Fábrica dinâmica de dublês: testando classes que possuem dependências não injetáveis

Leonardo Alexandre Ferreira Leite[1]

[1]Departamento de Ciência da Computação
IME - Universidade de São Paulo (USP)

leofl@ime.usp.br

Abstract. *This paper presents a test design pattern to help testing classes that contain internal dependencies that cannot be easily injected to the class. The solution uses a dynamic factory, that has its default behavior changed in test environment to return double objects.*

Resumo. *Este artigo apresenta um padrão de projeto para testes para a situação em que a classe sob teste possui alguma dependência interna que não pode ser facilmente injetada. A solução utiliza uma fábrica dinâmica, que possui seu comportamento padrão alterado no ambiente de testes para retornar objetos dublês.*

1. Introdução

A utilização de testes de unidade, advogada por métodos ágeis como o XP, já se tornou prática comum na indústria de software. Uma importante técnica para a criação de testes de unidade é a utilização de *objetos dublês*, que substituem a utilização de um objeto real para propósitos de teste [Meszaros 2007]. Os principais tipos de objetos dublês são *mock*s e *stub*s [Fowler 2007], que normalmente interagem com o objeto sob teste.

Há diferentes técnicas para estabelecer o relacionamento entre um objeto e o dublê de uma de suas dependências. Neste artigo discutiremos alguns compromissos entre as opções para estabelecer esse relacionamento, bem como apresentar um padrão de projeto com esse objetivo. O padrão apresentado contribui com uma opção de implementação a ser escolhida por desenvolvedores que se deparem com a situação apresentada.

2. Contexto

Na criação de testes de unidade é comum a utilização de objetos dublês para simular o comportamento de determinados objetos participantes do teste. Em algumas circunstâncias, pode ser que o objeto a ser dublado (objeto dependencia) seja uma *dependência interna* da classe sob teste (classe Testada), ou seja, clientes de Testada não deveriam ter conhecimento sobre dependencia. Um problema inicial consiste no fato de que Testada deve encapsular dependencia, enquanto que o teste precisa configurar o objeto dependencia a ser utilizado pela instância de Testada sendo testada.

Nesse caso, aconselha-se a utilização de mecanismos de injeção de dependência [Fowler 2004] para retirar de uma classe a responsabilidade de instanciar suas dependências. Isso faz com que o sistema tenha uma aderência maior ao *princípio da responsabilidade única* [Wampler 2008]. Existem sistemas especializados nesse propósito

que fornecem contêineres de injeção de dependência, como Spring[1] e o Google Guice[2]. O uso de tais ferramentas é o caminho mais indicado para lidar com a configuração de dependências, inclusive a configuração de dublês para testes de unidade.

Uma outra abordagem bem popular, utilizada para simular a injeção de dependência, é a criação de um construtor adicional em Testada para receber a dependencia construída pelo próprio cliente de Testada. Com essa técnica é possível criar no teste um dublê para dependencia, e passar esse dublê para classe testada através do construtor adicional.

Listing 1. Injetando a dependência por meio de um construtor adicional

```
public class Testada {

    private Dependencia dependencia;

    public Testada() {
        this.dependencia = new Dependencia();
    }

    Testada(Dependencia dependencia) {
        this.dependencia = dependencia;
    }

    public void algumMetodo() {
        // faz alguma coisa com dependencia
    }
}
```

Uma variação do construtor adicional é utilização de um método *setter* (Listagem 2). A utilização do *setter* evita a explosão de construtores que pode acontecer com a utilização do construtor adicional caso a classe sob teste já possua vários construtores. No entanto, ao contrário do construtor, permite que invariantes da classe (relações entre atributos) sejam quebradas mais facilmente. Para os objetivos deste artigo, o uso do *setter* para injetar a dependência será considerado equivalente ao uso do construtor adicional.

Listing 2. Injetando a dependência por meio de um *setter*

```
void setDependencia(Dependencia dependencia) {
    this.dependencia =  dependencia;
}
```

Note que o construtor adicional, assim como o *setter*, não é público, possuindo visibilidade apenas para o pacote. Isso ocorre pois eles não devem ser utilizados por código de produção. Apesar disso, a visibilidade para o pacote não impede que eles sejam usados indevidamente por outras classes no mesmo pacote.

Contudo, há casos em que a injeção de dependência não é adequada. Por vezes, a dependência é instanciada várias vezes, em um *loop* por exemplo. Em outros casos, a dependência deve ser construída em função do contexto de um método em execução.

[1] www.springsource.org
[2] code.google.com/p/google-guice/

Nesses casos, a dependência não pode ser simplesmente injetada antes da execução do método. É nessa situação que aplicaremos nosso padrão de fábrica dinâmica de dublês.

3. Problema

Como testar classes que possuem dependências internas na situação em que essas dependências internas *i)* devem ser dubladas durante o teste, e *ii)* não podem ser diretamente injetadas na classe sob teste? Consideramos que uma dependência não pode ser injetada diretamente em casos como o de múltiplas instanciações ou na construção baseada no contexto local de execução.

Neste artigo apresentaremos um padrão de projeto para resolver esse problema, que consiste na utilização de uma fábrica dinâmica para produzir as dependências das classes testadas. Essa fábrica retornará objetos reais no ambiente de produção e objetos dublês no ambiente de teste.

4. Forças

Duas forças que se confrontam no cenário esboçado são 1) o encapsulamento da dependência pela classe testada, e 2) a necessidade de configurar a dependência da classe testada para os testes de unidade. O uso do construtor adicional com visibilidade para o pacote é um compromisso entre essas duas forças, mas que porém não é ideal, pois tem como consequências indesejadas: 1) outras classes no mesmo pacote da classe testada podem utilizar o construtor adicional, e 2) rotinas de teste fora do pacote da classe testada não podem utilizar o construtor adicional.

Temos ainda a escolha da utilização de fábricas no lugar de um contêiner de injeção de dependências. Consideramos que o uso desses contêineres pode ser intimidador para iniciantes, pois se faz necessário o aprendizado de um novo *framework*. Em sistemas legados pode ser difícil a incorporação de um novo *framework* que impacte diversas partes do sistema, como seria o caso de um contêiner de injeção de dependências. Em contraposição, o uso de fábricas possibilita mais facilmente a construção dinâmica de objetos baseada em propriedades obtidas em tempo de execução, o que é conveniente para a situação já descrita, em que dependências internas não podem ser facilmente injetadas.

5. Solução

A solução proposta consiste na obtenção de uma dependência da classe testada por meio da utilização de uma fábrica, e da alteração dinâmica do comportamento dessa fábrica quando executada no ambiente de testes. Com essa alteração, a fábrica deve continuar retornando novas instâncias da dependência em ambiente de produção, mas quando em ambiente de testes, a fábrica deverá devolver um dublê da dependência previamente configurado pelo teste em execução.

5.1. Estrutura

A estrutura de classes do padrão pode ser visualizado na Figura 1.

Participantes:

- Testada: classe sob teste, que possui uma dependência da classe Dependencia.
- Dependencia: dependência interna da classe Testada.

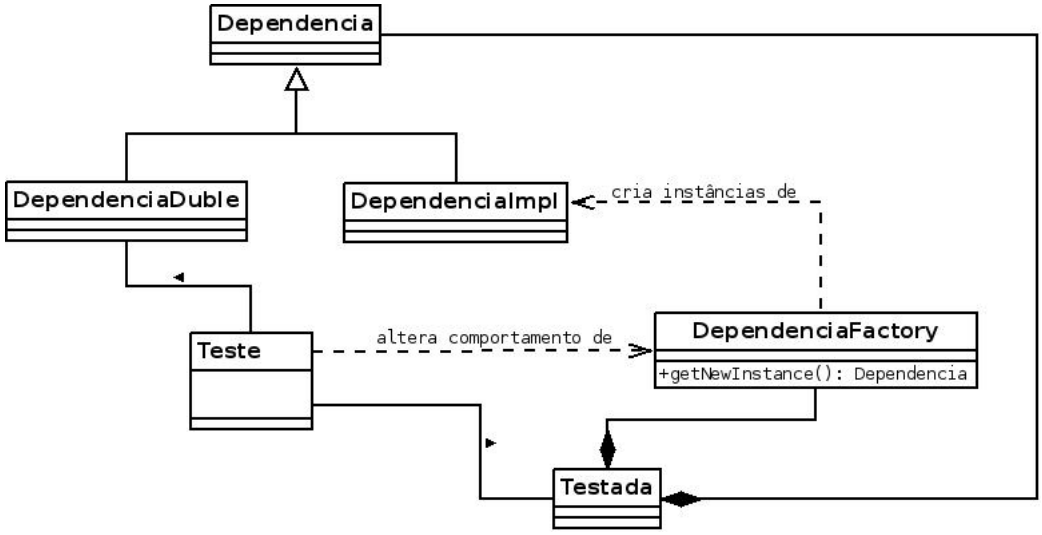

Figura 1. Estrutura de classes do padrão proposto

- DependenciaFactory: fábrica de objetos do tipo Dependencia; é utilizada pela classe Testada.
- DependenciaImpl: implementação de Dependencia normalmente utilizada em produção, sendo fornecida por DependenciaFactory.
- Teste: teste da classe Testada.
- DependenciaDuble: dublê de Dependencia para ser usado durante o teste da classe Testada. Pode tanto ser um *stub* quanto um *mock*.

5.2. Implementação

Uma possível implementação para a estratégia estabelecida é o uso de uma *flag* estática guardada na classe da fábrica e alterada pelo teste, além de uma variável estática que armazena o dublê a ser utilizado, como apresentado na Listagem 3. Dessa forma, a alteração no comportamento da fábrica ocorrerá somente quando necessário e somente durante o tempo necessário.

Listing 3. A fábrica dinâmica de dependências

```
public class DependenciaFactory {

    public static boolean testing = false;
    public static Dependencia dependenciaForTesting;

    public Dependencia getNewInstance() {
        if (testing)
            return  dependenciaForTesting;
        else
            return new DependenciaImpl();
    }
}
```

Utilizando a fábrica descrita na Listagem 3 poderemos escrever o teste da classe Testada, objetivo do padrão proposto, como mostrado na Listagem 4.

Listing 4. Teste da classe Testada

```
public class ClienteTest {

    private Dependencia dependenciaMock;

    @Before
    public void setUp() {
        dependenciaMock = ... // cria dublê
        DependenciaFactory.testing = true;
        DependenciaFactory.dependenciaForTesting =  dependenciaMock;
    }

    @Afer
    public void tearDown() {
         DependenciaFactory.testing = false;
    }

    @Test
    public void shouldDoSomething() {
        // testa Testada em alguma situação em que Dependente é utilizado
    }
}
```

Há outras alternativas para a implementação do padrão proposto, como o condicional de DependenciaFactory ser exercido em função de uma variável de ambiente[3]. Outra opção seria a utilização de Programação Orientada a Aspectos, com rotinas a serem executadas no teste que alterariam dinamicamente o código de DependenciaFactory para que ela retornasse o dublê desejado.

Por fim, uma variante da fábrica dinâmica é dublar a própria fábrica e passá-la para a classe testada através de um construtor adicional ou um `setter`.

5.3. Exemplo

A classe Carteiro (Listagem 5) depende do objeto rotaFinder, que mostra ao carteiro a rota que ele deve seguir para entregar uma carta. rotaFinder é um exemplo de dependência interna do Carteiro, ou seja, um detalhe de implementação que não deve ser de conhecimento dos clientes de Carteiro, pois a esses interessa apenas entregar a carta ao carteiro, e não como o carteiro decide o caminho para entregar a carta. Além disso, RotaFinder deve ser construído dinamicamente em função da localização atual do carteiro e do endereço da carta a ser entregue. Por esse motivo, não é adequado injetar RotaFinder através de um construtor adicional ou por um *setter*.

[3]No *framework* Django utiliza-se a variável de ambiente DJANGO_SETTINGS_MODULE para indicar quando o sistema se encontra em ambiente de produção, desenvolvimento ou testes.

Listing 5. Classe Carteiro original (difícil de ser testada)

```
public class Carteiro {

    private Endereco posicaoAtual;

    public void entrega(Carta carta) {
        Endereco enderecoDestino = carta.getEnderecoDestino();
        RotaFinder rotaFinder =
             new RotaFinder(this.posicaoAtual, enderecoDestino);
        Rota rota = rotaFinder.findRota();
        // usa a rota pra entregar a carta
    }

}
```

Como a classe RotaFinder utiliza serviços web reais para traçar uma rota, seu uso é indesejável no ambiente de testes. Por isso desejamos utilizar um dublê de RotaFinder no teste de Carteiro. Resolveremos o problema criando a fábrica RotaFinderFactory (Listagem 6), que retornará a implementação real de RotaFinder no ambiente de produção, e um dublê de RotaFinder no ambiente de testes. Dessa forma, a nova implementação de Carteiro ficará como a da Listagem 7.

Listing 6. RotaFinderFactory: a fábrica dinâmica de dublês

```
public class RotaFinderFactory {

    public static boolean testing = false;
    public static RotaFinder rotaFinderForTesting;

    public RotaFinder getNewRotaFinder(Endereco origem, Endereco destino) {
        if (testing) {
            return  rotaFinderForTesting;
        } else {
            return new RotaFinder(origem, destino);
        }
    }
}
```

Listing 7. Classe Carteiro que utiliza RotaFinderFactory

```
public class Carteiro {

    private Endereco posicaoAtual;

    public void entrega(Carta carta) {
        RotaFinderFactory rotaFinderFactory = new RotaFinderFactory();
        Endereco enderecoDestino = carta.getEnderecoDestino();
        RotaFinder rotaFinder = rotaFinderFactory.
              getNewRotaFinder(posicaoAtual, enderecoDestino);
        Rota rota = rotaFinder.findRota();
        // usa a rota pra entregar a carta
    }

}
```

Em fim, temos o teste de Carteiro como descreve a Listagem 8.

Listing 8. Teste de Carteiro que altera o comportamento de rotaFinderFactory

```
public class CarteiroTest {

    RotaFinder rotaFinderMock;

    @Before
    public void setUp() {
        this.rotaFinderMock = ... // cria mock
        RotaFinderFactory.testing = true;
        RotaFinderFactory.rotaFinderForTesting =  this.rotaFinderMock;
    }

    @Afer
    public void tearDown() {
         RotaFinderFactory.testing = false;
    }

    @Test
    public void deveriaEntregarCarta() {
        Carteiro carteiro = new Carteiro();
        Carta carta = ...
        carteiro.entrega(carta);
        // verifica de alguma forma se a carta foi entregue
        // no endereço retornado pelo rotaFinderMock
    }
}
```

5.4. Um exemplo mais extremo

Como mencionado anteriormente, o exemplo descrito poderia ser resolvido com a criação de um *mock* de RotaFinderFactory que seria injetado no Carteiro. Discutiremos mais para a frente algumas diferenças entre as implicações dessas alternativas. Mas vamos discutir agora uma situação peculiar em que o construtor adicional não poderia ser utilizado.

Supondo a existência de uma classe Correios que possui uma coleção de Carteiros, essa classe pode ser testada com a criação de dublês para seus carteiros. Mas consideremos que a menos da interação com RotaFinder, a implementação da classe Carteiro poderia ser utilizada no ambiente de teste, por não provocar nenhum efeito colateral no mundo real (como chamadas a serviços web, por exemplo). Nessa situação, há quem prefira não dublar Carteiro. Essa preferência por utilizar o máximo de objetos reais no ambiente de teste é classificada por Fowler como "TDD clássico" [Fowler 2004]. Além dessa preferência, a utilização de objetos reais pode ser considerada em testes de integração entre classes.

Na situação descrita, caso Correios esteja em pacote diferente de Carteiro, não será possível para o teste de Correios configurar um *mock* de RodaFinderFactory e injetá-la em instâncias de carteiros, impedindo a utilização de carteiros reais no teste. Mas com a utilização da fábrica dinâmica de dublês, é possível para o teste de Correios configurar RodaFinderFactory para retornar um *mock* adequado de RotaFinder durante a execução do teste.

6. Usos conhecidos

Mostraremos agora algumas aplicações da fábrica dinâmica de dublês utilizadas no projeto CHOReOS Enactment Engine[4].

No primeiro caso, desejamos testar o método `cast` da classe ContextCaster. Esse método recebe uma *coreografia*, que especifica uma rede de dependências entre serviços web. Para cada dependência, ContextCaster utiliza ContextSender para realizar uma requisição ao serviço dependente, informando sua dependência. A implementação SoapContextSender é que sabe dos detalhes sobre como fazer a chamada a um serviço SOAP para contar-lhe sobre suas dependências.

Como o método `cast` apenas realiza essas chamadas e não tem um retorno a ser verificado, o que podemos fazer para testa-lo é verificar seu comportamento com um *mock* de ContextSender. Assim, o *mock* serve tanto para evitar a realização de uma requisição SOAP real durante o teste, o que seria indesejável, quanto para podermos verificar as operações invocadas neste objeto. Mas a utilização do ContextSender deve ser transparente ao cliente de ContextCaster, uma vez que esse cliente quer simplesmente que as mensagens pertinentes sejam entregues, não se importando *como* elas serão entregues. Por esse motivo, o teste não tem como entregar o *mock* criado diretamente para a instância de ContextCaster sob teste. A saída para isso é fazer o teste alterar dinamicamente o comportamento de ContextSenderFactory. Assim, em produção, ContextCaster receberá de ContextSenderFactory uma instância de SoapContextSender, enquanto que durante a execução do teste, receberá o *mock* configurado pelo próprio teste.

Em nosso segundo caso, a classe NodesUpdater deve atualizar todos os nós em que se encontram os serviços passados por seu construtor. Para cada nó, NodesUpdater realiza uma chamada a UpdateNodeInvoker, que por sua vez utilizará uma instância de NodePoolManager para efetuar uma chamada REST que provocará a atualização do nó específico. No teste de NodesUpdater alteramos então o comportamento de RESTClientsRetriever (uma fábrica de NodePoolManager), para que ele entregue à UpdateNodeInvoker um dublê de NodePoolManager, a fim de evitar uma chamada REST real durante os testes. Nesse exemplo, a dependência direta (UpdateNodeInvoker) da classe sob teste nem aparece no teste, pois o que é configurado pelo teste é um dublê para uma dependência indireta, o NodePoolManager.

O terceiro exemplo de uso conhecido será o teste da classe NodeBootstrapper. Essa classe realiza uma série de configurações no sistema operacional de uma máquina virtual (um *nó)*. Para isso, dentre outras coisas, é preciso enviar alguns arquivos ao nó, o que é feito com o SCP. No entanto, no ambiente de testes de unidade não haverá um nó real para receber as chamadas SCP. Por isso o teste utiliza um dublê de Scp, que é a classe que implementa a chamada SCP. Assim, o teste de NodeBootstrapper configura a fábrica de Scps para retornar um *mock* de Scp a ser utilizado no teste. Assim, o teste é considerado sucesso se o *mock* de Scp, criado pelo teste e obtido por NodeBootstrapper pela fábrica dinâmica, recebeu as devidas chamadas para enviar os arquivos necessários.

[4] `http://choreos.eu/bin/Documentation/enactment_engine_doc`

7. Consequências

A consequência positiva mais importante é a facilidade para testar classes que possuem dependência internas que devem ser dubladas, mesmo na situação em que não é possível injetar diretamente tais dependências.

Como consequência negativa poder-se-ia considerar a combinação de código de produção com código de teste em uma mesma classe. No entanto, também podemos considerar que a solução do construtor adicional para injetar um dublê da fábrica de dependência também coloque código que só será usado no teste em meio ao código de produção. O vazamento de encapsulamento da classe testada é comprometido tanto no uso da fábrica dinâmica de dublês, quanto no construtor adicional para receber um dublê da fábrica de dependência.

Sobre esses dois problemas, a fábrica dinâmica de dublês possui uma ligeira vantagem. Tanto a mistura de código de produção com código de teste, quanto o vazamento de encapsulamento são feitos de forma semanticamente mais explícita em nosso padrão proposto. Ou seja, o entendimento da API da classe testada pode ficar mais comprometida no caso do construtor adicional, pois esse aparece de forma bem discreta, possivelmente bem próximo a um outro construtor muito parecido. Mas ao se utilizar a fábrica dinâmica de dublês, é mais explícito ao desenvolvedor que testing e dependenciaForTesting não devem ser utilizados em código de produção. Além disso, a implementação pode sempre seguir a forma indicada na Listagem 3, o que se seguido de forma sistemática ajudaria os desenvolvedores a identificarem o padrão. Não há uma coerção sintática que impeça o desenvolvedor de usar indevidamente os atributos de teste, porém lembramos que em linguagens dinâmicas é comum o uso de convenções para denotar visibilidade[5].

Uma desvantagem do padrão proposto é a baixa flexibilidade de seu *design*, pois no momento determinado pelo teste, a fábrica retornará o mesmo dublê a todos os seus clientes, mesmo que não estejam envolvidos no teste. Esse *design*, portanto, assume a premissa (razoável) de que o comportamento da fábrica só será alterado por testes de unidade e que os testes de unidade são executados sequencialmente, mas o que de fato é situação típica.

8. Padrões relacionados

Padrões de criação [Gamma et al. 1995] se relacionam ao nosso padrão proposto. Este texto apresentou a classe DependenciaFactory, que poderia ser usada no contexto do padrão *Factory Method*: a DependenciaFactory seria uma das fábricas concretas do padrão *Factory Method*. Nesse caso, o teste deve se responsabilizar por alterar o comportamento de todas as fábricas concretas que seriam utilizadas no teste.

No caso da utilização do padrão *Abstract Factory*, objetos dublês deverão ser fornecidos para cada produto de uma família de produtos que serão utilizados no teste. Assim, seguindo a implementação do padrão proposto, cada fábrica concreta teria uma variável testing, e mais uma variável para cada produto, a fim de armazenar os objeto dublês necessários. A discussão feita sobre o padrão *Factory Method* se aplica igualmente ao padrão *Abstract Factory*.

[5]Desenvolvedores Python utilizam o *underline* ("_") como prefixo de um atributo para denotar que se trata de um atributo privado.

A fábrica utilizada em nosso padrão é um caso particular do padrão de *Service Locator* [Fowler 2004], pois é ela que define qual implementação de Dependencia será utilizada pela classe testada. Do mesmo modo que Fowler mostra como um *service locator* pode ser configurado pelo teste, nossa fábrica é também dinamicamente configurada pelo teste. A diferença é que enquanto o *service locator* deve necessariamente ser configurado, seja em teste, seja em produção, nossa fábrica possui um *comportamento padrão*, sendo configurada somente no ambiente de teste.

Por fim, outros padrões relacionados são os padrões de teste [Meszaros 2007], como *mocks* e *stubs*, uma vez que o objetivo de nosso padrão é fornecer objetos dublês (*mocks* e *stubs*) ao código em execução no ambiente de testes.

9. Conclusão

O padrão proposto, a fábrica dinâmica de dublês, fornece uma forma simples para estabelecer uma associação entre objetos dublês e a classe testada em um teste de unidade. A implementação afeta apenas o código de teste e o código da fábrica do objeto a ser dublado, não afetando o código da própria classe testada.

Agradecimentos

Agradeço àqueles com quem pude inicialmente discutir o padrão proposto: Alan Fachini, Eduardo Hideo, Felipe Besson e Maurício Aniche. Agradeço muito também a Eduardo Guerra, cuja ajuda e revisões foram fundamentais para este artigo.

Referências

Fowler, M. (2004). Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, acessado em 4 de agosto de 2013.

Fowler, M. (2007). Mock aren't stubs. http://martinfowler.com/articles/mocksArentStubs.html, acessado em 30 de julho de 2013.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Chap. 3 Creational patterns. In *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley.

Wampler, K. D. (2008). Chap. 11 Systems. In Martin, R. C., editor, *Clean Code, A Handbook of Agile Software Craftsmanship*. Prentice Hall.