Um Modelo para Construção de Componentes Testáveis

Camila Ribeiro Rocha[1], Eliane Martins

Instituto de Computação – Universidade Estadual de Campinas (Unicamp)
Caixa Postal 6.176 – 13.083-970 – Campinas – SP – Brazil
{camila.rocha, eliane}@ic.unicamp.br

Abstract. *A software component must be tested every time it is reused, to guarantee the quality of both the component itself and the system in which it is to be integrated. To reduce the costs of the test phase, this article proposes a model to build highly testable components, embedding monitoring mechanisms and assertions, besides an infrastructure that generates test cases from an UML specification. Our approach proposes the insertion of the mechanisms directly into the intermediate code, allowing the creation of testable COTS components.*

Resumo. *Um componente de software deve ser testado a cada reutilização para garantir tanto sua qualidade quanto a do sistema na qual é integrado. Para diminuir os custos da fase de testes, este artigo propõe um modelo para construção de um componente com alta testabilidade, no qual são embutidos mecanismos de monitoração e assertivas, além da disponibilização de uma infraestrutura capaz de gerar casos de teste automaticamente a partir de uma especificação em UML. Nossa abordagem propõe a inserção dos mecanismos diretamente no código intermediário, possibilitando a criação de componentes COTS testáveis.*

1. Introdução

O desenvolvimento de software baseado em componentes vem sendo cada vez mais utilizado atualmente. Seu principal atrativo é a possibilidade de redução do tempo e custo de desenvolvimento, através da reutilização de código. Os componentes podem ser produzidos pela própria equipe de desenvolvimento ou adquiridos de terceiros (os chamados COTS – *Commercial off the Shelf*).

A garantia da qualidade, porém, continua dependente da realização de testes. Um componente precisa ser testado tanto isoladamente quanto a cada reutilização [Weyuker 98], pois pode acontecer que um componente funcione bem em um determinado contexto mas não em outros. Por isso, é importante que o componente tenha uma alta testabilidade, para que a fase de testes não seja muito onerosa.

Foram propostas diversas técnicas para a construção de componentes testáveis [Hörnstein e Edler 2002, Ukuma 2002, Gao et. al. 2002]. Este artigo apresenta uma

[1] Camila Ribeiro Rocha recebe apoio financeiro da CAPES.

proposta de modelo para a construção de um componente testável. A estrutura deverá ser desenvolvida pelo fornecedor do componente e disponibilizada para seus usuários. O componente testável inclui mecanismos de monitoração e assertivas, além da geração automática de casos de teste a partir de sua especificação, permitindo a utilização da técnica de teste caixa preta.

Algumas destas abordagens introduzem mecanismos no componente que facilitam a fase de testes, embutindo esses mecanismos diretamente no código fonte [Hörnstein e Edler 2002, Ukuma 2002, Wang et. al. 1999]. Nossa abordagem propõe sua inserção no código intermediário, deixando separados os aspectos funcionais e os aspectos de testes e tornando opcional a permanência dos aspectos de teste no componente operacional. Além disso, a independência do código fonte desobriga o fornecedor de disponibilizá-lo para a inclusão dos mecanismos de teste, permitindo a distribuição do componente testável na forma de um COTS.

O texto está organizado da seguinte maneira: a seção 2 lista os problemas relacionados a testabilidade de componentes, e algumas abordagens para solucioná-los; a seção 3 apresenta a estrutura proposta para a construção de um componente testável; a seção 4 descreve aspectos considerados no projeto do componente testável; a seção 5 descreve sucintamente a implementação da estrutura em um estudo de caso simples, exemplificando a aplicação de um caso de teste; a seção 6 apresenta as conclusões e perspectivas futuras deste trabalho.

2. Testabilidade de Componentes

Segundo Szyperski [Szyperski 1998], “um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas, que pode ser desenvolvido independentemente e ser utilizado para a composição de sistemas de terceiros”. Um sistema baseado em componentes é composto por componentes que interagem entre si para fornecer as funcionalidades desejadas.

A principal vantagem da utilização de componentes é a reutilização de código, que proporciona redução no tempo e custo do desenvolvimento. A reutilização deve ser acompanhada por uma fase de testes. Contudo, tanto o fornecedor quanto o usuário de um componente COTS enfrentam dificuldades na realização dos testes, que podem ser caracterizadas como falta de conhecimento [Beydeda e Gruhn 2003]. Falta de conhecimento por parte do fornecedor, que não conhece todos os diferentes contextos de utilização de um componente e por isso assume algumas hipóteses sobre seu contexto de uso, que direcionam a realização dos testes. Esse direcionamento torna imprescindível que o usuário valide o componente no seu próprio contexto de utilização, que pode não ter sido priorizado pelo fornecedor.

Falta de conhecimento também por parte do usuário, que conhece apenas a interface pública do componente, através da qual é feito o acesso às funções oferecidas. O código fonte, seja para a geração de testes (em especial os de caixa branca), seja para a depuração após a realização dos testes, não é disponibilizado. Mesmo a realização de testes de caixa preta é prejudicada quando a documentação sobre o componente é incompleta.

Por isso, é muito importante que o componente seja testável, diminuindo o custo da fase de testes. O conceito de testabilidade é relativo à facilidade e ao custo de se encontrar falhas em um *software* [Binder 1994], e está ligado à sua capacidade de ser controlado e

observado durante os testes. Aspectos como métodos BIT (do inglês *Built-in Test*), que relatam o estado do componente, e assertivas, que definem pré e pós condições para métodos, contribuem para o aumento da testabilidade de um componente.

Vários trabalhos vêm sendo elaborados para minimizar os diferentes problemas citados. Dentre eles, as abordagens *Component+* [Hörnstein e Edler 2002], *Component Test Bench* [Bundell et. al. 2000] e *Testable Beans* [Gao et. al. 2002].

A abordagem *Component+* busca facilitar a realização de testes pelo cliente do componente, propondo uma estrutura para um componente testável formada por três subcomponentes:

- Componente a ser testado (*Component BIT*), que além da implementação das funcionalidades possui mecanismos BIT, para observação de seu estado;
- Componente *Tester*, que armazena casos de teste para o *Component BIT* e possui a capacidade de realizá-los e avaliá-los, utilizando as interfaces BIT;
- *Handlers*, utilizados para implementar mecanismos de tolerância a falhas.

A proposta de [Bundell et. al. 2000] é a ferramenta *Component Test Bench* (CTB), que auxilia os fornecedores na geração de casos de testes e os usuários na aplicação dos testes de forma automatizada. A ferramenta auxilia a construção de uma especificação em formato XML (*Extensible Markup Language*) contendo a descrição das implementações do componente e um conjunto de casos de teste para cada interface. Além disso, são disponibilizadas funcionalidades como a geração automática de oráculos (resultados esperados dos casos de teste). Na entrega ao usuário, o componente é empacotado juntamente com sua especificação de testes e um módulo da ferramenta, possibilitando a realização dos testes no novo ambiente.

Ambas as abordagens têm como vantagem dispensar uma infraestrutura para a realização dos testes, já que os artefatos são empacotados junto ao componente. Esses artefatos não constituem um *overhead* ao componente operacional, pois podem ser desacoplados. Outra vantagem é permitir ao cliente a customização dos casos de teste, com a disponibilização do código fonte do componente *Tester*, no caso da abordagem *Component+*, e da especificação de testes em formato XML, na CTB.

Entretanto, a customização na abordagem *Component+* pode ser prejudicada pela dificuldade no entendimento do comportamento do componente, já que a presença da especificação não é prevista pela arquitetura. Já a CTB prevê uma especificação que traz informações sobre as implementações e interfaces do componente. Uma limitação de ambas é que a redução no custo da fase de testes é pequena para o fornecedor, pois não há geração automática de casos de teste em nenhuma das abordagens (apesar da CTB oferecer mais facilidades).

Na abordagem proposta por [Gao et. al 2002], a principal preocupação é o aumento da testabilidade. É criado o conceito de *testable bean*, um componente de *software* que incorpora mecanismos padronizados que facilitam a realização de testes e a monitoração de sua execução.

Ao componente é incorporada uma interface de testes, que possui métodos para a inicialização, execução e avaliação dos casos de teste, além de fornecer informações sobre classes e métodos do componente utilizando reflexão computacional. Também é incorporada uma interface de monitoração, que permite o rastreamento de sua execução. A

principal desvantagem do método é não oferecer facilidades para a criação de casos de teste, sendo recomendado o uso de ferramentas especializadas.

A proposta aqui apresentada é uma extensão dos trabalhos [Toyota 2000, Ukuma 2002], em que um componente testável contém, além dos mecanismos embutidos de teste e monitoração presentes na abordagem anterior, um modelo de especificação a partir do qual é possível a geração automática de testes. Diferentemente da abordagem apresentada em [Toyota 2000], em que o componente era considerado como uma única classe, aqui um componente pode ser constituído de diversas classes. Nossa abordagem estende o trabalho de [Ukuma 2002] de modo a considerar notações UML para modelagem e a inclusão de mecanismos de monitoração no componente.

Em relação às abordagens apresentadas, nossa proposta une as vantagens das três: a modularidade da *Component+*, que separa os aspectos de testes das funcionalidades; o acoplamento da especificação ao componente, como na *Component Test Bench*; a presença de funções de monitoração, como no *Testable Bean*; a possibilidade do desacoplamento dos mecanismos de teste, apresentado nas três abordagens. Outra vantagem da nossa abordagem é a geração automática de casos de teste a partir da especificação e a possibilidade de customização dos casos de teste. Naturalmente que a geração necessita de uma infraestrutura auxiliar.

De acordo com Gao et al. a construção de componentes testáveis deve levar em conta os seguintes aspectos [Gao et. al. 2002]: 1º) definir uma arquitetura e uma interface de testes que seja comum a todos os componentes testáveis; 2º) estabelecer uma forma sistemática para a geração desses componentes; 3º) definir como minimizar o *overhead* dos mecanismos embutidos e 4º) oferecer recursos que apóiem os testes desses componentes. Neste texto focaremos os pontos 1 e 3.

3. Arquitetura

O objetivo da abordagem proposta é o aumento da testabilidade de um componente para a aplicação de testes caixa preta, realizados tanto pelo fornecedor quanto pelo usuário. Para isso, é acoplada ao componente, além de mecanismos de testes e monitoração, sua especificação na forma de assertivas e de um modelo de comportamento. As primeiras servem como oráculo e o segundo é usado por uma infraestrutura capaz de gerar casos de teste de forma automática.

A arquitetura do componente é ilustrada na Figura 1. O componente testável, além do componente a ser testado e suas interfaces públicas (“Interface A” e “Interface B”), é composto por dois outros subcomponentes:

- “Tracker”, responsável pelas funcionalidades de monitoração, que implementa as interfaces ITracking (pública) e ILogTrace (visível internamente ao pacote);
- “Tester”, responsável pela inclusão de assertivas e pela disponibilização da especificação, implementa as interfaces ITesting (pública) e IAClasse_metodo1 ... IAClasse_metodoN (visível internamente ao pacote). As interfaces IAClasse_metodo serão criadas de acordo com o número de métodos públicos oferecidos pelo componente. O componente Tester também armazena o arquivo contendo a especificação do componente, construída separadamente, com a utilização da UML (*Unified Modeling Language*).

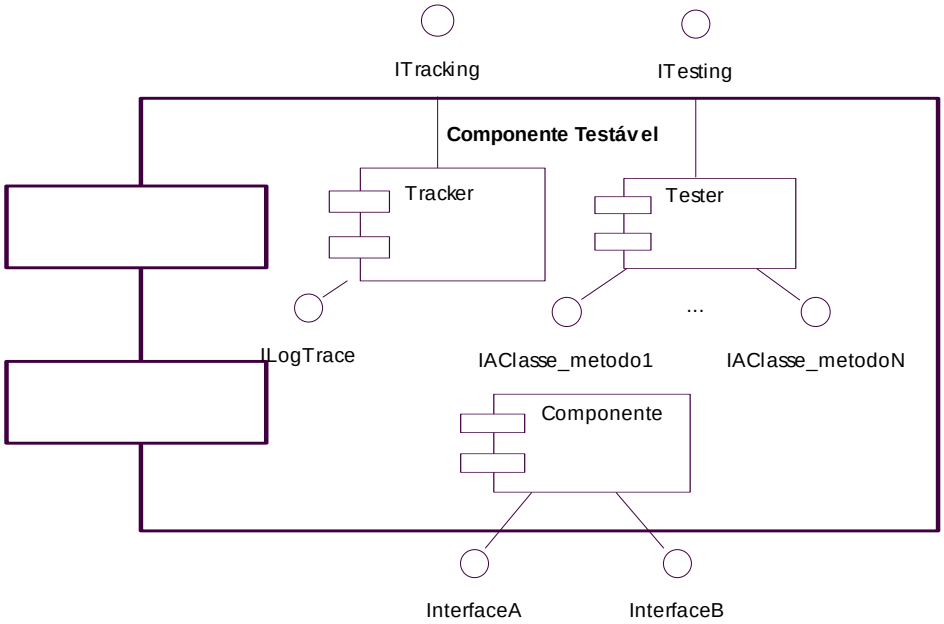

Figura 1. Estrutura interna do componente testável

3.1. Interfaces de Monitoração

Toda a interação com o componente testável é realizada através de interfaces. O subcomponente Tracker disponibiliza operações para a inserção de mecanismos de monitoração no componente. Os tipos de monitoração disponíveis são [Gao, Zhu and Shim 2000]: operacional, que rastreia as interações entre as operações do componente; de erros, que rastreia as mensagens de erro geradas durante a execução; e de estado, que monitora o valor dos atributos públicos do componente. A monitoração de performance não será disponibilizada inicialmente pois os testes serão voltados apenas para a verificação do correto comportamento do componente. A monitoração de eventos também não é considerada, já que é direcionada a um tipo específico de componentes (componentes GUI – para interface com o usuário). Entretanto, a arquitetura pode ser estendida para a incorporação desses tipos de monitoração.

A inclusão de mecanismos de monitoração é realizada no código intermediário do componente, através da interface ITracking. Após sua instrumentação, isto é, após a inclusão dos mecanismos, a execução do componente passa a ser acompanhada de acordo com o tipo de monitoração selecionado.

A interface ITracking disponibiliza as seguintes operações:

– `monitorOperacional(metodo : String, modo_chamada : char)`: o método passado como parâmetro será monitorado em relação as chamadas recebidas e realizadas (caso seu valor seja *null*, todos os métodos públicos serão rastreados). Caso o valor do parâmetro “modo_chamada” seja “e”, apenas as chamadas recebidas pelo método serão registradas no *log*; caso seja “s”, o registro será apenas das chamadas realizadas pelo método a outros métodos públicos do componente; caso seja “a”, todas serão registradas.

– `monitorEstado(atributos : String [])`: esse método habilita a monitoração dos atributos públicos do componente, que são passados como

parâmetro (caso “atributos” = *null*, todos os atributos públicos serão monitorados). O registro no *log* acontece a cada atualização.

- `monitorErro(excecao : String)`: esse método habilita a monitoração da exceção passada como parâmetro (caso “exceção” = *null*, todas as exceções serão monitoradas), registrando no *log* as informações sobre qual o tipo da exceção lançada, o método que a lançou, as classes que atravessou (através de *throw*), e a classe e método que a tratou.
- `setLog(arquivo: File)`: atribui o caminho do arquivo de *log*, onde as violações serão registradas.

A interface ILogTrace é visível apenas internamente. É invocada pelo componente sob teste após sua instrumentação, para o registro da monitoração. Disponibiliza as seguintes operações de registro no *log*:

- `incluirTraceErro(excecao : String, lançador : String)`: esse método registra o lançamento de uma exceção.
- `incluirTraceErro(excecao : String, tratador : String, sucesso : boolean)`: registra a tentativa de tratamento de uma exceção, e se o tratamento foi bem sucedido.
- `incluirTraceEstado(atributos : String [])`: registra o novo estado dos atributos.
- `incluirTraceOperacao(modo_chamada : char, chamado : String, chamador : String, parâmetros : String)`: registra a chamada de um método, além das informações sobre o método que o chamou, a hora que aconteceu a chamada, e os valores de parâmetros e de retorno.

3.2. Interfaces de Testes

O subcomponente Tester disponibiliza as interfaces ITesting, que oferece métodos para inclusão de assertivas no componente sob teste e para obtenção de sua especificação, e IAClasse_metodo1 ... IAClasse_metodoN, acessíveis apenas internamente ao componente testável e utilizadas pelo componente sob teste após sua instrumentação para verificação das assertivas. Cada interface IAClasse_metodo está relacionada a um método público do componente sob teste, sendo nomeada de acordo com a classe e método correspondentes. Por exemplo, se estiver relacionada ao método pop() da classe Pilha, será nomeada como “IAPilha_pop”.

A estrutura interna do componente Tester é ilustrada na Figura 2 e detalhada nas subseções a seguir. Foi incluída apenas uma interface IAClasse_metodo por simplicidade. O esteriótipo “+” indica visibilidade pública, e sua ausência visibilidade de pacote.

3.2.1 Interface ITesting

A interface ITesting disponibiliza as seguintes operações para inclusão de assertivas no componente sob teste: `incluirPreCondicao(metodo : String, continua: boolean)`, `incluirPosCondicao(metodo : String, continua: boolean)` e `incluirInvariante(metodo : String, continua: boolean)`. Os métodos, respectivamente, habilitam as pré-condições, pós-condições e invariantes do método passado como parâmetro (caso o valor de “método” seja *null*, as

assertivas são incluídas em todos os métodos para os quais foram especificadas). O parâmetro “continua” determina se a execução do sistema deve continuar após a violação de uma assertiva ou deve ser interrompida. Quando uma assertiva é violada, essa informação é registrada no *log*. O caminho para o arquivo de *log* é atribuído através da operação `setLog(arquivo: File)`, também oferecida pela interface ITesting.

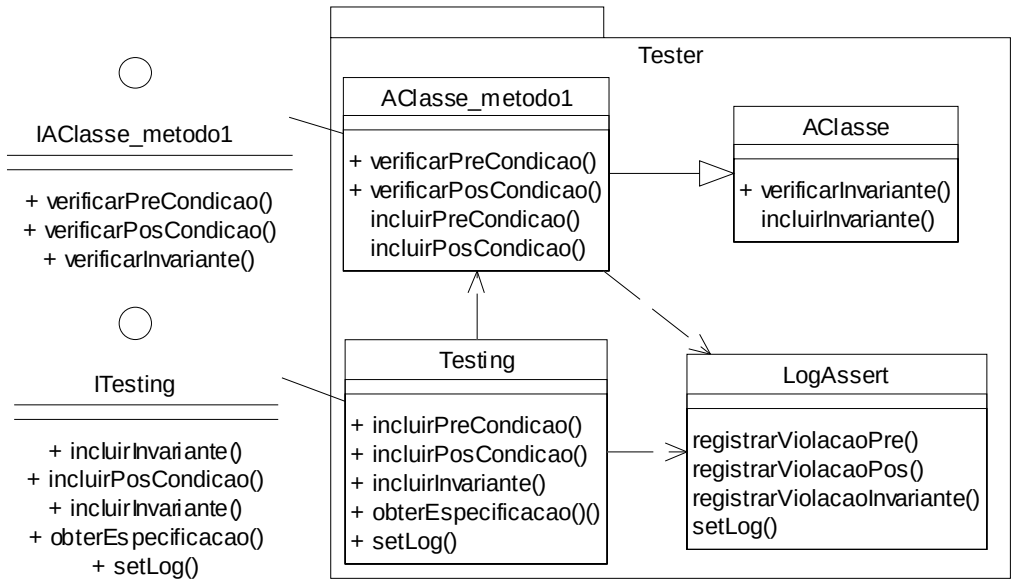

Figura 2. Estrutura interna do componente Tester

A interface ITesting é implementada pela classe Testing. As operações `incluirPreCondicao`, `incluirPosCondicao` e `incluirInvariante` delegam a inclusão dos mecanismos no componente sob teste à classe AClasse_metodo correspondente ao método solicitado. A operação `setLog` é repassada para a classe LogAssert.

Além dos métodos relativos às assertivas, a interface ITesting também oferece a operação `obterEspecificacao(): File`, que retorna o arquivo de especificação contido no componente Tester. Esse arquivo traz informações sobre a estrutura do componente, seu comportamento e as assertivas referentes a seus métodos públicos. O comportamento do componente é representado pelo diagrama de atividades da UML [Booch et. al. 2000], que contém informações sobre como os métodos interagem entre si, definindo em qual ordem eles devem ser chamados. As assertivas são registradas na linguagem OCL (*Object Constraint Language*) [Warner e Kleppe 1999], e originam as assertivas que serão embutidas no componente pelos métodos da interface ITesting.

A especificação será disponibilizada em forma de arquivo, e servirá de entrada para as ferramentas de geração automática de casos de teste e de tradução de assertivas em OCL para o formato a ser embutido no código dos métodos. A ferramenta ainda não foi construída, mas será baseada na ferramenta ConCAT, projetada por [Toyota 2000], que automatiza o processo de construção de um *driver* de teste para a classe a ser testada.

Um *driver* de teste é responsável por ativar o componente sobre teste, fornecendo as entradas e outras informações necessárias para a execução de casos de teste. A ferramenta ConCAT consiste em um *Driver Genérico* capaz de gerar um *driver* específico para o componente a partir de sua especificação [Toyota 2000]. O *driver* específico é um

conjunto executável de casos de teste, responsável por aplicar os testes e armazenar os resultados. São embutidas assertivas no código do componente que servem como oráculo para os casos de teste, isto é, determinam se uma saída obtida é correta para uma entrada determinada.

O *Driver Genérico* será capaz de obter um *driver* específico para o componente através do diagrama de atividades presente na especificação. Por gerar casos de teste a partir da especificação, a ferramenta dispensa a necessidade do código fonte para a geração dos casos de teste, e permite a aplicação de critérios de seleção de caminhos no diagrama de atividades. Além disso, como o código dos casos de teste estará disponível, o cliente poderá customizá-lo de acordo com suas necessidades.

O *Driver Genérico* não fica embutido no componente, diminuindo assim o espaço de armazenamento necessário. Apenas uma cópia deve ser mantida pelo cliente, e esta servirá para quaisquer componentes que tenham a especificação no formato exigido. Esse *driver* é independente, portanto, do componente em teste, e pretende-se que seja também independente do código em que o componente é escrito.

3.2.2 Interfaces IAClasse_metodo

As interfaces IAClasse_metodo devem ser construídas de acordo com os métodos públicos do componente sob teste. Suas operações são invocadas pelo componente sob teste após sua instrumentação, isto é, após a inclusão dos mecanismos para verificação de assertivas.

Cada método público do componente deve ter uma interface IAClasse_metodo correspondente, a qual oferece os métodos `verificarPreCondicao()`, `verificarPosCondicao()` e `verificarInvariante()`. Esses métodos são chamados pelo componente sob teste instrumentado durante sua execução, para verificar se a assertiva correspondente está sendo violada de acordo com os valores de seus parâmetros. Os parâmetros variam para cada interface, pois os métodos recebem os valores a serem verificados pelas assertivas.

As interfaces IAClasse_metodo são visíveis apenas internamente ao componente testável, e cada uma é implementada por uma classe AClasse_metodo, que por sua vez é descendente da classe AClasse. A superclasse AClasse é responsável pela implementação do método `verificarInvariante()`, já que a invariante de uma classe é a mesma para todos os seus métodos.

Caso os métodos `verificarPreCondicao()`, `verificarPosCondicao()` e `verificarInvariante()` verifiquem a violação da assertiva, o registro no *log* é realizado pelos métodos da classe LogAssert `registrarViolacaoPre (metodo : String, mensagem : String, continua : boolean)`, `registrarViolacaoPos (metodo : String, mensagem : String, continua : boolean)` e `registrarViolacaoInvariante (metodo : String, mensagem: String, continua : boolean)`. A violação é registrada no arquivo *log*, com o método no qual ocorreu e uma mensagem detalhando as causas do erro. O parâmetro "continua" indica se a execução deve continuar após a violação, e é determinado pelo método de inclusão de assertivas.

Além dos métodos de verificação de assertivas, as classes AClasse_método implementam os métodos `incluirPreCondicao()` e `incluirPosCondicao()`

(o método `incluirInvariante()` é implementado pela superclasse). Esses métodos são chamados pela classe Testing, e são responsáveis pela instrumentação do componente sob teste, incluindo respectivamente as pré, pós condições e invariantes no código intermediário do método correspondente.

4. Projeto Detalhado

A principal preocupação no projeto detalhado da estrutura do componente testável foi permitir que os mecanismos de monitoração e testes fossem desacoplados quando o componente entrasse em operação (inclusive a instrumentação inserida dentro do componente), já que eles serão utilizados apenas nas fases de teste.

Em [Gao, Zhu e Shim 2000] são apresentadas três maneiras para a inclusão de mecanismos de monitoração nos componentes (que podem ser estendidos para os mecanismos de teste): *Framework-based tracking* (monitoração baseada em *framework*), que orienta o fornecedor na adição dos mecanismos diretamente no código fonte; *Automatic code insertion* (inserção automática no código), na qual uma ferramenta adiciona os mecanismos ao código fonte do componente de forma automática; e *Automatic component wrapping* (empacotamento automático do componente), que adiciona mecanismos na forma de um *wrapper* para monitoração das interfaces externas do componente, sem necessidade do código fonte.

Como a terceira abordagem não é indicada para monitoração de erros, já que esta é altamente dependente das regras de negócio específicas da aplicação, a abordagem de inserção automática do código foi adaptada para componentes COTS, com a inclusão dos mecanismos no código intermediário do componente através de bibliotecas especializadas para sua manipulação. Outra opção para a inserção dos mecanismos seria a orientação a aspectos, usando, por exemplo, AspectJ [Kiczales et. al 2001], mas essa opção foi preterida pois seria necessário o acesso ao código fonte do componente.

A utilização de bibliotecas para manipulação do código intermediário tem como principal vantagem desobrigar o fornecedor a disponibilizar o código fonte do componente e ao mesmo tempo, oferecer ao usuário a opção de escolher os mecanismos a serem incluídos no componente para os testes e retirá-los quando preferir usar menos espaço para armazenamento. Essa inclusão pode ser realizada em tempo de carga, ou diretamente no código intermediário do componente, criando uma nova cópia instrumentada. Por ser um trabalho em andamento, a forma de inclusão ainda não foi definida.

5. Implementação

Os primeiros estudos de caso foram realizados em componentes implementados na linguagem Java. Após a compilação, é gerado um arquivo *.class* para cada classe Java. Os arquivos *.class* têm o formato de *byte code*, e são interpretados pela Máquina Virtual Java durante a execução. Assim, para a manipulação dos *byte codes* foi utilizada a biblioteca BCEL (do inglês *Byte Code Engineering Library*) [Dahm 2001].

A BCEL é uma biblioteca que permite a análise estática e a criação e modificação dinâmicas de arquivos *.class* Java. São disponibilizados vários tipos de operações, que auxiliam a modificação de uma classe [Dahm 2001]. É possível obter o código intermediário de método na forma de uma lista de instruções, que pode ser manipulada através de operações disponibilizadas pela biblioteca. A desvantagem da utilização da

biblioteca BCEL é sua complexidade, pois obriga o desenvolvedor a entender a estrutura e funcionamento do *byte codes*, mesmo que superficialmente.

Até o momento foram implementados apenas protótipos simples, mas que permitem comprovar a eficiência da biblioteca BCEL na inserção de assertivas e mecanismos de monitoração em um componente mesmo sem o código fonte, mostrando a viabilidade da arquitetura proposta.

O primeiro exemplo implementado foi o de uma Pilha de objetos do tipo *Object*. Por questão de espaço, não é possível apresentar sua especificação completa, nem detalhar sua instrumentação. Assim, serão descritos apenas os passos para a execução de um caso de teste, e qual foi o comportamento do componente instrumentado.

Como a ferramenta para a geração dos casos de teste ainda não foi implementada, foi construída a classe `Driver` para testar o comportamento do método `Pilha.pop()` quando a pilha está vazia. Para a realização do teste, o método foi instrumentado para inclusão de sua pós condição (o tamanho da pilha deve ter descrescido em uma unidade) e da invariante da classe (a pilha deve conter entre zero e cem elementos). A Figura 3 apresenta o código do caso de teste, que inicialmente requisita à interface ITesting a inclusão da pós condição e invariante, atribui o caminho do arquivo de *log*, e em seguida realiza o teste. Pode-se constatar o aumento da testabilidade do componente pois tanto para a instrumentação do componente quanto para a aplicação dos testes foi simplificada.

```
public class Driver {

  public static void main (String args[]) {
      Tester.ITesting teste = new Testing();
  //inclusão da pós condição do método pop
      teste.incluirPosCondicao ("pop", false);
  //inclusão da invariante no método pop
      teste.incluirInvariante ("pop", false);
      teste.setLog(“Pilha_vazia.log”);
      Componente.Pilha pilha = new Componente.Pilha();
      pilha.pop();
  }
}
```

Figura 3. Código fonte da classe Driver, que implementa um caso de teste

A invariante é verificada no início e no fim do método, mesmo que uma exceção seja lançada. Já a pós condição é verificada apenas ao final do método se nenhuma exceção tiver sido lançada. Como nenhum tipo de monitoração foi habilitado, a execução do caso de teste gera um *log* como ilustrado na Figura 4. Se outros casos de teste fossem executados seqüencialmente, as violações ocorridas nas assertivas continuariam sendo registradas no mesmo arquivo, assim como possíveis monitorações.

```
Invariante violada: Metodo Pilha.pop
Mensagem: Indice vetor = -1
```

Figura 4. Arquivo de *log* após a execução do caso de teste

6. Conclusões e Perspectivas

O objetivo principal deste trabalho é facilitar a realização de testes em componentes, diminuindo os custos da fase de testes. A testabilidade do componente é aumentada com a inclusão de assertivas e mecanismos de monitoração em seu código, além da disponibilização de uma infraestrutura para a geração automática de casos de teste a partir de uma especificação do comportamento do componente no formato do diagrama de atividades da UML.

A utilização da estrutura traz benefícios principalmente para o usuário do componente, que consegue testá-lo em um tempo reduzido sem prejudicar a eficiência do componente em tempo de operação. O fornecedor do componente também é beneficiado, já que os casos de teste são gerados de forma automática. A principal limitação do modelo é se tornar muito extenso caso o componente tenha um grande número de métodos públicos. O tamanho, porém, não é considerado um *overhead* em tempo de operação, já que todos os mecanismos podem ser desacoplados.

Como o trabalho está em fase inicial, as etapas realizadas foram a definição da arquitetura, de uma interface de testes e de uma maneira para minimizar o *overhead* dos mecanismos embutidos, através da inserção das assertivas e mecanismos de monitoração no código intermediário do componente. Foram implementados alguns protótipos simples, utilizando a linguagem Java e a biblioteca para manipulação de *byte codes* BCEL, que verificaram a viabilidade do modelo.

As próximas atividades do trabalho incluem a realização das duas outras etapas na construção de componentes testáveis [Gao et. al. 2002]: o estabelecimento de uma forma sistemática para a geração dos componentes testáveis e o oferecimento de recursos que apóiem os testes desses componentes. A geração dos componentes será sistematizada com a elaboração de um método para construção de componentes testáveis que possa ser acoplado a um método de desenvolvimento. Já os recursos de apoio serão formados por ferramentas que automatizem a geração e aplicação dos casos de teste e facilitem a criação dos componentes Tester e Tracker.

Referências

Beydeda, S. e Gruhn, V. (2003) “State of the art in testing components”. In: 3rd International Conference on Quality Software.

Booch, G., Rumbaugh, J. e Jacobson, I. (2000) UML: Guia do Usuário. Ed. Campus.

Bundell, G., Lee, G., Morris, J., Parker, K. (2000) “A Software Component Verification Tool”. In: Proceedings of International Conference on Software Methods and Tools.

Dahm, M. (2001) “Byte Code Engineering with the BCEL API”. Technical Report B-17-98, Freie Universität at Berlin, Institut für Informatik.

Gao, J., et. al. (2002) “On building testable software components”. In: Lecture Notes in Computer Science, V. 2255, pp. 108-121, Springer Verlag.

Gao, J., Zhu, E., e Shim, S. (2000) “Monitoring Software Components and Component-Based Software”. In: IEEE Computer Software and Application Conference (COMPSAC).

Hörnstein, J. e Edler, H. (2002) “Test reuse in CBSE Using Built-in Tests”. In: Workshop on Component-Based Software Engineering, Composing Systems from Components.

Kiczales, G., et. al. (2001) “An Overview of AspectJ”. In: Lecture Notes in Computer Science, V. 2072, p. 327-353, Springer Verlag.

Szyperski, C. (1998) Component Software: Beyond Object-Oriented Programming, Addison-Wesley.

Toyota, C. (2000) Melhoria da Testabilidade de Classes Usando o Conceito de Autoteste. Campinas: Instituto de Computação da UNICAMP. 119 p. (Dissertação, Mestrado em Ciência da Computação).

Ukuma, L. (2002) Uma Estratégia para o Desenvolvimento de Componentes de Software Autotestáveis. Campinas: Instituto de Computação da UNICAMP. 144 p. (Dissertação, Mestrado em Ciência da Computação).

Wang, Y., King, G. e Wickburg, H. (1999) “A Method for Built-in Tests in Component-based Software Maintenance”. In: IEEE International Conference on Software Maintenance and Reengineering (CSMR'99), p. 186-189.

Warmer, J e Klepper, A. (1999) “OCL: The constraint language of the UML”. Journal of Object-Oriented Programming, May.

Weyuker, E. (1998) “Testing Component-Based Software: A Cautionary Tale”, In: IEEE Software, 15(5): 54-59, September/October 1998.