição do módulo de interface gráfica do sistema e do módulo que representa as classes do framework Swing (linhas 1 e 2), uma restrição garante que não sejam estabelecidos acoplamentos indevidos com essa framework. Esse controle de dependências proporciona benefícios à manutenibilidade similares àqueles proporcionado por uma arquitetura em camadas, ou seja, garante nesse exemplo que modificações de versões de frameworks ou, até mesmo, a migração para outro framework, não impacte classes não relacionadas.

Herança

No momento em que várias classes compartilham propriedades e comportamentos comuns, herança é a solução adotada a fim de evitar duplicidade de código e promover a manutenibilidade. A figura 9 ilustra um exemplo em que todos os objetos de transferência de dados (DTOs) devem armazenar um identificador único, com a versão e data da última alteração. Diante disso, o arquiteto desenvolveu uma classe base padrão (BaseDTO) que todo DTO deve estender.

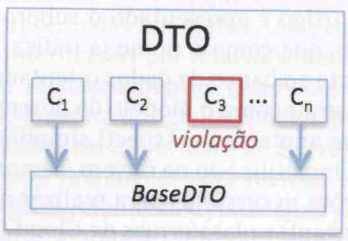

Figura 9. *Herança de uma classe base.*

Para forçar que desenvolvedores utilizem essa classe base, as seguintes restrições DCL foram definidas:

```
module DTO: org.foo.model.dto.*
DTO must-extend org.foo.model.dto.BaseDTO
```

Com o módulo DTO definido (linha 1), uma restrição força que todas as classes DTO estendam sua classe base. Isso garante que desenvolvedores não implementem classes fora do padrão ou operações que já foram implementadas, evitando duplicidade de código e mantendo um padrão de desenvolvimento. Além disso, uma alteração em todas as classes de um conjunto pode ser realizada por meio de sua classe base, pois se garante que todas as classes do conjunto a estendem.

Considerações finais

O problema de garantir que a arquitetura planejada de um sistema seja efetivamente seguida durante sua implementação e evolução possui inquestionável importância. No entanto, no decorrer do projeto, a arquitetura tende a se degradar devido a diversos fatores, como desconhecimento por parte dos desenvolvedores, dificuldades técnicas etc. Mais importante, a erosão arquitetural pode fazer com que benefícios típicos de um bom projeto arquitetural – tais como escalabilidade, reusabilidade e manutenibilidade – sejam anulados.

Diante disso, DCLcheck fornece aos desenvolvedores uma forma eficaz para reduzir ou mesmo evitar o processo de erosão arquitetural em seus projetos. Assim que definidas as restrições arquiteturais na linguagem DCL, DCLcheck provê conformação arquitetural por construção.

Embora seja inviável demonstrar todos os cenários em que DCLcheck possa ser aplicado, ilustrou-se neste artigo como utilizá-lo para especificar diversos padrões arquiteturais e boas práticas de programação, tais como arquitetura em camadas, programação por interface etc.

Com o trabalho em andamento, está sendo desenvolvido um sistema de recomendação chamado DCLfix que provê recomendações para corrigir violações detectadas por DCLcheck. Por exemplo, suponha que mesmo existindo uma fábrica de DAOs, um DAO seja criado diretamente. DCLcheck irá reportar uma violação e então DCLfix irá não apenas sugerir a substituição da criação direta pelo uso da fábrica, mas também complementar essa informação com o nome do método da fábrica que deve ser acessado.

/referências

> *Página Web do DCL. http://www.dclsuite.org.*

> *Leonardo Teixeira Passos; Ricardo Terra; Renato Diniz; Marco Túlio Valente; Nabor das Chagas Mendonça. Static Architecture Conformance Checking: An Illustrative Overview. IEEE Software, 2010.*

> *Ricardo Terra; Marco Túlio Valente. A Dependency Constraint Language to Manage Object-Oriented Software Architectures. Software: Practice and Experience, 2009.*

> *Ricardo Terra; Marco Túlio Valente. Definição de Padrões Arquiteturais e seu Impacto em Atividades de Manutenção de Software. VII Workshop de Manutenção de Software Moderna (WMSWM), 2010.*

> *Ricardo Terra; Marco Tulio Valente; Krzysztof Czarnecki; Roberto Bigonha. Recommending Refactorings to Reverse Software Architecture Erosion. 16th European Conference on Software Maintenance and Reengineering (CSMR), Early Research Achievements Track, 2012.*

> *Ricardo Terra; Marco Tulio Valente; Krzysztof Czarnecki; Roberto Bigonha. DCLfix: A Recommendation System for Repairing Architectural Violations. III Congresso Brasileiro de Software: Teoria e Prática (CBSOFT), Sessão de Ferramentas, 2012.*