Aplicação de Diretrizes para Teste de Unidade em Software Baseado em Componentes

Thelma E. Colanzi, Fábio A. Atala, Gustavo Y. Sato, Elisa H. M. Huzita

Departamento de Informática – Universidade Estadual de Maringá (UEM)
Av. Colombo, 5790 - 87020-900 – Maringá – PR - Brasil
{thelma, emhuzita}@din.uem.br, fabioatala@yahoo.com.br, gus.sato@gmail.com

Abstract. The Component-based Software Engineering provides many benefits to software development process. Nevertheless, it may increases the coupling level among objects of a software application. Thus, the unit testing of component-based software can be limited. Therefore, it is necessary to plan software testing strategies which make the execution of the tests possible for this kind of software. In this context, this paper presents alternatives for realization of unit testing and their application in two case studies in a distributed software engineering environment. At the end, they are compared emphasizing their advantages and their disadvantages.

Keywords: unit software testing, component-based software, test stubs

1. Introdução

O desenvolvimento de software baseado em componentes (DSBC) consiste na reutilização de software por meio do encapsulamento de funcionalidades em unidades que são independentes e podem ser acopladas a um sistema por meio de uma interface conhecida. Recentemente, grande parte dos sistemas de software desenvolvidos é baseada em componentes visando alcançar alta qualidade, alta produtividade e redução de custos. Além disso, por meio do DSBC há a possibilidade de integração rápida e da facilidade para a montagem do sistema [1].

Os componentes podem ser escritos em diversas linguagens de programação e executados em várias plataformas operacionais. Um componente pode ser implementado como caixa-branca ou caixa-preta [2]. Se ele for caixa-branca, a combinação de partes já existentes pode ser feita acessando e alterando o código do componente. Isso é possível se o componente tiver sido desenvolvido internamente à empresa. Caso seja caixa-preta, são feitas apenas combinações de componentes por meio das interfaces e criação de novos componentes sem alterar código dos componentes, como é o caso de COTS (do inglês *Commercial Off-The-Shelf*) que são geralmente obtidos de terceiros, mas podem também ser desenvolvidos internamente. Pode-se optar por usar só componentes do tipo caixa-preta, só caixa-branca ou uma

combinação dos dois tipos. Especificamente para o desenvolvimento de software em Java, há uma vasta gama de componentes disponíveis.

A concepção de software baseado em componentes normalmente leva ao desenvolvimento de classes específicas da aplicação para interagir com os componentes reusados. No momento da implementação, a busca por qualidade e produtividade costuma levar também ao uso de padrões de projeto de software. Esses descrevem soluções para problemas recorrentes no desenvolvimento de software orientado a objetos (OO). Um padrão de projeto estabelece um nome e define o problema, a solução, quando aplicá-la e suas consequências [3]. Dentre os padrões de projeto propostos por Gamma *et al*. [3], os padrões *Factory* e *Singleton* estão entre os mais comumente utilizados no desenvolvimento de software.

Visando a qualidade na produção de software, é importante a presença das atividades de teste em todo processo de desenvolvimento, pois ela tem como objetivo principal revelar defeitos em produtos que estão sendo testados. As atividades de teste devem acontecer em diferentes níveis, verificando primeiramente as unidades do software, posteriormente a integração entre unidades e, finalmente, o teste de sistema que direciona esforços para identificar erros e características que estejam contra a especificação do sistema após a sua integração por completo [4].

Apesar dos vários aspectos do DSBC, Vincenzi *et al*. [5] afirmam que as técnicas tradicionais de teste podem ser utilizadas nos testes deste tipo de software. No entanto, o uso de componentes no desenvolvimento de software, apesar de trazer muitos benefícios, traz uma maior complexidade à arquitetura do software e, consequentemente, gera problemas para a atividade de teste. Isto porque o encapsulamento dos componentes interfere no acoplamento entre os objetos e influencia o teste de unidade. Para realizar a primeira das atividades de teste, é preciso isolar cada uma das unidades que serão testadas. Para isso utiliza-se um *stub,* que é um programa que substitui os objetos que estejam subordinados ao método a ser testado simulando as suas funcionalidades. Normalmente, *stubs* são utilizados para isolar a unidade para a realização do teste. No entanto, algumas decisões de projeto podem criar dependências entre classes difíceis para serem quebradas e substituídas por *stubs*, necessitando de medidas adicionais para a realização dos testes de unidade.

Neste trabalho abordam-se quatro alternativas para viabilizar a realização do teste de unidade em software baseado em componentes visando oferecer, aos membros das comunidades industrial e acadêmica, um conjunto de diretrizes para a realização de teste de unidade em DSBC. Apresenta-se também a aplicação das diretrizes em dois estudos de caso, o que fornece indícios de quando é viável empregá-las.

Este artigo está organizado em cinco seções. Nas Seções 2 e 3 são apresentadas, respectivamente, as alternativas propostas e sua aplicação em dois estudos de caso. Na Seção 4 é feita uma breve comparação entre as alternativas e por fim, na Seção 5, são feitas algumas considerações finais a respeito do tema abordado.

2. Diretrizes para Teste de Unidade

Segundo Gazola e David [6] “a maneira mais simples de se organizar os testes de unidade em um projeto consiste em colocar tanto o código de teste quanto o código

testado no mesmo diretório. A vantagem dessa abordagem é que os testes podem acessar diretamente os membros protegidos das classes sob teste e de produção". Porém, de acordo com o projeto do software, nem mesmo o uso de *stubs* e da classe de teste no mesmo diretório da classe testada é suficiente para a execução do teste, por exemplo, quando é preciso acessar um atributo protegido da superclasse que se encontra em um outro pacote. Sendo assim, a seguir apresentam-se as alternativas para viabilizar o teste de unidade de software baseado em componentes constituindo-se num conjunto de diretrizes para a realização dessa atividade.

2.1 Utilizar métodos *setter e getter* públicos

Esta solução consiste em transformar para pública a visibilidade de alguns atributos e métodos *setters* e *getters* privados ou protegidos. Isto pode ser feito para os métodos *set* e *get* do atributo do qual o método a ser testado não tem visibilidade. Desta forma, o método de teste poderá instanciar o *stub* por meio de um método *setter* público.

A vantagem dessa solução é que as dependências podem ser alteradas em tempo de execução. Por outro lado, ela só é viável se a implementação utilizar *setters* e *getters* públicos e ela não é aplicável em software que utiliza o padrão *Singleton*, porque não é possível substituir a instância de um *Singleton* dinamicamente. Esse padrão permite que uma classe tenha apenas uma única instância na aplicação.

2.2 Alterar diretamente o Banco de Dados

Uma estratégia muito utilizada pelos testadores e aplicável a qualquer software é realizar os testes alterando diretamente o banco de dados (BD). A vantagem de utilizar o BD é que não é necessário criar *stubs.* Embora seja uma maneira muito simples de realizar os testes, é preciso tomar cuidado para não esquecer de remover o lixo que foi gerado dentro das tabelas do BD.

2.3 Utilizar injeção de dependência

Em sistemas nos quais o pedido de instanciação de objetos ocorre por meio de um arquivo de configuração XML, é possível modificar as configurações desse arquivo para realizar os testes [7]. Para isso pode-se utilizar a injeção de dependência, um padrão de projeto que ajuda a minimizar o nível de acoplamento entre módulos [8]. Utilizando esse recurso, as dependências entre os módulos são definidas por uma infra-estrutura de software que as injeta em cada componente. Assim, as dependências são criadas em tempo de execução por meio do arquivo de configuração, no qual no lugar das dependências reais do projeto são carregados os *stubs* com os quais as classes serão testadas.

Trata-se de uma solução simples, rápida e que altera apenas o arquivo de configuração. Outra vantagem é que essa solução é útil para simular as dependências geradas pelo padrão *Factory* (que contém um método que instancia um objeto de qualquer tipo). Ao contrário da primeira solução, esta não altera a instância do

mecanismo de persistência, mas altera a instância de atributo protegido retornado por ele sem modificar o código. É possível usar somente um *stub* por caso de teste porque a dependência é injetada no momento da inicialização do programa. No entanto, as alterações feitas no arquivo de configuração devem ser desfeitas ao fim do teste, ou, para evitar este tipo de problema, pode-se manter um arquivo de configuração de dependências de testes separado do arquivo de configuração da aplicação.

2.4 Utilizar Reflexão Computacional

Gazola e David [6] sugerem a API Java para reflexão como uma possível solução para testar métodos privados. Nesse caso, é possível obter e invocar, dinamicamente, métodos de qualquer classe (privados ou não). O maior benefício do uso da reflexão para os testadores de software é a possibilidade de quebrar dependências sem a necessidade de mexer diretamente no código fonte. Isto é, a classe de teste que utiliza reflexão é suficiente para quebrar as dependências por atributos protegidos ou privados. A desvantagem em usar reflexão é que o código de testes será menos claro e não é de domínio geral. Isto se deve ao fato de a reflexão ser considerada um recurso avançado que é utilizado geralmente por programadores experientes.

O recurso de reflexão permite fazer alterações adaptativas em tempo de execução para tratar atributos privados ou protegidos como uma classe à parte. A reflexão deixa o atributo acessível e o troca por um *stub* dentro da instância que o referencia. Assim o atributo é reconhecido pela classe de teste. Posteriormente, a reflexão retorna o atributo para seu estado original. Desta maneira é possível utilizar *stubs* para atributos protegidos que precisarão ser acessados no teste.

3. Aplicação das Alternativas Apresentadas

Cada uma das alternativas apresentadas neste artigo foi aplicada em dois estudos de caso. Todos os detalhes dos estudos de caso estão descritos no trabalho de Atala [9]. A realização de testes foi automatizada utilizando a ferramenta JUnit[10] que permite criar classes de teste para programas Java e testar o software em partes separadas, parcialmente integradas ou todas de uma só vez.

Os dois estudos de casos, descritos nas próximas subseções, consistiram de realizar teste de unidade em duas versões de um ambiente de desenvolvimento distribuído de software (ADDS), o DiSEN (*Distributed* Software *Engineering Environment*) [11], ambas desenvolvidas em Java. O DiSEN permite a definição de uma arquitetura de produtos; suporte à persistência de dados; suporte à gerência de recursos, de artefatos, de processos e de projetos; infra-estrutura de comunicação e tecnologia de colaboração. Durante os testes do DiSEN, as dependências entre classes de pacotes distintos, em função do uso de componentes, não puderam ser resolvidas por meio da proposta de Gazola e David [6] motivando a realização dos estudos de caso. No DiSEN, são usados componentes caixa-branca e caixa-preta, porém, as alternativas apresentadas neste trabalho foram aplicadas somente aos componentes caixa-branca.

A aplicação das alternativas será ilustrada utilizando o método *beforeInsert* que pertence à classe ProjetoServerBO cujo código fonte é mostrado abaixo. Esse método

é utilizado para apoiar a criação de uma nova fase dentro de um projeto no DiSEN. Ele verifica a existência de uma fase em um determinado projeto e retorna *true* quando a fase não existe e pode ser inserida, caso contrário, uma exceção é lançada.

Código fonte do método *beforeInsert* extraído do código do DiSEN

```
1 public class ProjetoServerBO extends StandardBO<Projeto>
1 implements ProjetoBO {
2 protected boolean beforeInsert(Projeto obj) throws Exception {
3   // verificar se existe um projeto com o mesmo nome
4   ProjetoDAO projetoDAO = (ProjetoDAO) dao;
5   if(projetoDAO.getProjetoByNome(obj.getNome()) != null)
6      {throw new Exception("Essa fase já existe");}
7   return true;
8  }
9 }
```

3.1 Estudo de Caso: DiSEN versão 1

Esta versão do DiSEN foi construída utilizando o mecanismo de persistência do Hibernate e vários padrões de projeto, dentre eles os padrões *Factory*, *Singleton* e *Data Access Object (DAO)*. No DiSEN, a *Factory* instancia objetos por meio de configuração XML e os objetos criados por ela possuem o comportamento de um *Singleton*. O emprego do padrão DAO[12] possibilita dividir a aplicação em camadas de DAO, BO e Cliente. A camada Cliente requisita um serviço, a camada BO se torna responsável por iniciar a transação, validar as regras de negócio e controlar as transações de banco de dados. Portanto, é uma classe BO quem solicita que uma classe DAO efetue a persistência dos dados. Por meio da interação dos padrões *Factory* e *Singleton*, os objetos criados ou retornados pela *Factory* são protegidos. Assim, eles não ficam visíveis às classes de teste de outros pacotes.

Nesta versão, o método *beforeInsert* depende indiretamente de um atributo protegido de uma classe que está em outro pacote, conforme ilustrado na Figura 1. A classe de teste para o método *beforeInsert* da classe ProjetoServerBO é denominada de ProjetoBOTest. Ela tem acesso aos atributos e métodos protegidos da classe ProjetoServerBO, pois estão no mesmo pacote. Porém, ProjetoServerBO é subclasse de StandardBO, do pacote disen.util.business, e StandardBO, que por sua vez, referencia uma instância de um atributo protegido dao, pertencente à classe DAO, no pacote disen.util.dao. Assim, não é possível acessar o atributo dao mesmo colocando o *stub,* a classe de teste e a classe testada no mesmo diretório Na sequência, apresenta-se o código do *stub* ProjetoDAO para a classe DAO que simula a criação de uma nova fase denominada Dominar o Mundo.

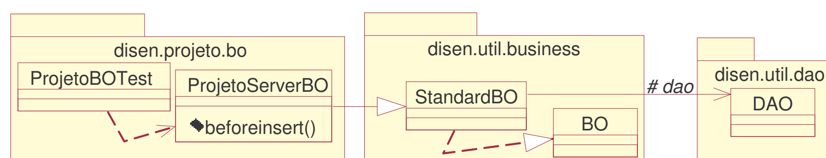

Fig 1. Diagrama de Classes parcial do DiSEN - Arquitetura 1

Stub de ProjetoDao utilizado para o teste do método *beforeInsert*

```
1 public class projetoDaoStub implements  ProjetoDAO{
2  public Projeto getProjetoByNome(String value) throws
2  Exception {
3      Projeto p = null;
4      if (value.equals("Dominar o mundo")){
5           p = new Projeto();
6           p.setId(1);
7           p.setNome("Dominar o mundo");
8      }
9    return p; }
10 }
```

A solução que sugere a manipulação direta do BD para a viabilização dos testes não necessita da criação de *stubs.* Ela foi realizada por meio de um método *SetUp* da JUnit que inseriu os dados obrigatórios de um projeto e de um processo no BD. Os testes foram realizados e, na sequência, por meio do método *TearDown,* os dados foram excluídos do BD.

Para solucionar o mesmo problema usando a primeira solução proposta, os métodos *setters* e *getters* do atributo protegido (dao) tiveram sua visibilidade alterada para pública. Na sequência a classe de teste (vide código a seguir) pôde instanciar o *stub* ProjetoDaoStub por meio do método *setter* público *setDao*. No entanto, no DiSEN, em algumas classes não existem *setters* inviabilizando esta solução.

Classe de teste usando *setter* público

```
1  Public void testBeforeInsert() {
2    System.out.println("beforeInsert");
3    Projeto obj = new Projeto();
4    obj.setNome("Dominar o mundo");
5    obj.setDataInicio(new Date(2007,7,10));
6    ProjetoServerBO instance = new ProjetoServerBO();
7    instance.setDao(new ProjetoDaoStub());
8    boolean expResult = false;
9    boolean result;
10   try{
11         result = instance.beforeInsert(obj);
12         assertEquals(expResult, result);
13   } catch(Exception ex){
14       assertTrue(true); }
15 }
```

Algumas classes do DiSEN são instanciadas com *Factory* usando um arquivo de configuração XML. O mecanismo de persistência busca em um arquivo XML qual classe de objetos dao será instanciada. E como, na estrutura de implementação da versão 1 do DiSEN (Figura 1), as referências dao são em sua maioria atributos protegidos, elas precisam ser simuladas por um *stub*. A seguir é mostrado o fragmento de código por meio do qual o mecanismo de persistência busca, em um arquivo XML, qual é a classe a ser instanciada (linha a). A injeção de dependência pode ser utilizada conforme mostrado na linha (b) onde o atributo protegido é trocado por seu *stub*.

Trechos de código usados na solução que usa injeção de dependência (DiSEN versão 1)

(a)
```
<mapping key ="disen.projeto.bean.Projeto"
target="disen.projeto.dao.ProjetoHibernateDAO"/>
```
(b)
```
<mapping key="disen.projeto.bean.Projeto"
target="disen.projeto.dao.StubProjetoHibernateDAO"/>
```

Na última solução apresentada ocorre a troca do atributo protegido dao por um *stub* dentro da instância que o referencia (*Instance*) usando reflexão. Ou seja, tomando como base a classe de teste mostrada anteriormente, para viabilizar o uso da reflexão nessa arquitetura do DiSEN, basta trocar a linha 7 (em negrito) pelo código abaixo.

Excerto da classe de teste que utiliza reflexão para a versão 1 do DiSEN

```
1 try{
2   Field f = instance.getClass().getDeclaredField("dao");
3   f.setAccessible(true);
4   f.set(instance,new ProjetoDaoStub());
5 ...
```

3.2 Estudo de Caso: DiSEN versão 2

A versão 2 do DiSEN é baseada no *framework* FRADE (*Framework* para Infra-Estrutura de um ADDS) [13] que sugere decisões arquiteturais com o intuito de melhorar a persistência dos dados. Neste estudo de caso, a classe de teste depende de uma classe denominada Global que se comporta como uma *factory* que instancia a classe ApplicationContext, responsável pela instanciação de objetos utilizando o método privado appContext. Os atributos instanciados ficam dentro de uma classe denominada Persistencia (apelido da classe MecanismoPersistencia) que por sua vez é um *singleton* e que contém os atributos protegidos dao. Nesta versão são utilizados apelidos para as classes com o intuito de melhorar a legibilidade e a manutenibilidade do código. A classe ApplicationContext utiliza um arquivo configurado por XML, denominado applicationContext.xml, que contém o apelido e o verdadeiro nome da classe que deve ser instanciada para realizar a criação da classe solicitada.

Para fazer a injeção de dependência, basta criar um *stub* de simulação da classe MecanismoPersistencia, e então realizar as devidas alterações no arquivo de configuração. A classe ApplicationContext buscará nesse arquivo qual classe de objeto será instanciada. A linha (a) do código fonte mostrado a seguir repassa a classe a ser instanciada para o apelido "persistência". Neste caso a classe que recebe este apelido é a que corresponde ao mecanismo de persistência. Para substituir o mecanismo de persistência pelo seu *stub* é preciso trocar linha (a) pela linha (b).

Trechos de código usados na solução que usa injeção de dependência (DiSEN versão 2)

(a)
```
<bean id="persistência"  class= "disen.serviço.
persistencia.impl.MecanismoPersistencia"/>
```
(b)
```
<bean id="persistência"  class= "disen.projeto.
bo.impl.test.ProjetoPersistenciaStub"/>
```

Porém, para o teste que utiliza reflexão serão necessários dois *stubs*: ProjetoPersistenciaStub que simula o Mecanismo de Persistência contendo dados

falsos e ApplicationContextStub que, ao invés de mapear os objetos que serão persistidos por meio de XML, realizará o mapeamento para uma tabela em memória.

A Figura 2 ilustra as dependências entre pacotes dessa arquitetura para o exemplo dado. A classe de teste ProjetoBOTest testará o método *beforeinsert* pertencente à classe ProjetoServerBO. A classe de teste tem acesso aos atributos/métodos protegidos da classe ProjetoServerBO, pois estão no mesmo pacote. Porém, ProjetoServerBO acessa a classe ApplicationContext, de pacote 2, por herança, e ApplicationContext, por sua vez, referencia um atributo protegido, dao, pertencente a Persistencia, no pacote 3. Entretanto, este caso de teste ainda possui a dependência que envolve a utilização de um arquivo XML, para a classe ApplicationContext conseguir instanciar a classe de Persistência. Neste caso, os demais métodos da classe ProjetoServerBO, podem ser testados usando o mesmo *stub* de ApplicationContext.

As alternativas que usam manipulação direta do BD e *setters* e *getters* públicos foram aplicadas nesse estudo de caso da mesma maneira do anterior.

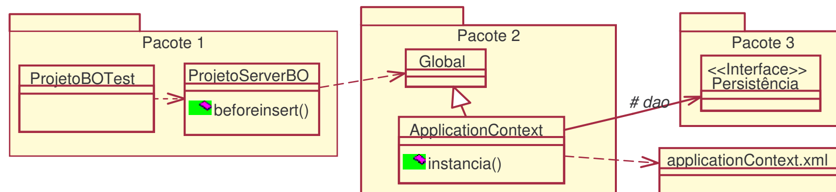

Fig 2. Diagrama de Classes parcial do DiSEN - Arquitetura 2

4. Comparação entre as Alternativas para Teste de Unidade

A Tabela 1 apresenta uma comparação entre as soluções apresentadas. Em aplicações inicialmente planejadas para utilizar acesso público, dentre as alternativas apresentadas, a solução que utiliza métodos *setters* e *getters* públicos seria a mais viável, pois as dependências podem ser alteradas em tempo de execução. O problema é que esta solução não é aplicável em muitos casos, como, por exemplo, em aplicações que utilizam o padrão *singleton* para métodos construtores privados, inviabilizando a alteração dinâmica. De uma forma geral, a única solução cuja aplicação não é viável ao DiSEN é a de *setter*/*getters* públicos.

Para software que utiliza arquivo XML, a injeção de dependência é a solução mais rápida e prática, porém o testador deve ficar atento para não esquecer de desfazer as alterações no arquivo XML ou usar um arquivo adicional para os testes. A solução que realiza as alterações diretamente no BD é também rápida e prática, contudo, é altamente suscetível a erro humano. A sua maior vantagem é a eliminação do uso de *stubs*, porque não é preciso simular os dados contidos no BD.

De forma geral e também para os estudos de caso do DiSEN, pode-se dizer que a solução que utiliza reflexão é a mais segura, pois depende somente da habilidade do testador e não exige alterações no código fonte. Além disso, ela é uma excelente solução para o teste de regressão porque os *stubs* podem ser reutilizados. No primeiro estudo de caso realizado foi necessário apenas um *stub* para cada unidade em teste. No segundo estudo de caso houve um trabalho adicional: o de escrever mais um *stub*, mas, ele serve para a realização do teste de todos os métodos de uma mesma classe.

Em relação à técnica de teste a ser empregada: funcional ou estrutural, pode-se afirmar que quando são usados componentes caixa-branca, como nesse trabalho, todas as alternativas são aplicáveis à técnica de teste estrutural. Por outro lado, para o teste funcional somente são viáveis as alternativas de injeção de dependência e de alteração direta no BD. Adicionalmente, o conhecimento das opções de *design* pode facilitar o trabalho de escrita dos casos de teste para as alternativas de injeção de dependência e de reflexão computacional e é indiferente para as outras duas.

Tabela 1. Comparação das soluções apresentadas

Solução / Característica	*Get/Setters* Públicos	Injeção de dependência	Alteração direta no BD	Reflexão
Alteração no código fonte	Sim	Apenas no arquivo XML	Apenas no BD	Não
Viabilidade de Aplicação no DiSEN	Não	Sim	Sim	Sim
Conhecimento necessário para a construção da classe de teste	Médio	Básico	Básico	Alto
Número de *stubs* necessários	0	1	0	1 (versão 1) 2 (versão 2)
Possibilidade de uso de técnica de teste funcional	Não	Sim	Sim	Não
Possibilidade de uso de técnica de teste estrutural	Sim	Sim	Sim	Sim

Todas as quatro alternativas podem ser aplicadas quando há dependências com qualquer do tipo de componente, caixa-branca ou caixa-preta. Isto porque: (i) na alternativa de alteração direta no BD não é necessário alterar código fonte, (ii) na alternativa de injeção de dependência as alterações ocorrem somente no arquivo de configuração independente do tipo de componente, e (iii) nas alternativas que utilizam reflexão computacional e *setters* e *getters* públicos é possível alterar partes internas do código a partir das classes de teste desde que os métodos do componente sejam bem documentados.

Adicionalmente, há outros trabalhos nos quais as alternativas apresentadas foram aplicadas. Por exemplo, o protocolo Guaraná [14] oferece recursos de instrumentação dinâmica via reflexão computacional por meio de uma implementação modificada da máquina virtual Java (JVM). A ferramenta KTEST [15] utiliza a tecnologia da reflexão computacional proporcionada pelo protocolo Guaraná[14] para viabilizar o teste baseado em estados de programas escritos em Java. O *framework* Spring [8] é um exemplo no qual o desenvolvimento de testes automatizados é feito por meio da alternativa de injeção de dependência.

5. Considerações Finais

Foram apresentadas alternativas para a realização de testes de unidade em software baseado em componentes nos quais somente o uso de *stubs* é insuficiente tendo em vista dependências criadas pelas decisões de projeto. A apresentação dessas alternativas contribui para com as comunidades científica e industrial oferecendo diretrizes para eliminar esse obstáculo e viabilizar a realização dos testes de unidade neste tipo de software. Com isso, procura-se atender a necessidade do mercado

oferecendo diretrizes para a realização dos testes de unidade apoiando-se nas disciplinas e técnicas recomendadas pela comunidade científica.

Também foram apresentadas a aplicação das alternativas em dois estudos de caso e uma breve comparação entre essas alternativas. De forma geral, pode-se dizer que, dentre elas, as alternativas que utilizam reflexão computacional e injeção de dependência são as mais seguras, podendo inclusive ter seus casos de teste reutilizados em testes de regressão. No caso do DiSEN, essas duas alternativas também foram as mais viáveis pelos motivos anteriormente citados.

Futuramente, pretende-se investigar a possibilidade de desenvolver uma ferramenta que automatize a realização de teste de unidade usando *stubs* e injeção de dependência e/ou reflexão computacional. Além disso, estão sendo planejados novos estudos de caso com o intuito de confirmar os resultados obtidos até o momento.

Referências

1. Wu, Y. Chen, M. Offut, J.: UML-based integration testing for component-based software. In: The 2nd International Conference on COTS-Based Software Systems (ICCBSS) (2003)
2. Szyperski, C. Component Software: Beyond Object-Oriented Programming. 2 ed. Addison-Wesley, Harlow, England, 624 p., 2002.
3. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. ISBN 0201633612. EUA: Addison-Wesley (1995)
4. Masiero, P.C., Lemos, O.A.L., Ferrari, F.C., Maldonado, J.C.: Teste de Software Orientado a Objetos e a Aspectos: Teoria e Prática. In: Breitman, K., Anido, R. (eds.) Atualizações em Informática, pp.13-72, Editora da PUC-Rio, SBC, Rio de Janeiro/RJ (2006)
5. Vincenzi, A.M.R., Delamaro, M.E., Wong, E., Maldonado, J.C. and Spoto, E.S.: Software baseado em componentes: uma revisão sobre teste. In: Desenvolvimento baseado em componentes: conceitos e técnicas. Rio de Janeiro: Editora Ciência Moderna. (2005)
6. Gazola, A., David, E.A.: Testes Unitários para camadas de negócios no mundo real. Revista Mundo Java. Editora Mundo. ISSN 1679-3978. Maio/Junho, 2007, n. 23, pp. 54-61. (2007)
7. May B., Sholar W.: Guia do desenvolvedor para BPEL Designer: Testando e depurando processos BPEL. http://www.netbeans.org/kb/55 /bpel_gsg_test_pt_BR.html. (2006)
8. Fowler, M.: Inversion of Control Containers and Dependency Injection Pattern. http://www.martinfowler.com/articles/injection.html
9. Atala, F. A.: Definição e Aplicação de Estratégias de Teste para Aplicações Distribuídas: Um Estudo de Caso. Monografia (Bacharelado em Informática), UEM, Maringá/PR (2007)
10. Gamma, E., Beck, K.: JUnit. http://www.junit.org/
11. Huzita, E.H.M.; Colanzi, T.E.; Tait, T.F.C.; Quinaia, M.A. Apoio à Cooperação, Persistência e Comunicação em um Ambiente de Desenvolvimento Distribuído de Software. INFOCOMP. Special Edition November 2008, pp. 71-80, Lavras/MG (2008).
12. Java-Sun. Core J2EE Patterns – Data Access Object. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html
13. Schiavoni, F.L.: FRADE – Framework para infra-estrutura de um Ambiente de Desenvolvimento Distribuído de Software. Dissertação (Mestrado), UEM, Maringá/PR (2007)
14. Oliva, A.; Buzato, L.E. Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java. Dissertação (Mestrado), UNICAMP, Campinas/SP (1998)
15. Silveira, F. F.; Price, A. M.de A. Ferramenta de apoio ao teste de aplicações Java baseada em Reflexão Computacional. Revista do CCEI, v.5, n.8, pp.32-40. Editora da URCAMP. Bagé/RS (2001)