Um Método de Testes para Componentes Tolerantes a Falhas

Camila Ribeiro Rocha

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Um Método de Testes para Componentes Tolerantes a Falhas

Camila Ribeiro Rocha

23 de novembro de 2005

Banca Examinadora:

- Prof. Dr. Eliane Martins (Orientadora)
- Prof. Dr. Flávio Moreira de Oliveira
  Faculdade de Informática – Pontifícia Universidade Católica do Rio Grande do Sul
- Prof. Dr. Cecília Mary Fischer Rubira
  Instituto de Computação – UNICAMP
- Prof. Dr. Maria Beatriz Felgar de Toledo
  Instituto de Computação – UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Rocha, Camila Ribeiro

R582m Um método de testes para componentes tolerantes a falhas / Camila Ribeiro Rocha -- Campinas, [S.P. :s.n.], 2005.

Orientadora : Eliane Martins

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Software – Testes. 2. Componentes de software. 3. Tolerância a falha (Computação) I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Um Método de Testes para Componentes Tolerantes a Falhas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Camila Ribeiro Rocha e aprovada pela Banca Examinadora.

Campinas, 23 de novembro de 2005.

Eliane Martins

Prof. Dr. Eliane Martins (Orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 23 de novembro de 2005, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. Flávio Moreira de Oliveira
PUC / RS

Profa. Dra. Cecilia Mary Fischer Rubira
IC - UNICAMP

Profa. Dra. Eliane Martins
IC - UNICAMP

Resumo

Componentes de *software* são cada vez mais utilizados no desenvolvimento de sistemas computacionais, incluindo sistemas críticos, dados os benefícios de redução de custo e tempo de desenvolvimento através da reutilização de código. A garantia da qualidade, porém, continua dependente da realização de testes a cada novo contexto, e é dificultada especialmente pela falta de conhecimento sobre o funcionamento do componente. O método proposto tem como foco principal componentes tolerantes a falhas, e busca facilitar a realização de testes caixa preta que exercitem tanto o comportamento normal quanto o comportamento excepcional do componente. O método é voltado para a construção de componentes testáveis, apresentando diretrizes para inclusão de mecanismos de monitoração e de verificação dos contratos dos componentes, mesmo sem a presença de código fonte, com a utilização de programação orientada a aspectos. Para a geração automática de *drivers* e *stubs*, são utilizados modelos comportamentais do componente, no formato do diagrama de atividade da linguagem UML. O método de testes pode ser usado paralelamente ao método de desenvolvimento. No trabalho é apresentado o uso do método em conjuto com o Método para Definição do Comportamento Excepcional (MDCE+), proposto em outra dissertação de mestrado do Instituto de Computação da Unicamp.

Abstract

Nowadays software components are widely used in software development, including critical systems, because of advantages such as time and cost reduction through code reuse. However, quality assurance, although, is still dependent on test execution at every new utilization context of the component, and usually faces difficulties specially related to lack of knowledge about component details. The test method proposed focuses fault-tolerant components, facilitating black-box testing of both normal and exceptional behavior. Aiming testable components development, the method proposed presents guidelines for inclusion of tracking and contract checking mechanisms, source code independently, using aspect-oriented programming. Drivers and stubs are automatically generated from component behavior models, in UML activity diagram form. The test method can be used together with a development method. In this work, the test method use is presented in parallel to Method for the Definition of Exceptional Behavior, proposed in a Master's thesis of the Institute of Computing at Unicamp.

Agradecimentos

Primordialmente, agradeço a Deus, por ter iluminado o meu caminho e me permitido chegar ao final com sucesso. Agradeço também aos meus pais pelo apoio, carinho e compreensão imprescindíveis.

Também tenho muito o que agradecer a minha orientadora Eliane Martins, pelos ensinamentos transmitidos e, principalmente, pela dedicação, apoio, amizade e confiança depositadas em mim.

Meus sinceros agradecimentos à prof. Cecília Rubira, Paulo Astério, Paulo Kato, Michelle e Denise, por possibilitarem a realização do estudo de caso em um ambiente real, contribuindo para o enriquecimento do trabalho. E um agradecimento especial ao colega Patrick Henrique, pela parceria neste estudo de caso e durante todo o mestrado.

Agradeço aos colegas do grupo de testes pelas valiosas sugestões, em especial à Regina Moraes.

Meu obrigada aos colegas do Laboratório de Sistemas Distribuídos Ulisses, Tomita, Patrick, Cláudio, Daniel e FábioBG pelas inúmeras ajudas, pelo companheirismo e pelas caravanas ao bandeijão (das quais não vou sentir tanta falta assim :).

Um agradecimento especial ao amigo Augusto, pelas revisões de artigos, scripts salvadores, cinemas alternativos e amizade sincera.

Agradeço também à turma da salinha de estudos, especialmente André, Nielsen, Bruno, Alexandro, FábioBG, Zé, Dani, Augusto e Patrick, pela ajuda para me livrar gloriosamente das disciplinas.

Agradeço à CAPES pelo apoio financeiro, e aos funcionários do IC pela estrutura essencial.

Meus agradecimentos especiais às colegas de república Thaisa, Silvania, Lorena, Ju e Mari, e aos vizinhos e amigos David, Wal e Miguel, pelo companheirismo e amizade.

Finalmente, um agradecimento especial a uma peça importantíssima da minha vida, meu namorado Guilherme. Sem a sua força e amor, teria sido tudo muito mais difícil. Obrigada por ser assim tão especial.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Alta qualidade a um baixo custo é uma exigência cada vez mais presente no desenvolvimento de sistemas computacionais. Para resolver esse impasse, o desenvolvimento baseado em componentes vem sendo atualmente adotado, por proporcionar reutilização de código e, conseqüentemente, redução no tempo e custo do desenvolvimento. Pela ampla aceitação, o desenvolvimento baseado em componentes vem sendo empregado inclusive para a construção de sistemas computacionais críticos, como dispositivos automotivos, por exemplo.

O desenvolvimento baseado em componentes propõe que o sistema seja estruturado em "unidades de composição com interfaces especificadas através de contratos e dependências de contexto explícitas" [70]. Esta estrutura favorece a reutilização, já que o acoplamento entre componentes é minimizado. É possível até mesmo a utilização de componentes produzidos por terceiros, os chamados componentes de prateleira, ou COTS (do inglês *Commercial off-the-shelf*). Neste caso, a economia do tempo de desenvolvimento é ainda maior.

A utilização de componentes proporciona baixo custo, mas não garante alta qualidade para a construção de sistemas confiáveis. Contrariamente à crença generalizada, componentes reutilizáveis devem ser amplamente testados, tanto isoladamente quanto a cada vez que são integrados a um sistema [84], pois um componente pode funcionar bem em um contexto e não em outro.

Testes são uma atividade de verificação e validação que exercita o sistema de *software*, fornecendo-lhe entradas em busca de falhas. Na grande maioria dos casos, não é possível testar um sistema exaustivamente, por isso é importante que os testes sejam planejados e executados para que permitam a descoberta do maior número possível de falhas.

Um dos fatores para a eficiência dos testes é a testabilidade do sistema. O conceito de testabilidade está ligado à facilidade da realização de testes [15] e, conseqüentemente, da descoberta de falhas em um sistema. É determinada por vários aspectos, sendo os

principais a observabilidade, que indica a capacidade de se determinar o quanto as entradas fornecidas afetam as saídas obtidas, e a controlabilidade, que consiste na capacidade de se produzir uma saída desejada a partir de uma entrada fornecida.

Para componentes reutilizáveis, a testabilidade é ainda mais importante. Além de serem testados a cada reutilização, a realização de testes nestes componentes apresenta algumas dificuldades adicionais, que podem ser caracterizadas como falta de conhecimento [11]: por um lado, o fornecedor não conhece todos os diferentes contextos de utilização do componente, assumindo algumas hipóteses para que possa desenvolvê-lo e testá-lo. Essas hipóteses podem levar a problemas de incompatibilidade com os sistemas nos quais o componente será utilizado, justificando a importância da realização de testes a cada integração em um novo sistema.

O cliente, por outro lado, tem acesso apenas à interface pública do componente, e enfrenta o problema da baixa testabilidade. A presença de especificações incompletas e o encapsulamento, que reduz a observabilidade e controlabilidade do componente, dificultam o entendimento sobre seu funcionamento, prejudicando a elaboração de casos de teste abrangentes e que exercitem tanto o comportamento normal quanto o excepcional. Para componentes desenvolvidos por terceiros, o código fonte, necessário para a geração de testes ou para a depuração após a realização dos testes, não é disponibilizado. Mesmo quando o é, o cliente pode não ter tempo ou conhecimento suficientes para entendê-lo e realizar as tarefas necessárias para testes e depuração.

A baixa testabilidade pode comprometer a economia de tempo obtida pela reutilização do código, já que a fase de testes se torna onerosa. A testabilidade também pode ser comprometida pela complexidade e tamanho do componente, já que um grande volume de testes terá que ser realizado. Para minimizar o problema, o ideal seria a reutilização dos testes realizados pelo desenvolvedor [14], evitando o retrabalho na elaboração dos casos de testes. A maioria dos desenvolvedores, porém, não incorpora informações sobre os testes realizados à especificação do componente, impossibilitando a reutilização.

Portanto, a testabilidade é um aspecto essencial para que um componente reutilizável alcance alta qualidade. Abordagens para a melhoria da testabilidade foram inicialmente introduzidas na área de *hardware* [46], na qual mecanismos adicionais eram incluídos no circuito para aumentar sua observabilidade e controlabilidade.

Na área de software, foram propostos diversos trabalhos que buscavam o aumento da testabilidade através de propostas como a melhoria da observabilidade e controlabilidade [40], melhoria do entendimento com a disponibilização da especificação [21, 61], auxílio na geração [10] e execução dos casos de teste [48, 75, 81].

Este trabalho é baseado no conceito de classes autotestáveis, apresentado em [74] e estendido para componentes por [76]. As classes autotestáveis incluem mecanismos auxiliares para as funções de teste, como aumento da observabilidade e controlabilidade. Além

disso, foi construído o protótipo de uma ferramenta para auxiliar a geração e execução de casos de teste, a ConCAT, escrita em C++.

O enfoque principal deste trabalho são componentes tolerantes a falhas, utilizados na construção de sistemas confiáveis. Tolerância a falhas é um conjunto de técnicas que visam manter o correto funcionamento do sistema mesmo na presença de falhas. Sua utilização acrescenta grande complexidade ao código do sistema [25], dificultando seu projeto, implementação e testes. A dificuldade de simulação das situações de falha aumenta ainda mais a complexidade na realização dos testes nestes sistemas.

O objetivo deste trabalho é a elaboração de um método para a melhoria da testabilidade de componentes reutilizáveis, em especial componentes tolerantes a falhas. A testabilidade é aumentada com a criação de artefatos de teste pelos desenvolvedores, facilitando a realização dos testes tanto pelos desenvolvedores quanto pelos clientes. Neste caso, os componentes geralmente são fornecidos totalmente fechados, dos quais nem a arquitetura interna nem o código fonte são conhecidos pelos clientes.

Por isso, para que os clientes também usufruam dos benefícios da testabilidade, o método proposto considera componentes independentemente de código fonte, atrelado a uma técnica de testes caixa preta (que não considera a estrutura interna), e voltado para testes de componentes isoladamente. O comportamento excepcional, que trata das situações de falha, recebe atenção especial, já que representa uma grande parte do código de componentes tolerantes a falhas. Neste trabalho, as três formas de melhoria são exploradas: aumento da observabilidade, disponibilização da especificação e auxílio na geração e execução dos testes.

A observabilidade do componente é aumentada com a inclusão de mecanismos para monitoração de sua execução e verificação de seu contrato [56]. Todo o projeto é baseado em UML (*Unified Modeling Language*), uma notação que tornou-se padrão para modelagem de sistemas computacionais. As especificações disponibilizadas aderem a este padrão, facilitando a utilização pelos desenvolvedores e testadores de *software*. A geração dos casos de teste é realizada a partir das especificações.

O método apresenta diretrizes para a criação dos artefatos de teste paralelamente ao processo de desenvolvimento. Neste trabalho, foi realizado paralelamente ao método de desenvolvimento para sistemas confiáveis MDCE+ [26] (Método para a Definição do Comportamento Excepcional), que é baseado no processo de desenvolvimento *UMLComponents* (Componentes UML) [22]. O acoplamento entre os métodos de desenvolvimento e testes foi validado em um estudo de caso realizado em um ambiente real.

1.1 Organização da Dissertação

Esta dissertação está organizada em oito capítulos da seguinte forma:

- **Capítulo 2 - Fundamentação Teórica**: Neste capítulo são apresentados os dois principais conceitos utilizados no trabalho: desenvolvimento baseado em componentes e testes de software.
- **Capítulo 3 - Testabilidade de Componentes**: Neste capítulo é descrito o conceito de testabilidade e apresentados trabalhos propostos para a melhoria da testabilidade de componentes, juntamente com uma análise comparativa com o trabalho proposto.
- **Capítulo 4 - Arquitetura do Componente Testável**: Neste capítulo é apresentada a arquitetura proposta para a melhoria da testabilidade de um componente, com o aumento de sua observabilidade e a disponibilização de suas especificações. Também são apresentadas algumas técnicas para a instrumentação de componentes sem a presença do código fonte, e a descrição de um estudo de caso simples, que será utilizado como exemplo para os Capítulos 4, 5 e 6.
- **Capítulo 5 - Projeto e Implementação dos Testes usando o Componente Testável**: Neste capítulo são descritos os formatos das especificações do componente e como os artefatos de teste podem ser obtidos a partir destas especificações. São descritas a especificação comportamental, que serve de base para a geração de *drivers* e *stubs*, e a especificação contratual, a base para a geração do código do contrato do componente, verificados durante a execução.
- **Capítulo 6 - Método para Construção de um Componente Testável**: Neste capítulo é apresentado o método de testes proposto. São descritos rapidamente os processos de desenvolvimento *UMLComponents* e MDCE+, que serviram de base para a paralelização do método de testes ao desenvolvimento, e também os passos do método de teste, voltados especialmente para a criação dos artefatos de teste.
- **Capítulo 7 - Estudo de Caso**: Neste capítulo são descritos os resultados obtidos em um estudo de caso realizado em um ambiente real, em paralelo ao processo de desenvolvimento dos componentes.
- **Capítulo 8 - Conclusões**: Neste capítulo são apresentadas as conclusões do trabalho, as principais contribuições e propostas para trabalhos futuros.
- **Apêndice A - Configuração do Componente Testável**: Este apêndice traz o código fonte das classes `Testing` e `Setup`, importantes para a criação da arquitetura do componente testável.
- **Apêndice B - Estrutura Lógica do XML dos Casos de Teste**: Este apêndice traz a estrutura a ser seguida para a construção dos casos de teste em XML.

Capítulo 2

Componentes e Testes

Este capítulo apresenta os principais conceitos utilizados neste trabalho: desenvolvimento baseado em componentes e testes de *software*. Inicialmente, na Seção 2.1 são apresentados os principais conceitos de desenvolvimento baseado em componentes: processos de desenvolvimento baseados em componentes, destacando os que serviram de base para este trabalho; contratos; e especificidades de um componente crítico. A Seção 2.2 apresenta os princípios de testes de *software*: estratégias, critérios e um processo genérico. A Seção 2.3 apresenta um sumário do capítulo.

2.1 Desenvolvimento Baseado em Componentes

O conceito de desenvolvimento baseado em componentes surgiu devido às pressões sofridas pela indústria de *software* por prazos mais curtos e produtos de maior qualidade. Ainda não há um consenso na literatura quanto ao conceito de componentes de *software* [20]. Neste trabalho, foi a adotada a visão de Szyperski: "um componente de *software* é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Um componente de *software* pode ser desenvolvido independentemente e ser utilizado para a composição de sistemas de terceiros" [70]. Componentes possuem interfaces bem definidas, tanto para suas dependências (interfaces requeridas) quanto para os serviços oferecidos (interfaces providas).

Um componente pode assumir vários formatos, cada um relativo a uma etapa do ciclo de desenvolvimento [22]:

1. *Padrão*: formato imposto ao componente para que ele se adapte a um determinado modelo de componentes, facilitando a posterior montagem da aplicação. Exemplos de padrões de componentes são OMG Corba [43] e Sun Enterprise JavaBeans [51].

2. *Especificação*: componente como abstração arquitetural, representando uma unidade de composição separada do restante do sistema, com funcionalidades e interfaces bem definidas. É parte da arquitetura e independente de implementação.

3. *Implementação*: componente que pode ser executado por um computador, podendo ser integrado a outro *software*. É o conceito utilizado para os componentes de prateleira, ou COTS.

Um sistema baseado em componentes é composto por componentes que interagem entre si para fornecer as funcionalidades desejadas. A Figura 2.1 ilustra a notação UML 2.0 [44] para um sistema baseado em componentes: as interfaces providas são representadas por círculos preenchidos, e as requeridas, por semicírculos. Uma conexão ilustra uma dependência entre os componentes.

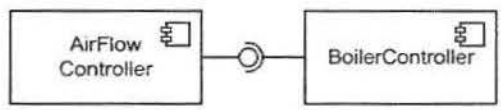

Figura 2.1: Notação UML 2.0 para um sistema baseado em componentes.

Os principais benefícios proporcionados pelo desenvolvimento baseado em componentes são a redução no tempo de desenvolvimento e a simplificação da manutenção, reduzindo o custo do sistema como um todo [47, 77]. A redução no tempo é proporcionada pela reutilização de código, diminuindo o tempo necessário às fases de projeto detalhado e implementação. A manutenção é simplificada pela facilidade da localização das mudanças, e por essas não afetarem outros componentes.

As subseções seguintes apresentam a abordagem *Design-by-contract* aplicada no contexto do desenvolvimento baseado em componentes; os componentes tolerantes a falhas, que implementam técnicas para recuperação na presença de falhas; e um processo de desenvolvimento genérico para sistemas baseados em componentes, seguido de descrições sucintas dos métodos de desenvolvimento utilizados como base deste trabalho.

2.1.1 *Design-by-contract*

O termo *Design-by-contract* foi introduzido no contexto de orientação a objetos [56], como uma maneira de melhorar a confiabilidade dos sistemas, contribuindo em sua corretude. O conceito *Design-by-contract* "representa os relacionamentos entre uma classe e seus clientes como um acordo formal, expressando os direitos e obrigações de ambas as partes" [56]. O mesmo conceito é utilizado no contexto de interfaces de componentes [70].

O contrato é especificado através de assertivas, que expressam "uma propriedade que algumas entidades do *software* têm que satisfazer em determinados pontos da execução" [56]. As propriedades devem cobrir o comportamento normal e excepcional esperado do componente, e podem ser relativas tanto a aspectos funcionais quanto não-funcionais do sistema (evitando-se, porém, referências à plataforma específica utilizada).

Um contrato é formado por três tipos de assertivas: invariante, pré-condição e pós-condição. Invariantes são propriedades que devem ser satisfeitas por todas as operações de um componente, antes e depois de sua execução; pré-condições são específicas de cada operação e devem ser satisfeitas pelo cliente para que a operação seja corretamente executada; pós-condições também são específicas de operações e são propriedades garantidas pelo componente após a execução da operação.

Contratos são um recurso simples e prático, porém limitado. Não é possível, por exemplo, no contexto de um componente caixa preta isolado, verificar se uma operação sem um valor de retorno foi corretamente executada, apenas se terminou normalmente. Outra limitação é a criação de contratos em sistemas assíncronos que trabalham com *callbacks*, já que o estado do sistema pode ser modificado no intervalo entre o registro do *callback* e sua execução [70].

Apesar das limitações, contratos são de grande valia para os testes. Experimentos comprovam que, mesmo pouco precisos, a presença de contratos facilita a descoberta e localização de falhas [18]. Os contratos podem ser utilizados como oráculos parciais, isto é, representando os resultados esperados dos casos de teste, possibilitanto economia de tempo na sua criação. Quando verificado em tempo de execução, uma violação do contrato manifesta a presença de falhas no sistema, seja no cliente (pré-condição violada) seja no componente (pós-condição ou invariante violada).

Além disso, os contratos contribuem para a melhoria da qualidade da documentação e para a diminuição da taxa de erros, pois aumentam o entendimento da equipe sobre a especificação do sistema. Eles também simplificam a implementação de mecanismos de tolerância a falhas, pois garantem que os parâmetros passados para as operações são corretos.

A UML possui uma linguagem especial que pode ser utilizada para a especificação de contratos, a *Object Constraint Language* (OCL) [82]. É uma linguagem formal, baseada em teoria de conjuntos e lógica de primeira ordem, que permite aos desenvolvedores especificar restrições relativas ao modelos de objetos em UML textualmente, de uma forma precisa e simplificada. Com OCL, é possível especificar os contratos de uma forma precisa ainda na fase de especificação do sistema, possibilitando a geração automática dos contratos a serem verificados em tempo de execução.

2.1.2 Componentes Tolerantes a Falhas

Um componente pode ser considerado crítico quando seu correto funcionamento é essencial para o funcionamento do sistema como um todo. Esses componentes devem receber atenção especial durante o desenvolvimento e, especialmente, na fase de testes.

Em sistemas considerados críticos, como controladores de transporte em massa por exemplo, falhas podem levar inclusive a perdas humanas. Por isso, é essencial que estes sistemas sejam confiáveis, isto é, mantenham seu correto funcionamento mesmo na presença de falhas [3]. Estas situações podem ser tratadas com a prevenção das falhas (com técnicas de teste, por exemplo), e com tolerância a falhas, que visam manter o funcionamento do sistema mesmo na presença de falhas.

Como técnicas para tolerância a falhas acrescentam muita complexidade ao código, é importante que o sistema seja estruturado de forma que a qualidade de sua arquitetura seja mantida. Para o contexto de desenvolvimento baseado em componentes, foi proposto o conceito de componente ideal tolerante a falhas, que pode ser utilizado tanto para a construção de componentes isolados, quanto para a estruturação de sistemas, construindo componentes recursivamente.

O conceito foi introduzido por Lee e Anderson [3] para denominar um componente tolerante a falhas cujo comportamento é categorizado em dois tipos: comportamento normal e comportamento excepcional (ou anormal). Com o comportamento normal, o componente executa adequadamente os serviços, produzindo respostas corretas. Entretanto, na ocorrência de alguma falha, o componente assume o comportamento excepcional, tentando tratar as falhas ocorridas.

A divisão do comportamento é refletida na sua estrutura, como ilustrado na Figura 2.2. A estrutura do componente ideal é dividida em duas partes: uma define o comportamento normal do componente, enquanto a outra é responsável pelo tratamento de exceções, implementando o comportamento excepcional [3].

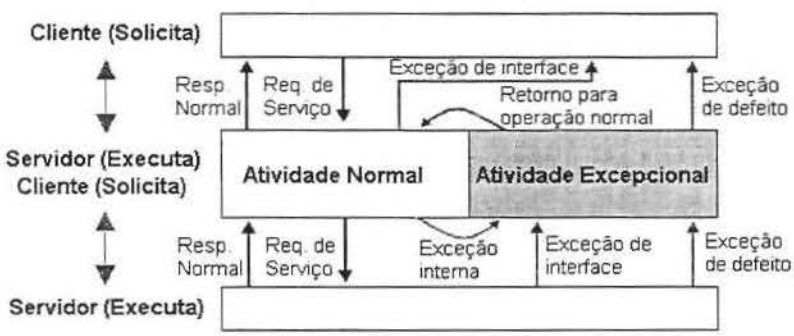

Figura 2.2: Componente ideal tolerante a falhas [36]

As exceções ocorridas em um componente ideal podem ser internas, externas (incapacidade de fornecer um serviço), ou de interface (requisições incorretas ou inexistentes) [36]. Na ocorrência de qualquer um dos tipos, o tratador de exceções interromperá a execução normal do componente e ativará o tratador específico para a exceção ocorrida. Caso não seja possível a execução do tratamento adequado, é retornada uma exceção de defeito às camadas superiores.

2.1.3 Processo de Desenvolvimento

Metodologias tradicionais de desenvolvimento de *software* não são adequadas para sistemas baseados em componentes [63]. Metodologias mais adequadas devem contemplar os dois aspectos desse tipo de desenvolvimento: a engenharia de domínio e a engenharia de aplicação. A engenharia de domínio guia a construção de componentes que possam ser reutilizados, enquanto a engenharia de aplicação determina as fases para um desenvolvimento *bottom-up*: seleção, adaptação e testes de componentes, além das fases tradicionais (análise, projeto, implementação e testes) [63].

Pressman sugere uma metodologia focada na engenharia de aplicação que começa com a fase de análise tradicional, seguida pelo projeto arquitetural do sistema. Após a definição da arquitetura, é iniciada a etapa de seleção dos componentes a serem incorporados a essa arquitetura, que podem ser adquiridos de uma biblioteca própria, de terceiros ou serem produzidos de acordo com a necessidade do novo *software*.

À seleção segue-se a fase de qualificação, na qual os componentes são examinados, testados e avaliados. Caso nesta fase sejam encontrados problemas em algum componente, é iniciada a fase de adaptação, na qual o componente é adaptado para as necessidades específicas do *software*. Ao término da fase de adaptação, acontece a fase de composição, na qual os componentes são integrados à arquitetura estabelecida para a aplicação.

Exemplos de metodologias para a engenharia da aplicação são Catalysis [33], Kobra [5] e *UMLComponents* [22]. Por ser a base do método de testes proposto, a metodologia *UMLComponents* é descrita na próxima subseção. Em seguida é descrito o método MDCE+, que adapta a metodologia *UMLComponents* para o contexto de sistemas confiáveis, e foi a metodologia de desenvolvimento utilizada durante o estudo de caso realizados.

UMLComponents

O *UMLComponents* [22] é um processo de desenvolvimento para sistemas baseados em componentes que utiliza diagramas da UML em suas especificações. É um processo simples, de fácil compreensão e utilização prática [26]. Por simplicidade, propõe a estruturação do sistema em quatro camadas: (1) interface de usuário, com as interfaces gráficas; (2) diálogo com a interface, que faz a ponte entre as telas e a lógica da aplicação;

(3) serviços do sistema, que implementa as funcionalidades; e (4) serviços de negócio, que armazena os dados da aplicação.

O processo *UMLComponents* guia o desenvolvimento das duas últimas camadas de forma iterativa, e é dividido em seis fases:

1. *Especificação de Requisitos*: desenvolvimento dos casos de uso, definindo os requisitos da aplicação; modelo do conceito do negócio (*Business Concept Model*), um esboço inicial das entidades básicas do sistema.

2. *Especificação dos Componentes*: fase mais importante do processo, na qual os componentes são especificados. É dividida em três subfases:

   (a) *Identificação dos Componentes*: identificação das interfaces providas dos componentes da camada de sistema a partir dos casos de uso; identificação dos componentes de negócio.

   (b) *Interação dos Componentes*: identificação das interfaces requeridas dos componentes de sistema e das interfaces providas dos componentes de negócio, a partir da criação de um diagrama de colaboração para cada operação das interfaces providas dos componentes de sistema.

   (c) *Especificação Final dos Componentes*: refatoração das interfaces de cada componente (divisão ou agrupamento), mantendo a coesão e evitando o excesso de interfaces; formalização opcional dos contratos das interfaces providas dos componentes de sistema.

3. *Provisionamento (Provisioning)*: implementação ou aquisição dos componentes especificados.

4. *Montagem*: integração dos componentes de acordo com a arquitetura especificada. Caso seja necessário, nesta fase são construídos conectores (Seção 4.1) para adaptações entre interfaces providas e requeridas.

5. *Testes*: aplicação dos testes de sistema.

6. *Instalação*: início da operação do sistema no ambiente do usuário.

Com as fases bem definidas, o processo facilita a utilização de componentes na construção de sistemas computacionais. Uma limitação, porém, é a pequena preocupação com a reutilização de componentes: apenas na fase de provisionamento a aquisição de componentes é considerada. Como as interfaces requeridas já foram especificadas, geralmente é necessário um uso maciço de conectores e invólucros (*wrappers*) para a adaptação das interfaces.

Outra grave limitação é a pouca importância dada aos testes. É prevista apenas a realização de testes de sistema, e mesmo para estes não são fornecidas indicações para sua aplicação.

MDCE+

O método MDCE+ [26] sistematiza a especificação e implementação do comportamento excepcional nas diversas fases do desenvolvimento de um sistema baseado em componentes. É um refinamento da MDCE [36], cujo enfoque eram as fases de especificação de requisitos e análise. O MDCE+ tem como enfoque as fases de projeto arquitetural e implementação.

Todos os componentes especificados adotam a estrutura do componente ideal tolerante a falhas (Seção 2.1.2) como forma de separação entre os comportamentos normal e excepcional. O método foi incorporado ao *UMLComponents* e, além das atividades propostas pelo *UMLComponents* para cada uma de suas fases, foram acrescentadas atividades específicas para determinação do comportamento excepcional pelo MDCE+:

1. *Especificação de Requisitos*: definição dos cenários excepcionais; especificação de pré, pós-condições e invariantes para cada caso de uso; classificação das entidades componentes do modelo conceitual do negócio em relação à criticidade, para escolha de quais terão mecanismos de tolerância a falhas implementados.

2. *Especificação dos Componentes*:

   (a) *Identificação dos Componentes*: juntamente com as interfaces providas dos componentes de sistema, identificação das exceções que podem ser lançadas por essas interfaces; criação de uma interface para cada exceção, encapsulando seus diversos tratadores; criação dos componentes ideais: cada componente normal é associado ao seu componente excepcional específico, que funciona como um conector entre o componente normal e os tratadores das suas possíveis exceções.

   (b) *Interação dos Componentes*: descoberta das exceções surgidas a partir das interações entre os componentes, as chamadas exceções arquiteturais, e especificação de seus tratadores. O diagrama de colaboração do *UMLComponents* foi substituído pelo diagrama de detalhamento (Seção 5.1.1), que possibilita tanto a descoberta das operações das interfaces requeridas quanto das exceções lançadas por estas interfaces.

   (c) *Especificação Final dos Componentes*: formalização opcional dos contratos definidos na fase de especificação de requisitos, e avaliação para a criação de conectores que tratem possíveis exceções através de conversão de tipos.

3. *Provisionamento (Provisioning)*: implementação/aquisição dos envólucros para adaptação dos componentes adquiridos, com a contextualização das exceções para as especificidades da aplicação.

4. *Montagem*: montagem dos componentes ideais, identificação do comportamento excepcional dos conectores e sua implementação.

O custo para a especificação detalhada das exceções e dos tratadores é compensado pela melhoria da qualidade no tratamento das situações excepcionais, com a distribuição das atividades durante todo o processo de desenvolvimento. Esta contextualização das exceções facilita a identificação, fazendo com que um maior número de situações excepcionais seja prevista e, conseqüentemente, aumentando a segurança no funcionamento (*dependability*), essencial para sistemas críticos.

2.2 Teste de *Software*

Nesta seção são apresentadas as as bases teóricas sobre testes de *software* utilizadas neste trabalho. Durante todo o texto será utilizada, para os termos falha, erro e defeito, a terminologia em português [54] que segue o padrão IEEE (IEEE STD. *Glossary of Software Engineering Terminology*, padrão 610.12/1990).

O padrão IEEE define que os problemas introduzidos no *software* pelo desenvolvedor são chamados de falhas (*fault*). Enganos podem ser cometidos tanto na especificação quanto no código do sistema. Quando uma falha é ativada durante a execução do *software*, um erro é gerado (*error*). Caso o problema se manifeste nas fronteiras do sistema, ocorre um defeito (*failure*), que pode ser percebido pelo usuário.

Seguindo a terminologia apresentada, teste de *software* é o processo de executar um sistema com o objetivo de encontrar defeitos [58]. Um caso de teste bem sucedido é aquele que revela uma falha no sistema que ainda não tinha sido descoberta. Os testes não são capazes de provar a ausência de falhas em um sistema: caso nenhum defeito ocorra durante os testes, isso não significa que o sistema não tenha falhas.

Não é possível testar cada caminho de execução: mesmo em sistemas triviais, o número de casos de teste geralmente é tão grande que inviabilizaria sua execução. Assim, é importante adotar critérios de seleção que produzam casos de teste com alta probabilidade de encontrar uma falha. Para a criação desses casos de teste, é recomendada a adoção de uma estratégia que possibilite sua criação em paralelo com o desenvolvimento.

As subseções seguintes descrevem: uma estratégia genérica de teste e suas fases; os principais critérios de teste utilizados; e um processo de teste genérico, que serve de base para o método proposto neste trabalho.

2.2.1 Estratégia de Teste

Uma estratégia de testes deve envolver tanto testes de baixo nível, que verificam se uma pequena parte do código fonte foi corretamente implementada, quanto testes de alto nível, que validam funções do sistema relativas aos requisitos do cliente. Pode ser dividida em quatro fases [9]:

1. *Teste de Unidade*: Uma unidade é a menor porção de um *software* que pode ser executada. O teste de unidade verifica se uma porção do código executa adequadamente a sua funcionalidade, isoladamente do resto do sistema. Por isso, geralmente o caso de teste faz uso de *drivers* e *stubs*. Um *driver* é um elemento (classe, programa principal ou *software* externo) que aplica os casos de teste à unidade sob teste [16]. O *driver* é responsável por fornecer os dados de entrada, coletar os dados de saída e apresentá-los ao usuário. Um *stub* substitui módulos necessários para a execução da unidade, simulando seu comportamento. A utilização de *drivers* e *stubs* possibilita que a unidade seja testada isoladamente.

2. *Teste de Componente*: Um componente é integração de diversas unidades, com interfaces bem definidas. Nesta fase o componente é testado de acordo com a especificação das funcionalidades e de sua estrutura. Também podem ser necessários *drivers* e *stubs*.

3. *Teste de Integração*: Nesta fase os componentes são integrados, e os casos de teste são direcionados à descoberta de erros arquiteturais, relacionados às interfaces dos componentes. As três principais abordagens de teste de integração são: *big bang*, na qual todos os componentes são integrados de uma só vez, dispensando o uso de *drivers* e *stubs* mas dificultando a localização de falhas; *top-down*, na qual os componentes são integrados a partir do bloco principal do programa, dispensando a construção de *drivers* mas fazendo um uso intensivo de *stubs*; e *bottom-up*, na qual os componentes são integrados a partir dos componentes independentes (que não possuem ligação com outros), dispensando o uso de *stubs* mas necessitando da construção de novos *drivers* a cada integração.

4. *Teste de Sistema*: Nesta fase todo o sistema é integrado, incluindo outros sistemas, *hardware*, etc. São testadas as interações do sistema com os subsistemas, além de aspectos funcionais e não-funcionais (desempenho, segurança e confiabilidade, por exemplo).

Além das cinco fases acima, existem ainda os testes de regressão, realizados durante a manutenção do sistema. Nesta fase um conjunto de casos de teste é novamente executado

com o objetivo de atestar que partes do sistema não foram afetadas pelas mudanças realizadas.

2.2.2 Critérios de Teste

Como não é possível testar exaustivamente um sistema, é importante a adoção de critérios que estabeleçam condições a serem satisfeitas durante os testes. Os três principais critérios são o critério funcional, o estrutural e o baseado em falhas.

No critério funcional de testes, também conhecido como "caixa-preta", os detalhes internos do sistema não são considerados na elaboração e execução dos casos de teste [9], apenas suas funcionalidades. São fornecidas entradas ao sistema, e suas saídas são verificadas de acordo com o comportamento especificado.

As fontes utilizadas pelos testes funcionais são as especificações de requisitos e de projeto. A principal dificuldade na realização de testes funcionais é a obtenção do menor conjunto de dados de teste de entrada para a obtenção do maior número de defeitos, a chamada eficácia dos testes [37]. Algumas abordagens para os testes funcionais são [63]:

1. *Baseado em Grafos*: o comportamento do sistema é descrito em forma de elementos e os relacionamentos entre eles, e o objetivo dos testes é a cobertura deste grafo, exercitando cada elemento e relação. Existem várias formas de representar o sistema como um grafo, como por exemplo, modelo de fluxo de transação e máquina de estados finitos [8]. Por ser uma técnica para geração de casos de teste, pode ser utilizada em conjunto com técnicas para geração de dados de entrada.

2. *Partição de Equivalência*: essa técnica é utilizada para a escolha de dados de entrada para os testes. O domínio de entrada é dividido em classes de equivalência, que representam um conjunto de valores válidos ou inválidos para as condições de entrada. Por exemplo, caso os dados válidos sejam representados por uma faixa de valores, três classes de equivalência são derivadas: os números menores que a faixa, os dentro da faixa, e os maiores que a faixa. Os valores de entrada dos testes são escolhidos dentro de cada classe de equivalência.

3. *Análise de Valores Limite*: essa técnica também é utilizada para a escolha de dados de entrada, combinada com a de partição de equivalência. Como um grande número de erros tende a ocorrer nos limites dos valores de entrada, a técnica de análise de valores limite determina que os dados de entrada escolhidos sejam valores limite de cada uma das classes de equivalência.

No critério estrutural, também conhecido como caixa branca ou caixa de vidro, considera-se a estrutura interna do sistema e sua implementação. Os casos de teste são derivados do código fonte do sistema, e exploram a cobertura do código para que os possíveis

defeitos sejam detectados. Algumas técnicas são [63]: teste baseado em caminhos, que busca executar caminhos de execução de maior complexidade dentro do código; teste de condições, no qual as condições lógicas do código são testadas; e teste baseado em dados, que seleciona os caminhos a serem testados de acordo com a localização das definições e usos de variáveis.

O critério baseado em falhas busca testar o comportamento do sistema na presença de falhas. Falhas podem acontecer tanto no próprio sistema sob teste, quanto nos sistemas com os quais ele interage. As principais técnicas são mutação, na qual falhas são introduzidas diretamente no programa, e injeção de falhas, modificando-se o estado do sistema durante sua execução.

2.2.3 Processo de Teste

Um processo de testes tem duas grandes atividades: preparação e realização dos testes [31]. Durante a preparação, são elaborados os planos e especificações de teste, além da implementação dos casos de teste. Durante a realização, os testes são executados e seus resultados analisados.

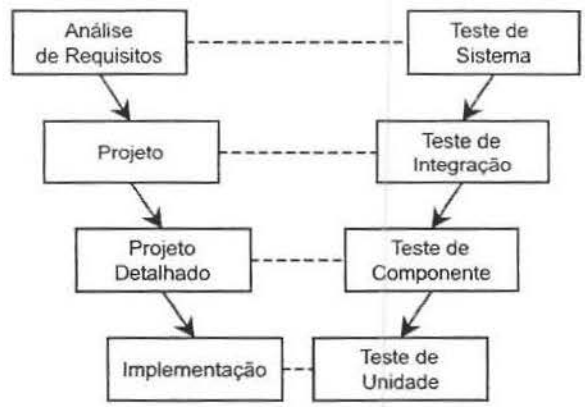

Figura 2.3: Paralelismo entre as fases do processo de desenvolvimento e da estratégia de teste.

Como apenas para a fase de realização é necessário que o sistema esteja implementado, a fase de preparação pode ser paralelizada com o desenvolvimento, como ilustra a Figura 2.3. O plano de testes de sistema é baseado na análise de requisitos, o plano de testes de integração, na arquitetura do sistema, e o de componentes, no projeto detalhado. Os testes de unidade geralmente são planejados e executados pelos próprios programadores durante a implementação [31].

Cada uma das fases da estratégia de testes segue um processo de teste, cujas fases são ilustradas na Figura 2.4 [32]. Apesar da utilização da notação do desenvolvimento em cascata, resultados de uma fase podem interferir na fase anterior. Cada uma das fases deve ser documentada: o conjunto de documentos definidos pela norma IEEE 829 [1, 16] são ilustrados na Figura 2.5, de acordo com as fases nas quais são confeccionados.

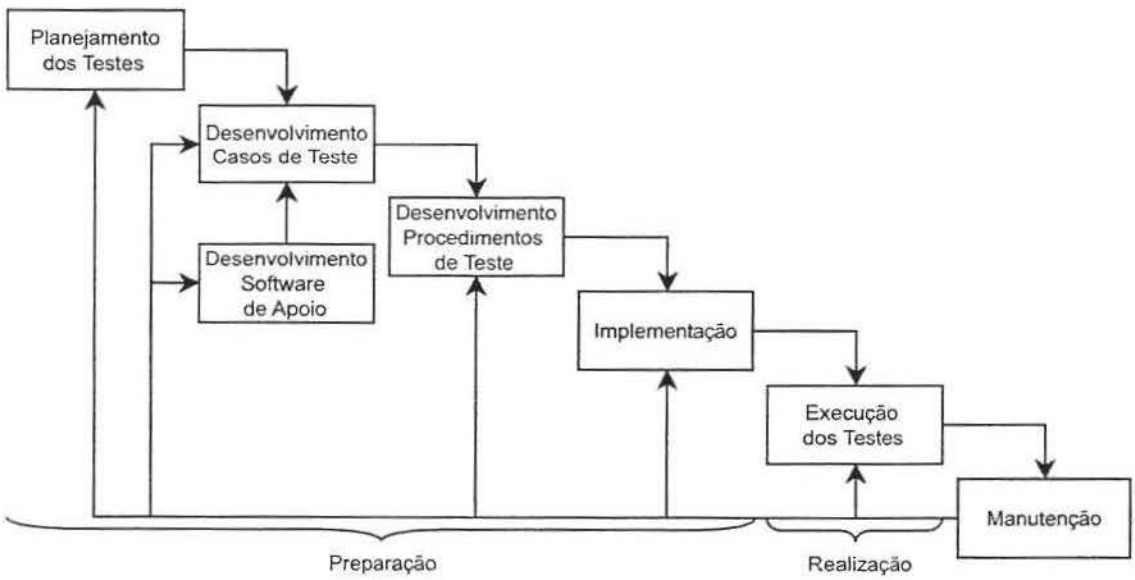

Figura 2.4: Fases do processo de testes.

A primeira fase é o planejamento dos testes, na qual são definidos quais serão os itens e aspectos a testar, as abordagens adotadas, os critérios de cobertura, os recursos necessários e cronogramas para a realização dos testes. Essas informações compõe o documento "Plano de teste".

O documento "Especificação de Projeto de Testes"é preenchido no início da segunda fase, "Desenvolvimento dos Casos de Teste". Apresenta um refinamento do plano de testes, detalhando as funcionalidades e características a serem testadas. Durante a segunda fase também é preenchido o documento "Especificação dos Casos de Testes", contendo as entradas e saídas esperadas para cada caso de teste, além das condições gerais para sua execução. Paralelamente, se necessário, são especificados *softwares* de apoio aos testes.

Na terceira fase, "Desenvolvimento dos Procedimentos de Testes", são descritos os passos para a execução de um conjunto de casos de teste, registrados no documento "Especificação dos Procedimentos de Testes". A fase de preparação é encerrada com a implementação dos casos de teste e *softwares* de apoio, e com a preparação do ambiente.

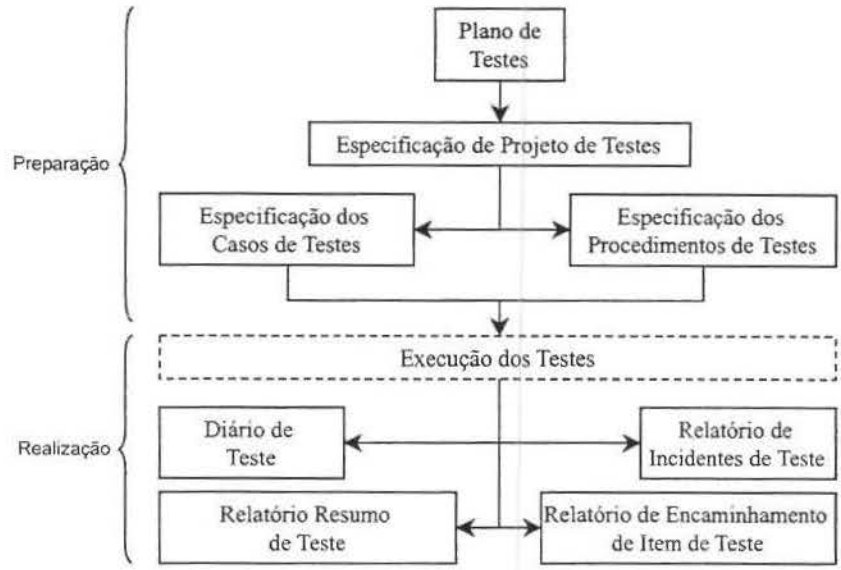

Figura 2.5: Documentos de teste propostos pela norma IEEE 829 [24].

A fase de realização é iniciada após a implementação do sistema ou partes deste. Nesta fase os casos de teste são executados, e são produzidos os seguintes documentos:

- Diário de teste, com os registros cronológicos da execução;
- Relatório resumo dos testes, com a descrição resumida das atividades de teste e uma avaliação dos resultados;
- Relatório de incidência de testes, registrando eventos ocorridos durante os testes que mereçam análise posterior;
- Relatório de encaminhamento do item de teste, encaminhando as correções para as equipes responsáveis. Apesar de a norma IEEE 829 considerar o preenchimento de documentos como a fase de registro, neste trabalho considerou-se a unificação das fases de execução e registro, já que os documentos são preenchidos paralelamente à execução dos testes.

2.3 Considerações Finais

Neste capítulo foram sucintamente apresentados os principais conceitos utilizados no trabalho: componentes e testes de *software*. Sobre componentes, foram apresentados os conceitos básicos; a abordagem *Design-by-Contract*, importante para o aumento da testabilidade; componentes tolerantes a falhas, os principais alvos do método de testes proposto; processos de desenvolvimento baseados em componentes, com destaque para o

método MDCE+, que propõe passos para a contrução de sistemas confiáveis formados por componentes tolerantes a falhas.

Neste capítulo também foram descritos o conceito de testes de *software*, suas estratégias, critérios e processo modelo. Na descrição de estratégias e critérios foram enfatizados a fase de teste de componentes e o critério funcional, enfoques deste trabalho. O processo de testes apresentado serve como base para o método de testes proposto.

O outro fundamento utilizado na pesquisa, testabilidade, é apresentado no próximo capítulo, juntamente com os trabalhos relacionados, fechando a preparação para a apresentação da proposta: uma arquitetura para melhoria da testabilidade (utilizando a abordagem *Design-by-contract* e a linguagem OCL) e um processo para criação dos mecanismos de teste, geração de casos de teste e aplicação destes.

Capítulo 3

Testabilidade de Componentes

A testabilidade de um sistema revela o grau de facilidade para testá-lo [15]. A construção de sistemas com boa testabilidade simplifica as operações de teste, reduz seus custos e aumenta a qualidade do software [39].

O conceito de testabilidade e os fatores que a influenciam são descritos na Seção 3.1, e na Seção 3.2 são apresentadas propostas para aumento da testabilidade de componentes. A Seção 3.3 apresenta uma análise comparativa entre os trabalhos estudados e o trabalho proposto. A Seção 3.4 apresenta um sumário do capítulo.

3.1 Conceito de Testabilidade

Testabilidade é a medida da probabilidade de ocorrer um defeito caso o software contenha falhas. Testes e testabilidade são complementares: os testes revelam as falhas, e a testabilidade sugere locais onde estas podem estar ocultas dos testes [79].

A testabilidade de um sistema é influenciada por várias características [63], sendo controlabilidade e observabilidade as mais influentes [15]. Controlabilidade é a capacidade de se produzir uma saída desejada a partir de uma entrada fornecida. Observabilidade indica a capacidade de se determinar o quanto as entradas afetam as saídas obtidas.

Vários fatores são considerados para a testabilidade de um componente, relativos tanto ao seu desenvolvimento quanto dos casos de teste e à execução destes. Binder apresenta os principais fatores a serem considerados na construção de um sistema testável [15]:

- *Representação*: A qualidade da documentação do sistema é crucial para sua testabilidade, já que os casos de teste são gerados a partir desta. É importante que a documentação seja mapeável à implementação e esteja sempre atualizada, para que os casos de teste tenham objetivos bem definidos e reflitam os requisitos reais do sistema.

- *Implementação*: Sistemas com componentes altamente coesos e pouco acoplados têm melhor testabilidade, pois facilitam a realização de testes de unidade e de componentes, e a localização das falhas. Fatores que contribuem para a baixa testabilidade são otimizações ligadas ao desempenho, concorrência e tratamento de exceções, por serem situações mais difíceis de serem simuladas em um ambiente de testes.

- *Capacidade de Teste Embutido (Built-in test)*: A testabilidade pode ser melhorada com a adição de métodos *set/reset*, para a definição de um estado para o sistema; métodos relatores, que permitem a observação do estado do sistema; e assertivas, que implementam a abordagem *Design-by-contract* (Seção 2.1.1). A inclusão de tais mecanismos contribui fortemente para a observabilidade e controlabilidade do sistema, mas seu impacto sobre o armazenamento e o desempenho do sistema deve ser considerado durante os testes. É importante que sejam habilitados/desabilitados automaticamente pelo compilador, e acessados através de interfaces específicas, separando-os das funcionalidades do componente.

- *Conjunto de Testes*: Um conjunto de testes adequado deve conter um plano de testes, projeto de teste, especificação dos casos de teste, procedimentos de teste, relatórios de aplicação e resultados e arquivos de *log*. Um conjunto de testes bem construído facilita principalmente a realização de testes de regressão, reduzindo os custos de manutenção.

- *Ferramentas de teste*: A automação dos testes é crucial para a testabilidade do sistema, já que permite a realização de um grande volume de testes a um custo reduzido.

- *Processo de desenvolvimento*: Além de contribuir com a qualidade do sistema e de sua documentação, um processo estruturado permite a construção dos casos de teste paralelamente ao desenvolvimento do sistema, gerando testes com maior cobertura e qualidade.

Neste trabalho, a testabilidade é considerada nas três fases: desenvolvimento do sistema, dos casos de teste e execução dos testes. A abordagem apresenta diretrizes para aumento da testabilidade de componentes nos aspectos de representação, capacidade de teste embutido, construção de conjuntos e ferramentas de teste, e processo de desenvolvimento. A próxima Seção apresenta diversas abordagens estudadas para aumento da testabilidade de componentes.

3.2 Abordagens para a construção de componentes testáveis

Esta seção apresenta algumas das várias abordagens propostas pela literatura para aumento da testabilidade de componentes. São descritos tanto trabalhos que serviram de base para a abordagem proposta (destacando-se [21, 40, 76]), quanto trabalhos mais atuais [35, 62, 69]. As subseções a seguir contêm as descrições e análise de algumas destas abordagens. Uma análise comparativa entre as abordagens e o trabalho proposto é apresentada ao final da Seção.

3.2.1 Componentes Autotestáveis

A abordagem foi proposta por Traon et. al [75] e é voltada para testes de unidade de componentes formados por apenas uma classe. Juntamente com a descrição dos passos necessários para a atividade de testes, é proposta a construção de componentes autotestáveis, compostos por uma especificação, um conjunto de casos de teste e a implementação. A especificação é o conjunto formado pela documentação do componente, a assinatura e as invariantes de seus métodos.

A abordagem *Design by contract* é utilizada para aumentar a observabilidade, sendo as assertivas utilizadas como oráculos para os casos de teste. O conjunto de casos de teste é dividido em unidades de teste, que testam um ou mais métodos separadamente. Tanto as assertivas quanto os casos de teste são inseridos no código fonte através de comentários com um formato pré-definido, e embutidos no código compilado por um pré-processador.

A análise da qualidade do componente é feita através da análise da qualidade de seus casos de teste, utilizando as técnicas de injeção de falhas e mutação para verificar se o conjunto de casos de teste consegue detectar os erros que foram inseridos. A limitação é que a análise é dependente da qualidade dos mutantes criados.

A abordagem contribui para a testabilidade do componente nos fatores de representação (disponibilização da especificação), capacidade de teste embutido (assertivas) e ferramentas de teste (disponibilização dos casos de teste). Uma de suas limitações é o fato de se restringir a apenas uma classe. Outra limitação é a inclusão de assertivas e casos de teste diretamente no código fonte, dificultando a manutenção. A inclusão no código fonte também limita a manipulação dos casos de teste pelo cliente, que tem acesso apenas à interface do componente.

3.2.2 Component+

Buscando facilitar a realização de testes pelo cliente do componente, Hörnstein e Hakan propõem a construção do *Component+* [48], um componente autotestável formado por três componentes: (1) *Component BIT*, que traz a implementação do componente sob teste instrumentada com mecanismos de teste embutido BIT, que podem ser utilizados através de uma ou mais interfaces BIT; (2) *Tester*, no qual os casos de teste estão implementados, que realiza os testes no componente BIT e avalia as informações fornecidas pelas interfaces BIT; (3) *Handlers*, utilizados para implementar mecanismos de tolerância a falhas.

As principais vantagens da abordagem são: não necessitar de ferramentas adicionais; possibilitar o desacoplamento dos casos de teste, permitindo que o componente em operação ocupe menos espaço de armazenamento; a preocupação explícita com o comportamento excepcional; a possibilidade de customização dos testes, caso o código fonte do componente *Tester* seja fornecido. Como desvantagem, a abordagem não facilita o entendimento do funcionamento do componente por parte do cliente, já que não disponibiliza sua especificação. Caso o código fonte dos testes não seja disponibilizado, o cliente perde a possibildade de adaptar os casos de teste a sua realidade.

3.2.3 Component Test Bench

Bundell et. al apresentam uma ferramenta de testes chamada *Component Test Bench* (CTB) [21], que permite aos desenvolvedores a geração de casos de testes, e aos clientes a verificação do componente em seu ambiente de trabalho.

A ferramenta contém um editor para a especificação de testes do componente, contendo a descrição de suas implementações, as interfaces de cada implementação e um conjunto de testes apropriado para cada uma das interfaces. O formato definido para as especificações é o XML (*Extensible Markup Language*). A ferramenta CTB também oferece para o desenvolvedor do componente a geração de oráculos, a cobertura dos testes e o histórico da execução das diferentes versões de um componente. Quando o componente é entregue ao cliente, a especificação de testes e um módulo da ferramenta CTB são empacotados junto ao componente, para que o cliente possa repetir a execução dos testes em seu ambiente.

A ferramenta agiliza a execução dos testes do componente, e a utilização do formato XML torna possível a customizacão dos casos de teste por parte do cliente. Entretanto essa customização pode ser prejudicada, já que não é fornecido outro tipo de especificação que facilite o entendimento do componente. Para o desenvolvedor, há a desvantagem da necessidade de codificação manual dos casos de teste.

3.2.4 Metadata

A abordagem constitui-se na adição de diferentes tipos de metadados ao componente [61], melhorando sua testabilidade com o aumento da qualidade de sua documentação. Para isso, é proposto um *framework* que permita ao cliente do componente explorar os metadados disponíveis, que podem estar tanto encapsulados com o componente quanto disponíveis remotamente. O formato proposto para os metadados é o XML, sendo que cada tipo diferente teria uma DTD (*Document Type Definition*) associada.

A solução proposta tem a vantagem de ser versátil, podendo ser adaptada para a inclusão de qualquer tipo de informação no componente. A desvantagem é que a versatilidade também dificulta a padronização de um estilo único para cada tipo de documento.

3.2.5 *Framework* para Testes Caixa Preta de Componentes

A abordagem de Edwards [34] propõe o aumento da testabilidade através tanto da inclusão de mecanismos BIT quanto da geração automática de casos de teste. Os mecanismos BIT compreendem a verificação do contrato do componente, através da construção de um *wrapper* que intercepta a chamada das operações, o qual é gerado automaticamente a partir da especificação formal do componente na linguagem RESOLVE [67]. Os casos de teste e seus dados são gerados a partir de um grafo de fluxo que modela o comportamento do componente.

A geração de casos e dados de teste, apesar de limitada, é a grande vantagem da abordagem. As principais desvantagens são a utilização de uma linguagem pouco conhecida, e a utilização de *wrappers*, que pode ter grande impacto no tempo de execução do componente durante os testes.

3.2.6 SPACES

A ferramenta SPACES (*SPecification bAsed Component tESter*) foi construída por Barbosa et. al [6] para dar apoio à execução do método de teste funcional de componentes proposto por Farias [35]. O método propõe a geração de casos de teste a partir de artefatos gerados durante o desenvolvimento: a partir dos casos de uso são atribuídas prioridades às funcionalidades a serem testadas; os diferentes cenários de uso são interligados na forma de um diagrama de estados, cujas arestas possuem pesos para priorização dos caminhos de execução; cada cenário é descrito em um diagrama de seqüência UML, composto pelas mensagens trocadas pelas classes do componente para a realização do cenário. As pré e pós-condições de cada cenário são definidas na forma de expressões OCL.

Os casos de teste são derivados, com auxílio da SPACES, dos caminhos de maior prioridade do diagrama de estados e seqüência. As pós-condições, verificadas em tempo

de execução, servem como oráculos dos testes. Todo o método é executado em paralelo ao método de desenvolvimento *UMLComponents* [22], e os artefatos produzidos são empacotados junto ao componente.

A principal vantagem do método é a possibilidade de priorizar cenários, especialmente útil em componentes muito complexos e em testes de regressão. A disponibilização dos modelos também auxilia o entendimento do componente, porém é limitado a componentes implementados em linguagens orientadas a objetos e possui um aspecto caixa branca (apresentando o comportamento interno do componente) que pode não ser útil para o cliente do componente.

3.2.7 STECC

A estratégia STECC (do inglês *Self-Testing Cots Components*) [12] tem como idéia básica a inclusão no componente de funcionalidades similares às de ferramentas de análise e testes, gerando casos de teste caixa branca e caixa preta. É encapsulado ao componente um grafo que ilustra tanto o fluxo de controle interno de cada método, quanto o fluxo de controle entre os métodos, isto é, a especificação caixa preta do componente. A especificação é utilizada pelo próprio componente (através de suas funcionalidades de autoteste) para a geração de casos de teste.

Estes casos de teste podem ser parametrizados pelo usuário, que pode escolher entre as estratégias caixa preta e caixa branca conforme sua necessidade. A construção do *driver* de testes, a execução e a avaliação dos casos de teste são automatizadas. A vantagem da estratégia, além da geração automática de casos de teste, é proporcionar um grande poder ao usuário para a customização dos casos de teste que serão executados, através das ferramentas oferecidas, sem a necessidade de disponibilização do código fonte. A principal desvantagem é o grande aumento do custo da produção do componente, com a construção de vários tipos de documentações, todas com alto nível de detalhes.

3.2.8 Testable Beans

Nesta abordagem é criado o conceito de *testable bean* [40], uma arquitetura para um componente de software que, além das interfaces providas para a execução das funcionalidades do componente, possui uma interface de testes e uma de monitoração. A interface de testes possui métodos para preparação, execução e avaliação do resultado dos testes, além de métodos para obtenção de informações sobre as classes e métodos do *testable bean*. A interface de monitoração possui métodos para habilitar diversos tipos de monitoração.

A abordagem contribui com a capacidade de teste embutido, melhorando a observabiliade do componente sem afetar a estrutura de sua implementação. A principal vantagem é a modularização promovida, que facilita tanto manutenção do componente quanto das

novas funcionalidades, e permite a retirada de mecanismos de teste e monitoração quando o componente estiver em operação. No entanto, as informações fornecidas podem ser insuficientes para a construção de casos de teste de qualidade pelo cliente, já que não são disponibilizadas informações sobre o funcionamento do componente.

3.2.9 ConCAT

A abordagem propõe a construção de componentes autotestáveis [74, 76], com a inserção no componente de métodos relatores, assertivas e sua especificação. Os métodos relatores são disponibilizados através de uma interface de testes, que pode ser desabilitada (assim como as assertivas) através de opções de compilação.

Os casos de teste são gerados pela ferramenta ConCAT, a partir da especificação do componente. A ConCAT recebe como entrada a especificação (no formato de Modelo de Fluxo de Transação [8]) e gera automaticamente um *driver* de testes específico para o componente, utilizando a abordagem caixa preta. Esse *driver* é construído em linguagem C++ e prepara, executa e avalia os casos de teste, utilizando os métodos relatores e as assertivas embutidas no código.

A testabilidade do componente é melhorada através de sua representação, da capacidade de teste embutido, do conjunto e das ferramentas de teste. As desvantagens da abordagem são a geração do *driver* apenas na linguagem C++, restringindo a possibilidade de utilização para outras linguagens; a necessidade de compilação para inclusão dos mecanismos de teste; a limitação para componentes de apenas uma classe.

3.2.10 Testes baseados em Cenários

Nesta técnica, proposta por Strembeck e Zdun [69], os casos de teste são gerados manualmente a partir dos casos de uso do componente, e os procedimentos de teste, a partir dos cenários normais e excepcionais de cada caso de uso. Todos os artefatos (casos de uso e casos de teste) são traduzidos para a linguagem XOTcl [59], e armazenados junto ao componente como metadados, permitindo o rastreamento entre os casos de uso e os casos de teste. A partir dos casos de teste em XOTcl, são gerados automaticamente os *drivers* de teste na linguagem de programação do componente.

A principal vantagem desta técnica é relativa a testes de regressão, já que a definição de seu escopo é realizada com base no rastreamento entre casos de uso e casos de teste. A presença dos casos de uso como metadados do componente também é vantajosa por melhorar o entendimento sobre o componente. As desvantagens são a geração manual dos casos de teste em XOTcl e a própria utilização da XOTcl para construção dos modelos pois, apesar de sua sintaxe ser simples, não é largamente conhecida.

3.3 Análise Comparativa

A análise das abordagens foi baseada nos fatores de testabilidade apresentados por Binder (Seção 3.1). A tabela 3.1 apresenta um sumário da avaliação.

Fatores de Testabilidade	Representação	Implementação	BIT	Conjunto de Testes	Ferramentas	Processo
Comp. Autotestáveis	•		•	•		
Component+		•	•	•		
Component Test Bench				•	•	
Metadata	•					
Framework	•		•	•	•	
SPACES	•		•	•	•	•
STECC				•	•	
Testable Beans			•			
ConCAT	•		•	•	•	
Testes baseados em Cenários	•			•	•	
Abordagem Proposta	•		•	•	•	•

Tabela 3.1: Tabela comparativa das abordagens para melhoria da testabilidade

As abordagens que disponibilizam as especificações do CST foram consideradas no aspecto *Representação*, por facilitarem tanto a geração dos testes quanto o entendimento do funcionamento do componente pelo usuário. A abordagem proposta também foi enquadrada no aspecto *Representação* pois são disponibilizadas as especificações comportamentais e contratuais do componente, utilizando a linguagem UML.

O aspecto Implementação é considerado apenas pela abordagem Component+, que propõe uma arquitetura para a implementação do componente. Nesta arquitetura há a separação do componente em implementação normal, tratadores de exceção e casos de teste. Nenhuma das outras abordagens propõe formas de implementação para aumento da testabilidade. A abordagem proposta, inclusive, nem mesmo considera a forma que o componente é implementado: o CST é considerado como uma caixa preta. A vantagem é que a proposta é adaptável a qualquer tipo de componente.

O aspecto BIT é considerado pela maioria das abordagens, de diferentes maneiras: inclusão de mecanismos *set/reset* (*Component+*, *Testable Beans*); inclusão de métodos relatores (*Component+*, ConCAT); verificação do contrato do componente em tempo

de execução (Componentes Autotestáveis, *Framework*, SPACES, Abordagem proposta); inclusão de mecanismos de monitoração (*Testable Beans*, Abordagem proposta).

O aspecto Conjunto de Teste é aprimorado pela maioria dos métodos, de diferentes maneiras: alguns, como *Component+* e Componentes Autotestáveis, disponibilizam os casos de teste já prontos na linguagem de programação em questão, para que possam ser reaplicados pelo cliente. Apesar de trazer ao cliente a vantagem de reaproveitamento dos casos de teste já desenvolvidos, as abordagens possuem a limitação da necessidade de disponibilização do código fonte dos testes, no caso da *Component+*, e do componente, no caso da abordagem Componentes Autotestáveis, para que o cliente possa customizar os casos de teste, adaptando-os para sua realidade.

Outras abordagens, como a *Component Test Bench* e a de Testes Baseados em Cenários, disponibilizam os casos de teste em uma linguagem intermediária, possibilitando sua customização pelo cliente. Estas abordagens também contribuem para a testabilidade no aspecto *Ferramentas*, pois disponibilizam tradutores para conversão dos casos de teste na linguagem intermediária para a linguagem de programação em questão, gerando os *drivers* de teste automaticamente. Os casos de teste, entretanto, continuam sendo manualmente gerados, ainda que em uma linguagem intermediária.

As demais abordagens (*Framework*, STECC, SPACES, Abordagem proposta) que trabalham o aspecto *Conjunto de Teste* possibilitam a geração automática de casos de teste a partir da especificação do componente, unindo os aspectos *Representação* e *Ferramentas*. A abordagem proposta trata, inclusive, da geração automática de *stubs* e de mecanismos BIT a partir das especificações do componente.

O último aspecto da testabilidade, processo, é tratado apenas pelas abordagens SPACES e a proposta neste trabalho. Ambas descrevem os passos para realização dos testes em paralelo a um método de desenvolvimento, guiando inclusive a construção de especificações do CST durante o desenvolvimento, com a diferença que a abordagem SPACES também considera o comportamento interno do componente para os testes. A abordagem proposta, porém, tem as vantagens de maior acoplamento com o método de desenvolvimento, possibilitando inclusive o aproveitamento das especificações tanto para o desenvolvimento quanto para os testes. Além disso, por ser totalmente caixa preta, facilita a reutilização dos casos de teste pelo usuário do componente.

Além dos pontos citados, a abordagem proposta tem a vantagem de fornecer diretrizes específicas para os testes do comportamento excepcional do componente, auxiliando nos testes de componentes tolerantes a falhas. Entre as outras abordagens estudadas, apenas a abordagem Component+ considera componentes tolerantes a falhas, e mesmo assim, fornece diretrizes apenas para sua implementação, não para os testes dos mecanismos.

3.4 Considerações Finais

Este capítulo apresentou o conceito de testabilidade de *software* e os fatores que a influenciam. Além disso, foi realizada uma revisão bibliográfica de diversos trabalhos sobre o assunto, especialmente os que serviram de base para esta dissertação.

O próximo capítulo inicia a descrição do método de testes proposto, com a apresentação da arquitetura do componente testável.

Capítulo 4

Arquitetura Proposta do Componente Testável

Neste capítulo é descrita a arquitetura proposta para melhorar a testabilidade de um componente de *software* através do aumento de sua observabilidade. A melhoria da observabilidade contribui para a descoberta de falhas, uma vez que o comportamento do componente pode ser melhor observado durante os testes. A construção do componente testável é um dos principais passos na execução do método de testes proposto.

Um componente testável contém, adicionalmente à sua implementação, mecanismos que auxiliam as atividades de teste: mecanismos de monitoração, assertivas e especificações comportamentais do componente a partir das quais casos de teste podem ser obtidos automaticamente. A arquitetura foi projetada e implementada com base nos trabalhos apresentados no Capítulo 3, e é voltada para uma futura automação na sua construção.

A Seção 4.2 descreve quais foram as diretrizes seguidas durante o projeto da arquitetura, para garantia de sua qualidade; a Seção 4.3 apresenta a arquitetura do componente testável de uma forma genérica, independente da linguagem a ser utilizada na sua implementação; a Seção 4.4 apresenta a análise de várias técnicas estudadas para a implementação, e a descrição da abordagem para implementação da arquitetura utilizando a técnica escolhida.

Além da arquitetura, é também sucintamente descrito um estudo de caso simples, utilizado nos Capítulos 4, 5 e 6 para exemplificação das abordagens. Sua descrição é apresentada na Seção 4.1.

A Seção 4.5 traz um sumário do capítulo.

4.1 Exemplo: Caldeira a Vapor

Como o estudo de caso realizado em um ambiente real é de maior complexidade (descrito no Capítulo 7), optou-se pela utilização de um estudo de caso simples para a exemplificação durante o texto. O escolhido foi a especificação do controlador de caldeira de vapor, proposto em [4].

A implementação utilizada foi desenvolvida por P. Guerra [29] e é baseada no estilo arquitetural C2 [71]. No estilo C2, o sistema é estruturado em camadas, integradas com a utilização de conectores, que são componentes especiais responsáveis pelo roteamento, transmissão e filtragem de mensagens. As requisições fluem de baixo para cima na arquitetura, e as respostas (notificações), de cima pra baixo.

A arquitetura do sistema é ilustrada na Figura 4.1. O sistema é composto pelas camadas 1, 2 e 3, as quais são integradas com os conectores `conn1`, `conn2` e `conn3`. A quarta camada é composta por sensores e atuadores, que em um ambiente real seriam peças de *hardware*, mas que na implementação utilizada são simulados por *software*.

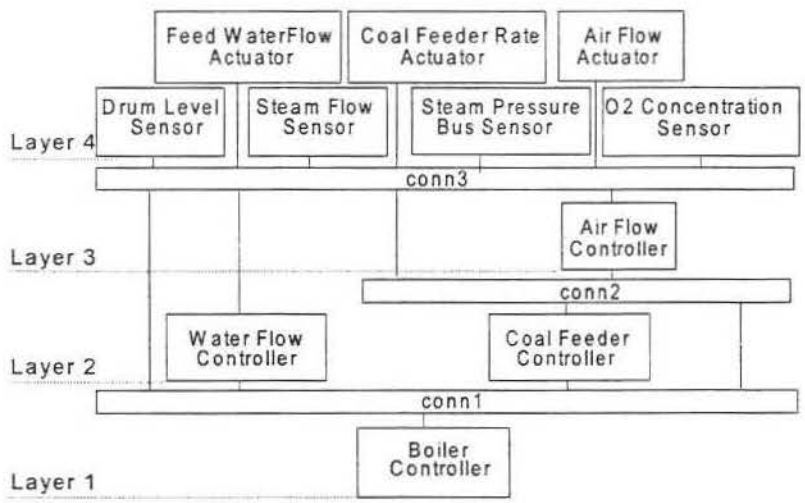

Figura 4.1: Arquitetura do controlador de caldeira de vapor.

O componente escolhido para o estudo de caso foi o `AirFlowController` (controlador de fluxo de ar), por ser o único implementado como um componente tolerante a falhas ideal (Seção 2.1.2) na implementação utilizada. Sua única interface (`IAirFlowController`) é ilustrada na Figura 4.2. Possui três operações: uma definida internamente, e duas herdadas da interface `IController`.

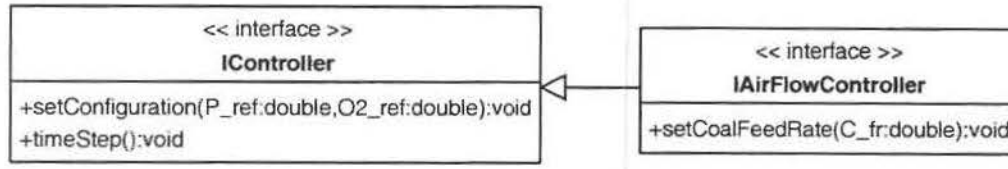

Figura 4.2: Interface do componente `AirFlowController` (estudo de caso)

4.2 Diretrizes do Projeto do Componente Testável

Durante o projeto da arquitetura do componente testável, algumas diretrizes foram firmadas para a garantia da qualidade. Essas diretrizes são enumeradas a seguir:

1. *Facilidade de uso*: o componente testável deve ser desenvolvido e utilizado a um baixo custo de programação;

2. *Separação de interesses* (do inglês *Separation of concerns*): o mecanismos de testes e monitoração devem ser independentes tanto do componente sob teste (CST), quanto um do outro;

3. *Extensibilidade* (do inglês *Extensibility*): a adição de novos mecanismos e a manutenção dos já existentes deve ser realizada a um baixo custo;

4. *Intrusão reduzida*: os mecanismos incluídos no CST não devem gerar um impacto demasiado em seu desempenho e espaço de armazenamento;

5. *Independência do código fonte*: os mecanismos devem ser incluídos/removidos do CST sem a necessidade de compilação, permitindo a instrumentação de componentes adquiridos de terceiros, os chamados COTS (do inglês *Commercial off-the-shelf*).

A partir das diretrizes estabelecidas e dos trabalhos estudados, definiu-se um formato para o componente testável e para os serviços a serem oferecidos, cobrindo vários fatores para aumento da testabilidade do componente: representação, capacidade de teste embutido e ferramentas de teste.

A capacidade de teste embutido foi implementada com a instrumentação do componente, isto é, a inclusão de funções auxiliares no CST. Neste trabalho, a inclusão da instrumentação tem como objetivo facilitar a validação do componente no ambiente do usuário, que não conhece o componente detalhadamente. Assim, não é interessante que o encapsulamento do componente seja quebrado pelos mecanismos de monitoração e testes, já que o usuário não possui o conhecimento suficiente para entender os detalhes internos do componente.

A partir das propostas apresentadas no trabalho *Testable Beans* (Seção 3.2), a abordagem escolhida para a implementação da arquitetura do componente testável foi o empacotamento automático do componente (*Automatic component wrapping*) [40], no qual apenas os métodos e atributos públicos do componente são instrumentados. Outra decisão foi permitir que o usuário escolha quais métodos devem ser instrumentados, seguindo a diretriz de intrusão reduzida.

Baseando-se nos trabalhos *Testable Beans* e *Componentes Autotestáveis* (Seção 3.2), foi decidido pela instrumentação do componente com mecanismos de monitoração e de verificação. Os mecanismos de monitoração são responsáveis pelo registro das chamadas de métodos das interfaces providas e requeridas do componente, enquanto os de verificação são responsáveis pela implementação da abordagem "*Design-by-contract*" (Seção 2.1.1). O contrato deve ser obtido a partir dos requisitos do componente e registrados na especificação do CST, que também é fornecida pelo componente testável.

Dois tipos de especificações são disponibilizados pelo componente testável:

- Contratual: consiste no contrato das interfaces do componente, especificados em OCL. A partir desta especificação, é possível a geração automática dos mecanismos de verificação a serem incluídos no componente.

- Comportamental: consiste na descrição do comportamento das interfaces providas e requeridas. Esta especificação é utilizada para geração automática de casos de teste.

O formato das especificações e os algoritmos para geração dos mecanismos de verificação e dos casos de teste são apresentados no Capítulo 5.

4.3 Descrição da Arquitetura

De acordo com as diretrizes estabelecidas e com os trabalhos estudados, o componente testável oferece duas interfaces públicas adicionais: uma responsável pelos mecanismos de monitoração e outra pelos mecanismos de teste. A arquitetura desenvolvida é apresentada na Figura 4.3. Seguindo as diretrizes de separação de interesses e extensibilidade, além do CST foram adicionados os subcomponentes `Tracker`, que implementa a interface de monitoração `ITracking`, e `Tester`, que implementa a interface de testes `ITesting`. Essas interfaces devem ser usadas pelo *driver* de testes para a preparação do CST. Os componentes são detalhados nas subseções a seguir.

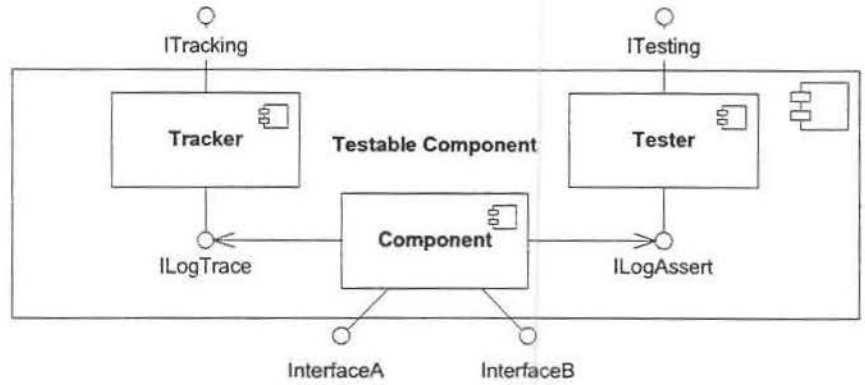

Figura 4.3: Arquitetura do componente testável.

4.3.1 Componente Tracker

O componente `Tracker` oferece métodos para inclusão de mecanismos de monitoração no CST. São oferecidos três tipos de monitoração [40]: operacional, que registra as interações entre os métodos públicos do CST (tanto das interfaces providas quanto requeridas); de erros, que registra as exceções lançadas pelo CST; e de estados, que registra o estado de atributos públicos ou propriedades do componente. Outros tipos de monitoração, como de desempenho ou de eventos [40], não foram considerados na construção dessa arquitetura, mas podem ser incorporados posteriormente.

São oferecidas a interface pública `ITracking`, utilizada pelos clientes do CST para a escolha dos mecanismos de monitoração a serem incluídos, e a interface `ILogTrace`, que tem visibilidade de pacote e é utilizada pelo CST após a instrumentação para o registro da execução em um arquivo de *log*. O caminho do arquivo de *log* é determinado através de um arquivo de configuração.

A estrutura das interfaces `ITracking` e `ILogTrace` é exibida na Figura 4.4. Os métodos da interface `ITracking` permitem que o usuário escolha o que será monitorado (para a escolha de mais de um item, basta chamar o método várias vezes):

1. `operationalTrace`: escolha de qual interface terá seus métodos monitorados, podendo optar pelos providos, requeridos ou ambos;

2. `stateTrace`: escolha de qual atributo/propriedade será monitorado;

3. `errorTrace`: escolha de qual exceção (e seu tratamento/lançamento) será monitorada.

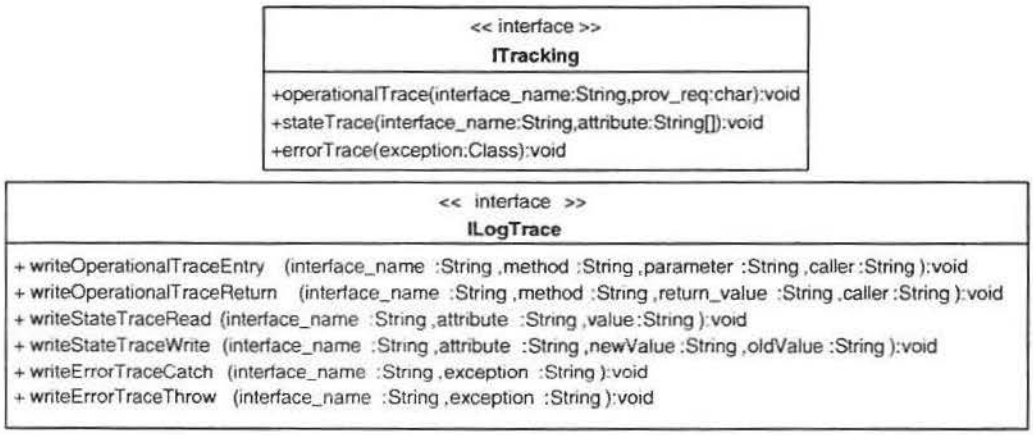

Figura 4.4: Interfaces do componente `Tracker`: `ITracking` e `ILogTrace`.

4.3.2 Componente Tester

O componente `Tester` oferece serviços para a inclusão de assertivas no CST, notificação de violações e resgate das especificações. Possui duas interfaces: `ITesting` e `ILogAssert`. A interface `ITesting` é pública e utilizada tanto pelos clientes do CST, quanto para a escolha das assertivas a serem incluídas, e por ferramentas de geração de testes, que buscam as especificações do componente. A interface `ILogAssert` tem visibilidade de pacote e é utilizada pelo CST após sua instrumentação, para o registro de violações de assertivas em um arquivo de *log*. As estruturas de ambas são exibidas na Figura 4.5.

Quando uma assertiva é violada, uma ação deve ser produzida com as responsabilidades de notificação e continuação [16]. A notificação é realizada pelo registro no arquivo de *log*, e a ação de continuação determina se a execução deve continuar após a violação de uma assertiva ou não. Tanto as opções de continuação quanto o caminho do arquivo de *log* são definidos em um arquivo de configuração.

O usuário deve escolher a ação de continuação de acordo com seus objetivos de teste. Interrompendo a continuação, cada violação é tratada separadamente. Permitindo a continuação, é possível observar o comportamento do componente caso a violação aconteça em um ambiente real. A continuação é interessante especialmente para pré-condições e invariantes, pois permite a observação da reação do componente frente a uma entrada inválida.

As especificações são obtidas pelos métodos `ITesting.getAssertions()`, que fornece as especificações contratuais das interfaces providas, e `ITesting.getModel()`, que fornece

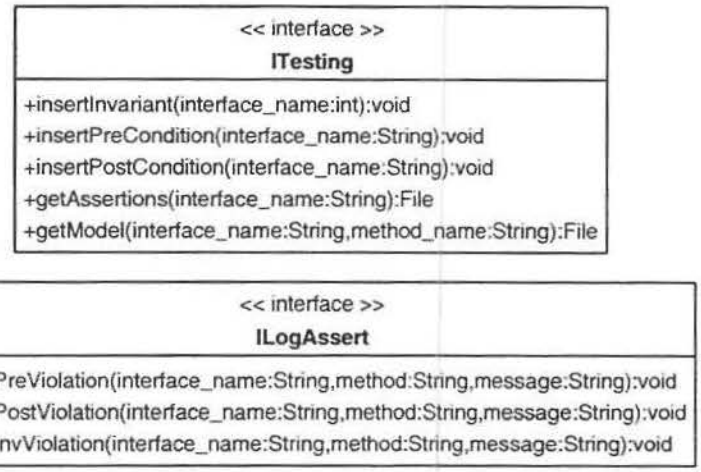

Figura 4.5: Interfaces do componente `Tester`: `ITesting` e `ILogAssert`.

as especificações comportamentais das interfaces providas e requeridas do componente. O formato das especificações e sua utilização para a automação da estratégia são descritos no Capítulo 5.

4.4 Implementação do Componente Testável

A implementação do componente testável consiste na criação do código a ser embutido e na implementação dos componentes `Tester` e `Tracker`. Durante a criação do código *built-in*, são construídas as bibliotecas de testes e monitoramento. A biblioteca de testes contém o código para verificação das assertivas, implementando o contrato do CST, enquanto a biblioteca de monitoramento contém o código para implementação dos mecanismos de monitoramento de métodos, atributos e erros.

Para a definição de como o código de verificação de assertivas e de monitoramento seria incluído no componente sob teste, várias técnicas de instrumentação foram analisadas. As subseções seguintes trazem uma descrição das técnicas estudadas, com destaque para a abordagem adotada, programação orientada a aspectos [52]. Em seguida, é descrito como os componentes `Tester` e `Tracker` e suas bibliotecas foram implementados, utilizando este paradigma.

4.4.1 Técnicas para instrumentação dos componentes

Como a inclusão dos mecanismos ao componente reutilizável é um aspecto importante da implementação do componente testável, foram estudadas diversas técnicas para a modificação de componentes. Além das diretrizes apresentadas na Seção 4.2, considerou-se também os seguintes aspectos para a escolha da forma de instrumentação dos componentes:

- Possibilidade sobre a escolha do código a ser incluído, permitindo ao cliente do componente a instrumentação apenas dos métodos testados pela bateria de testes corrente, prevenindo a criação de *logs* muito extensos;
- Utilização da linguagem Java, que vem sendo largamente adotada atualmente e com a qual os estudos de caso neste trabalho foram realizados.

As abordagens estudadas e uma avaliação sobre cada uma são apresentadas nas subseções a seguir.

Wrapping

Nesta técnica, o componente é envolvido por um componente especial denominado *wrapper* (invólucro) [78]. O *wrapper* intercepta todas as requisições para o componente ou feitas por ele, podendo realizar computações sobre elas. Neste caso, o cliente do componente faz requisições ao *wrapper*, que as encaminha ao componente original. Um *wrapper* pode encapsular um ou mais componentes.

A principal vantagem dessa técnica é a facilidade de implementação, já que não é necessário o aprendizado de novas linguagens ou paradigmas. A desvantagem é que todas as requisições devem obrigatoriamente passar pelo *wrapper*, independentemente se o método foi instrumentado ou não, incluindo um custo desnecessário para a execução.

Proxy

Esta técnica cria um componente *proxy* [38], que atua como um intermediário na comunicação entre um componente e seu(s) cliente(s). O *proxy* funciona de maneira semelhante ao *wrapper*, recebendo as requisições e repassando para o componente, podendo modificá-las. A diferença é que o *proxy* não empacota o componente, por isso outros componentes da aplicação ainda podem continuar invocando diretamente o componente original.

Assim como na técnica de *wrapping*, a principal vantagem da técnica de *proxy* é a facilidade de implementação. A desvantagem é que, mesmo permitindo a escolha da localização da instrumentação, os métodos instrumentados têm seu desempenho afetado pela interceptação do *proxy*.

Reflexão Computacional

Reflexão computacional é a atividade realizada por um sistema computacional quando faz computações sobre (e possivelmente afetando) suas próprias computações [55]. A reflexão define uma arquitetura composta por níveis, no qual o nível-base corresponde aos elementos do sistema que compõe as funcionalidades da aplicação, e o meta-nível contém uma auto-representação do sistema, permitindo a obtenção de informações sobre sua estrutura e comportamento. Essa estrutura pode ser recursiva: o meta-nível pode ter um meta-meta-nível, e assim por diante [83].

A partir das informações contidas no meta-nível, o sistema pode interferir em suas próprias computações, do meta-nível para o nível-base, tanto para obter informações sobre a estrutura (introspecção), para modificar a estrutura (reflexão estrutural) ou para realizar transformações no comportamento no nível base (reflexão comportamental). A principal desvantagem da abordagem de reflexão computacional é a necessidade de monitoração constante do nível-base pelo meta-nível, prejudicando o desempenho da aplicação através das interferências.

Manipulação de *bytecodes*

Nesta técnica, os *bytecodes* Java são modificados diretamente com a utilização de bibliotecas que possibilitam inclusão de código, exclusão ou até mesmo modificação do código já implementado. As principais vantagens da abordagem são o grande poder para a modificação das classes e a minimização do custo relativo ao desempenho, pois o código é inserido diretamente nos *bytecodes* Java. Apesar dessas vantagens, o principal problema é a dificuldade de utilização, já que é necessário o entendimento da estrutura dos *bytecodes* Java.

A linguagem BCEL (*Byte Code Engineering Library*) [28] é uma biblioteca para a manipulação de *bytecodes* Java. A biblioteca fornece métodos para obtenção de informações sobre a classe, como métodos, atributos e até mesmo instruções do *bytecode*, e para a modificação destes. Essa biblioteca foi utilizada em um primeiro estudo de caso neste trabalho (descrito em [66]), porém a abordagem foi abandonada devido ao alto custo para criação dos mecanismos a serem embutidos no componente, o que invibializaria sua utilização em larga escala.

Programação Orientada a Aspectos

Segundo G. Kickzales et. al [52], a programação orientada a aspectos é uma nova metodologia que possibilita a separação de interesses transversais[1] (*crosscuting concerns*)

[1] A tradução para português segue a terminologia definida durante o Primeiro Workshop Brasileiro de Desenvolvimento de *Software* Orientado a Aspectos (WASP04), e está disponível em

pela introdução de uma nova unidade de modularização - o aspecto. Esses interesses transversais são implementados como aspectos, e a composição do sistema é feita pelo combinador (*weaver*), que combina o componente de negócio e os aspectos contendo os interesses transversais. Essa combinação pode ocorrer em tempo de compilação, de carga ou até de execução. Diversas ferramentas implementam programação orientada a aspectos para vários ambientes, como AspectJ para Java [53], e Aspect# para .Net [72].

Assertivas e mecanismos de monitoração podem ser considerados interesses transversais, já que estão presentes em todos os módulos mas não fazem parte diretamente da funcionalidade do sistema. A implementação desses conceitos como aspectos, além de contribuir com a manutenibilidade, permite que eles sejam incluídos e retirados do sistema automaticamente, mesmo sem a presença do código fonte (em tempo de carga).

Assim, apesar de ser uma abordagem nova, a programação orientada a aspectos foi escolhida para a implementação dos artefatos propostos. Além do desafio da utilização, foi escolhida por propor uma estrutura de fácil entendimento e possibilitar a instrumentação sem a presença do código fonte. Além disso, apresenta um desempenho relativamente melhor do que outras abordagens (*wrapper*, *proxy* e reflexão), pois o código para interceptação é inserido diretamente nos *bytecodes* dos pontos interceptados, não havendo necessidade de monitoração constante. A ferramenta utilizada foi AspectJ, uma extensão orientada a aspectos para a linguagem Java [53]. É a ferramenta mais completa para orientação a aspectos em Java, sendo desenvolvida como código aberto desde 1997 e integrada ao projeto da plataforma de desenvolvimento Eclipse [64].

A extensão para suporte à orientação a aspectos introduziu alguns novos conceitos à linguagem Java [53], específicos para a implementação dos aspectos. O conceito fundamental é o ponto de junção (*join point*), um conceito abstrato, isto é, não implementado diretamente, que representa um determinado ponto no fluxo de execução que possivelmente será interceptado por um aspecto. A chamada ao método `timeStep()`, da interface `IAirFlowController`, por exemplo, pode ser escolhida como um ponto de junção.

Os demais conceitos são implementados na linguagem Java:

- Conjunto de junção (*pointcut*): é a implementação do conceito de ponto de junção. O conjunto de junção representa determinados pontos de junção do fluxo de execução, podendo agrupá-los. Esses pontos podem ser interceptados durante a execução, podendo receber regras de combinação, como por exemplo, a chamada de um método de monitoração antes da execução de cada um dos pontos interceptados.

  Um conjunto de junção pode, por exemplo, representar o conjunto das chamadas dos métodos públicos da interface `IAirFlowController`, agrupando todos esses pontos de junção (chamada de cada método) em um só conjunto. A sintaxe para esse

http://twiki.im.ufba.br/bin/view/AOSDbr/TermosEmPortugues

exemplo seria

```
private pointcut cj_metodos(): //nome do conjunto de junção
execution (public * IAirFlowController.*(..)); //declaração do ponto de junção
```

Os conjuntos de junção apresentam diferentes parâmetros, capazes representar de diversos tipos de pontos no código. É possível, por exemplo, interceptar tanto a execução (`execution`, conforme exemplo) quanto a chamada de um método (`call`). No caso de interceptar a execução, o código do próprio método seria instrumentado. No caso de interceptação da chamada, apenas o código do método que o invoca. É possível também interceptar a leitura (`get`) e escrita (`set`) de atributos.

Outros parâmetros do conjunto de junção utilizados neste trabalho são: `args`, para coleta de valores dos parâmetros de um método interceptado; e `cflow`, que permite a limitação de cobertura do conjunto de junção. Um exemplo do uso de `cflow` seria um conjunto de junção que intercepta chamadas a quaisquer métodos das interfaces requeridas pelo componente, sendo limitado para apenas as chamadas realizadas por determinado método provido.

- Adendo (*advice*): contém o código a ser executado quando um determinado ponto de junção (capturado por um conjunto de junção) é atingido. Existem três tipos: adendo anterior (*before*), que define ações a serem realizadas antes do ponto de junção; adendo posterior (*after*), que define ações a serem realizadas após o ponto de junção; e adendo de contorno (*around*), que define ações que englobam a execução do ponto de junção. Um exemplo de adendo anterior seria a implementação da chamada ao método de monitoração para os métodos públicos da `IAirFlowController`. Considerando o conjunto de junção exibido no exemplo anterior, um adendo para a chamada de um método de monitoração (ex., `monitora()`) antes de cada método interceptado pelo conjunto de junção seria:

  ```
  before(): cj_metodos() { //nome do conjunto de junção a ser interceptado
  monitora();//código a ser executado durante a interceptação
  }
  ```

- Declarações intertipos (*inter-type declarations*): possibilitam a modificação da estrutura do programa estaticamente, como a criação de novos atributos e métodos, por exemplo. Este tipo de construção não será utilizada neste trabalho.

- Aspectos (*aspects*): estruturas similares a classes Java que encapsulam as regras de combinação construídas com a utilização dos conceitos anteriores.

Os conceitos de conjunto de junção, adendo e aspecto, citados anteriormente, foram utilizados e serão exemplificados neste capítulo. Maiores informações sobre Programação Orientada a Aspectos e a ferramenta AspectJ podem ser obtidas em [53].

4.4.2 Arquitetura Interna dos Componentes

A Figura 4.6 mostra a arquitetura interna do componente `Tester`. A arquitetura do componente `Tracker` é similar, com mudanças nos nomes das interfaces `ITesting` e `ILogAssert` e suas respectivas classes implementadoras. O componente `Tracker` também não possui o pacote `specification`, onde as especificações do CST são armazenadas.

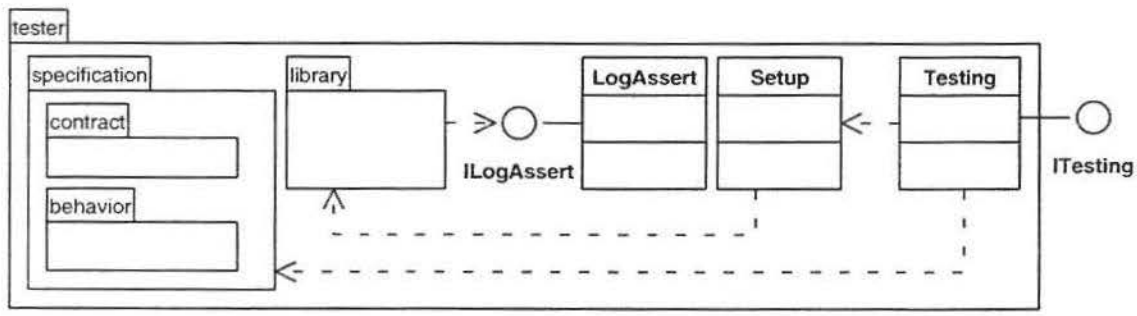

Figura 4.6: Arquitetura do componente `Tester`.

As classes exibidas na Figura 4.6 implementam a funcionalidade que permite ao usuário escolher qual instrumentação será incluída no CST. A escolha é baseada na estrutura da biblioteca de aspectos: o usuário, de forma transparente a partir da interface `ITesting`, escolhe quais aspectos serão combinados ao CST. Esses aspectos são agrupados em um arquivo pela classe `Setup`. Para facilitar a implementação do componente testável por trabalhos futuros, o código de ambas as classes é apresentado no Apêndice A.

Para facilitar e melhorar a performance da criação do arquivo com os aspectos, a interface `ITesting` sofreu algumas mudanças em relação a versão da fase de projetos. Recebeu mais um método, `void finalize()`, que indica que todos os aspectos já foram escolhidos, permitindo que o arquivo seja gravado em disco e o processo de combinação comece. A exceção `FileNotFoundException` passou a ser lançada pelos métodos `insertPrecondition()`, `insertPostcondition()` e `insertInvariant()`, indicando que o aspecto a ser inserido não foi encontrado, e também pelo método `getModel()`, quando uma determinada especificação não está disponível.

O pacote `specification` possui outros dois pacotes internos, para separação das especificações contratuais e comportamentais. A estrutura do pacote `library` para os componentes `Tester` e `Tracker` é apresentada na próxima seção.

4.4.3 Arquitetura Interna das Bibliotecas

As bibliotecas dos componentes `Tester` e `Tracker` são compostas por aspectos, que contêm o código a ser incluído no CST. Os aspectos foram organizados para que o usuário escolha quais devem ser incluídos no CST. As subseções seguintes apresentam a estrutura das bibliotecas dos componentes `Tracker` e `Tester`.

Biblioteca Tracker

A biblioteca `Tracker` é dividida nos pacotes `operational`, `error` e `state`, como ilustra a Figura 4.7. O pacote `operational` contém os aspectos relativos ao monitoramento dos métodos, que são divididos de acordo com as interfaces do componente: há um aspecto para cada interface provida e requerida, contendo o código necessário para a monitoração de todos os métodos públicos da respectiva interface. Por exemplo, para o componente AirFlowController, existem os aspectos `IAirFlowControllerP`, para a interface provida, e o `IAirFlowControllerR`, para a requerida. Os nomes dos aspectos são formados pelo nome da interface ("IAirFlowController") mais o indicativo de provida ("P") ou requerida ("R").

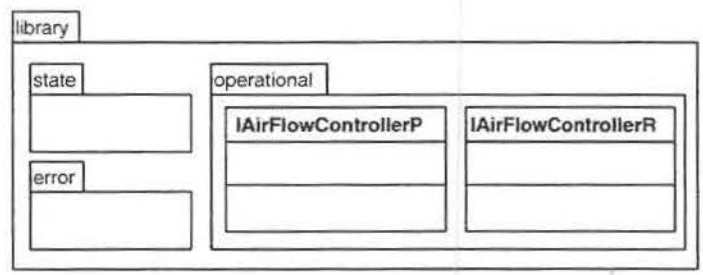

Figura 4.7: Estrutura da biblioteca de aspectos do componente `Tracker`.

O código dos aspectos operacionais é simples. Os aspectos para interfaces providas contêm, para cada método público da interface, um conjunto de junção para a execução do método, e três adendos contendo o código a ser executado antes e depois do método.

A Figura 4.8 representa parte do aspecto `IAirFlowControllerP`, que implementa a monitoração operacional da interface provida `IAirFlowController`. É exibido o código relativo à monitoração do método `setCoalFeedRate`. O conjunto de junção `setCoalFeedRateMethod` (linhas 6 a 8) intercepta a execução do método (linha 7) e captura o valor de seu parâmetro de entrada (linha 8). Os adendos realizam o registro da execução no arquivo de *log*. O código contido no adendo anterior (linhas 11 a 14) é executado antes do conjunto de junção, enquanto o contido nos adendos posteriores (linhas 17 a 24) é executado depois. Porém, o adendo posterior marcado com a palavra

returning() (linhas 16 a 19) é executado somente se o método interceptado tiver terminado normalmente. Caso tenha terminado de forma excepcional, o adendo posterior marcado com a palavra throwing() é executado (linhas 21 a 24), registrando no *log* a exceção lançada.

```
1  public privileged aspect AirFlowControllerP {
2
3    private LogTrace log = LogTrace.singleton();
4
5    //Conjunto de junção para monitoramento do metodo setCoalFeedRate
6    private pointcut setCoalFeedRateMethod(double C_fr) :
7      execution (public void AfcWrappedNormal.setCoalFeedRate(double))
8      && args(C_fr);
9
10   //Adendo anterior para monitoramento do metodo setCoalFeedRate
11   before (double C_fr): setCoalFeedRateMethod(C_fr){
12     log.writeOperationalTraceEntry("ftBoilerSystem.AfcWrappedNormal",
13                                    "setCoalFeedRate", "C_fr = " + C_fr);
14   }
15   //Adendo posterior normal para monitoramento do metodo setCoalFeedRate
16   after (double C_fr) returning(): setCoalFeedRateMethod(C_fr){
17     log.writeOperationalTraceReturn("ftBoilerSystem.AfcWrappedNormal",
18                                     "setCoalFeedRate", "");
19
20   //Adendo posterior excepcional para monitoramento do setCoalFeedRate
21   after (double C_fr) throwing (Exception e): setCoalFeedRateMethod(C_fr){
22     log.writeOperationalTraceReturn("ftBoilerSystem.AfcWrappedNormal",
23                                     "setCoalFeedRate", e.getClass().getName());
24  }
25 }
```

Figura 4.8: Código dos aspectos para monitoração do método setCoalFeedRate, da interface provida IAirFlowController.

Os aspectos para monitoração das interfaces requeridas são similares, possuindo também um conjunto de junção e três adendos (um anterior, um posterior normal e um posterior excepcional). Os conjuntos de junção, porém, devem ser do tipo call, e devem escolher apenas pontos de junção que forem chamados pelos métodos da interface correspondente, como exemplifica a Figura 4.9. Nesta figura é ilustrado o código de um conjunto de junção que intercepta a chamada ao método controlInputA da interface requerida PIDController (linha 2). São recolhidos o valor do parâmetro de entrada (linha 3), e os métodos interceptados são limitados às chamadas realizadas pelos métodos públicos da interface provida. Os adendos são similares aos exibidos na Figura 4.8.

A estrutura do pacote state é similar à do pacote operational: há um aspecto

```
1  private pointcut controller_access(double value):
2    call(public double PIDController.controlInputA(double))
3    && args(value)
4    && cflow(public * IAirFlowController.*(..));
```

Figura 4.9: Código do conjunto de junção para monitoração do método `controlInputA(double)`, da interface requerida `PIDController`.

para cada atributo/propriedade pública do componente. Caso haja um grande número de atributos, os aspectos podem ser agrupados de acordo com as interfaces. Os conjuntos de junção são os de leitura e escrita do atributo (`get/set`), e há adendos anterior e posterior para registrar o valor antes e depois do acesso, respectivamente. A estrutura do pacote `error` ainda não foi definida, e será descrita em um trabalho futuro. Neste trabalho, as exceções recebidas e lançadas são monitoradas no componente isolado através do monitoramento operacional.

As bibliotecas do componente `Tracker` podem ser geradas automaticamente, a partir das interfaces providas e requeridas do componente. Os passos para essa geração são apresentados no Capítulo 5.

Biblioteca Tester

A biblioteca do componente `Tester` contém os aspectos para a verificação das assertivas dos métodos públicos das interfaces providas do CST. Como ilustrado na Figura 4.10, a biblioteca é dividida em pacotes relativos às interfaces providas do componente. Cada pacote pode conter um aspecto para a verificação da invariante e dois aspectos para cada método: um para as pré-condições e outro para as pós-condições.

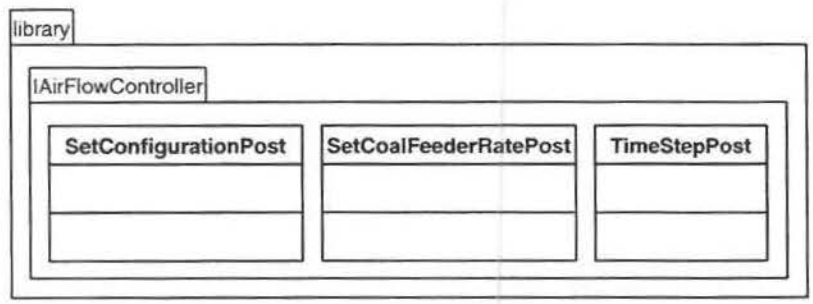

Figura 4.10: Estrutura da biblioteca de aspectos do componente `Tester`.

```
1  public privileged aspect SetCoalFeedRatePost {
2
3   private LogAssert log = LogAssert.singleton();
4   private double req_value1;
5
6   //Conjunto de junção para acesso ao retorno do método da interface
7   //requerida PIDController.controlInputA(double)
8   private pointcut req1():
9    call(public double PIDController.controlInputA(double))
10   && cflow(assertion(double));
11
12  //Adendo posterior para coleta campo req_value1
13  after() returning (double par_req_value1): req1() {
14   req_value1 = par_req_value1;
15  }
16
17  //Conjunto de junção pra assertivas
18  private pointcut assertion(double C_fr):
19   execution(public void AfcWrappedNormal.setCoalFeedRate(double))
20   && args (C_fr);
21
22  //Adendo posterior para a verificação da pós condição sem exceções
23  after(double C_fr) returning (): assertion(C_fr) {
24   //InvalidCoalFeederRate
25   if ((C_fr < 0) || (C_fr > 1))
26    log.writePostViolation("ftBoilerSystem.AfcWrappedNormal", "setCoalFeedRate(double)",
27                           "Excecao InvalidCoalFeederRate nao foi lancada");
28   //InvalidAirFlowRate
29   if ((req_value1 < 0) || (req_value1 > 0.1))
30    log.writePostViolation("ftBoilerSystem.AfcWrappedNormal", "setCoalFeedRate(double)",
31                           "Excecao InvalidAirFlowRate nao foi lancada");
32  }
33
34  //Adendo posterior para a verificação da pós condição com exceções, isto é, se as
35  //exceções esperadas foram lançadas
36  after(double C_fr) throwing (Exception e): assertion(C_fr) {
37   //InvalidCoalFeederRate
38   if (((C_fr < 0) || (C_fr > 1))
39      && !(e.getClass().getName().equals("ftBoilerSystem.InvalidCoalFeederRate")))
40    log.writePostViolation("ftBoilerSystem.AfcWrappedNormal", "setCoalFeedRate(double)",
41                           "Excecao InvalidCoalFeederRate nao foi lancada");
42   //InvalidAirFlowRate
43   if (((req_value1 < 0) || (req_value1 > 0.1)) &
44      && !(e.getClass().getName().equals("ftBoilerSystem.InvalidAirFlowRate")))
45    log.writePostViolation("ftBoilerSystem.AfcWrappedNormal", "setCoalFeedRate(double)",
46                           "Excecao InvalidAirFlowRate nao foi lancada");
47  }
48 }
```

Figura 4.11: Código do aspecto para verificação da pré-condição do método `setConfiguration`.

No pacote `IAirFlowController` (Figura 4.10) estão contidos os aspectos de pós-condições para os métodos desta interface: `setConfiguration()`, `setCoalFeedRate()` e `timeStep()`. Como o componente não tem estado interno, não foram definidas invariantes para essa interface. Também não foram criados os aspectos referentes às pré-condições, pois na implementação do componente foi utilizada a abordagem de programação defensiva [56]. Nesta abordagem, a validade das entradas fornecidas ao componente é verificada pelo próprio método que as recebe. Por isso, métodos nos quais a programação defensiva é utilizada não possuem pré-condições, já que todas as entradas são aceitas. No componente ideal tolerante a falhas (Seção 2.1.2), entradas inválidas são sinalizadas como exceções de interface.

O código do aspecto para verificação das pós-condições do método `setCoalFeedRate()` é exibido na Figura 4.11. O algoritmo para sua geração automática a partir da especificação contratual da interface é apresentado no Capítulo 5.

O conjunto de junção `assertion` (linhas 17 a 19) e seus adendos (linhas 22 a 44) implementam a verificação das pós-condições, quando o método retorna normal ou excepcionalmente. Mesmo o método não possuindo valor de retorno, o retorno normal é interceptado para verificar se alguma exceção deveria ter sido lançada. Nas linhas 24 a 26, é verificado se a exceção de interface `InvalidCoalFeederRate` deveria ter sido lançada. Nas linhas 28 a 30, a exceção `InvalidAirFlowRate`, que deve ser lançada caso a interface requerida `PIDController` retorne um valor inválido. O valor de retorno da interface requerida é obtido através do conjunto de junção `req1` (linhas 8 a 10), que intercepta a chamada do método `PIDController.controlInputA()`, e de seu adendo posterior (linhas 13 a 15), que captura o valor de retorno e o armazena na variável `req1`.

O adendo de retorno excepcional verifica se a exceção foi lançada corretamente. Nas linhas 35 a 38, o lançamento da exceção de interface `InvalidCoalFeederRate` é verificado em relação aos valores passados como parâmetro, e nas linhas 40 a 44, o lançamento da exceção `InvalidAirFlowRate`, examinando o valor retornado pelo método `PIDController.controlInputA()` através da variável `req1`.

4.5 Considerações Finais

Neste capítulo foi apresentada a arquitetura do componente testável, que é a base para o método de testes apresentado no Capítulo 6. O método utiliza tanto a verificação do contrato do componente, quanto as especificações fornecidas pela arquitetura do componente testável, a partir das quais são gerados casos de teste, como descrito no próximo capítulo.

Foram descritas as propriedades que guiaram o desenvolvimento da arquitetura; seus componentes e a arquitetura interna destes; sua implementação utilizando programação

orientada a aspectos. A automação da geração da arquitetura é descrita no próximo capítulo.

Capítulo 5

Projeto e Implementação dos Testes usando o Componente Testável

A arquitetura do componente testável, além da inserção de assertivas e mecanismos de monitoração detalhados no Capítulo /refcap:arquitetura, também disponibiliza dois tipos de especificação do componente sob teste. O primeiro tipo retrata aspectos comportamentais, e o segundo registra o contrato das interfaces na linguagem OCL (*Object Constraint Language*). Ambos foram preparados para que possam ser utilizados por ferramentas desenvolvidas futuramente, respectivamente para geração automática de casos de teste e do código executável das assertivas, reforçando a diretriz de projeto "Facilidade de uso" (Seção 4.2).

O objetivo é a incorporação dessas funcionalidades no ambiente Bellatrix [73], um projeto do IC/Unicamp para criação de um ambiente para desenvolvimento de sistemas baseados em componentes, integrado à plataforma Eclipse [64]. O ambiente Bellatrix é voltado para o desenvolvimento de sistemas confiáveis, e fornecerá suporte a todo o ciclo de desenvolvimento: modelagem, implementação e testes.

O módulo específico para os testes implementará a geração automática de casos de teste (incluindo a geração da estrutura do componente testável), além do suporte à execução e avaliação dos resultados. A integração facilita a utilização dos modelos tanto durante o desenvolvimento quanto durante os testes, e permite que até a avaliação dos resultados da execução dos testes seja automatizada.

A estrutura montada para os testes de um componente isolado é ilustrada na Figura 5.1, contendo:

- Os componentes Tester e Tracker, detalhados no Capítulo 4, que encapsulam bibliotecas de aspectos para verificação do contrato e monitoração do componente sob teste (CST).

- Os *stubs*, implementações parciais e temporárias das interfaces requeridas pelo CST [16], que podem ser controladas durante a execução dos testes.

- O *driver*, definida como uma classe ou programa responsável pela aplicação dos testes no CST [16]. Nesta estrutura, além da execução dos testes e avaliação dos resultados, o *driver* também é responsável pela chamada dos métodos para instrumentação do CST, e pela preparação dos *stubs* antes da execução de cada teste.

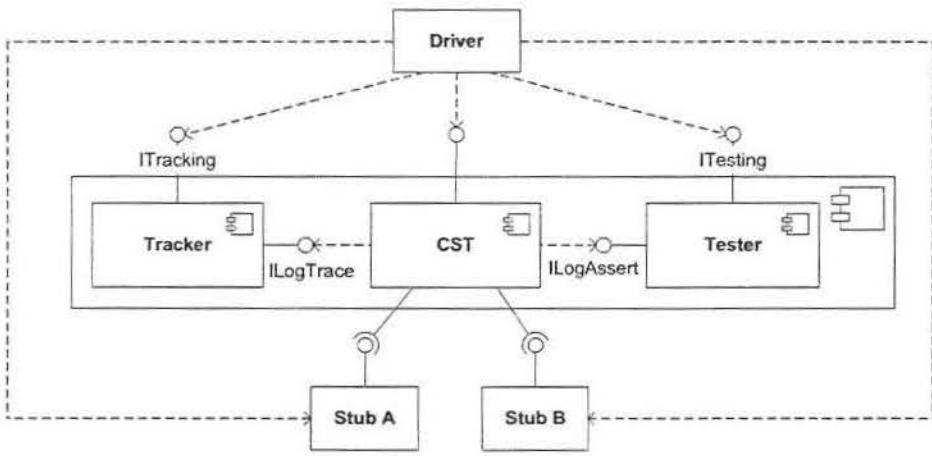

Figura 5.1: Componentes auxiliares para o teste de um componente isolado.

Este capítulo apresenta diretrizes a serem seguidas para a geração dos *drivers*, *stubs* e bibliotecas de aspectos a partir das especificações do componente. Os casos de teste são gerados a partir de caminhos no diagrama comportamental, e são especificados em uma linguagem intermediária independente de linguagens de programação. Esta especificação dá subsídios tanto para geração de *drivers* quanto de *stubs*.

A Seção 5.1 descreve os passos para a geração dos testes: como o diagrama deve ser construído, como os casos de teste são extraídos deste diagrama, como são especificados em uma linguagem intermediária e traduzidos para uma linguagem de programação. A Seção 5.2 descreve os passos para geração dos *stubs*, seguindo as mesmas diretrizes que a geração dos testes. A Seção 5.3 descreve passos simplificados para a geração de dados de teste.

Os passos para geração dos mecanismos de inserção de assertivas e de monitoração, que fazem parte da estrutura do componente testável, também são descritos neste capítulo, especialmente com o objetivo de servirem como oráculos de teste e mecanismos de depuração. A Seção 5.4 apresenta os passos para criação dos oráculos, isto é, para a geração do código executável das assertivas a partir da especificação contratual do componente.

São descritos o formato em OCL da especificação contratual e como a biblioteca do componente `Tester` é obtida a partir desta especificação. A Seção 5.5 descreve as diretrizes para geração das bibliotecas do componente `Tracker` a partir das interfaces do componente.

Finalmente, a Seção 5.6 descreve como os *drivers* de teste são estruturados para coordenar a criação de todos os outros artefatos. A Seção 5.7 apresenta as considerações finais do capítulo.

5.1 Geração de Testes

Os passos para geração automática dos casos de teste propostos neste trabalho são baseados na proposta da ferramenta ConCAT [74], que gera casos de teste a partir de uma especificação do comportamento do componente na forma de um modelo de fluxo de transação [8].

Para a ferramenta ConCAT, o modelo de fluxo de transação representa a seqüência a ser seguida para a chamada dos métodos de uma interface. Cada nó contém a assinatura de um método, e cada caminho no grafo (do nó inicial a algum nó final), equivalente a uma seqüência de métodos, representa um caso de teste.

A ferramenta, no entanto, possui limitações, como: a restrição de geração para a linguagem C++, falta de suporte à manipulação de *stubs*, a geração apenas de dados aleatórios (que muitas vezes não ativam a execução de um determinado caminho), baixa usabilidade (o modelo é fornecido através de uma classe C++). Além disso, o modelo de fluxo de transação não é muito difundido, limitando a utilização da ferramenta.

Neste trabalho, as técnicas para a geração de casos de teste a partir do fluxo de transação foram reformuladas. O modelo adotado foi o diagrama de atividades da UML [17], cuja semântica é semelhante à do modelo de fluxo de transação, mas com as vantagens de ser parte da UML, que é um padrão de fato, e ser implementado por várias ferramentas de modelagem, facilitando sua construção.

Nas subseções seguintes são descritas as fases para o projeto dos casos de teste. Inicialmente é descrito o diagrama de atividades, e como ele deve ser construído para a posterior geração dos testes; em seguida, são descritos os passos para o projeto dos casos de teste e como estes devem ser especificados em uma linguagem intermediária.

5.1.1 Especificação Comportamental

Neste trabalho o modelo adotado para especificação comportamental foi o diagrama de atividades da UML. As diferenças principais frente ao modelo adotado pela ConCAT são:

1. Inclusão de situações excepcionais explicitamente no diagrama, visando o teste de mecanismos de tratamento de exceção;

2. Inclusão de condições de guarda nas arestas, facilitando a geração dos dados de teste;

3. Criação de um diagrama específico para modelagem de detalhes de cada método, relativos ao fluxo de execução ocasionado pela invocação do método. Esse diagrama contém informações sobre a interação do método com interfaces requeridas, que servirá para a geração automática de *stubs*.

Nas subseções seguintes são descritos o diagrama de atividades da UML e o formato da especificação comportamental.

Diagrama de Atividades da UML

O diagrama de atividades é um dos diagramas dinâmicos da UML, utilizado principalmente na modelagem de seqüências de ações válidas (e possivelmente concorrentes) de um processo computacional. O diagrama representa uma atividade completa, composta por ações. De forma geral, o diagrama ilustra o fluxo de controle de uma ação a outra [17]. Assim como o modelo de fluxo de transação, tem a forma de um grafo orientado, com nós inicial e finais, e possui nós especiais para representação, por exemplo, de decisões e concorrência (*forks* e *joins*).

Neste trabalho foi adotado o diagrama de atividades da versão 2.0 da UML [44], pelo aumento considerável em seu poder de representação e legibilidade em relação a versão 1.5 [41]. Nesta nova versão, algumas das notações anteriores foram substituídas (como a introdução de partições no lugar de raias), e novas notações foram incluídas, como a modelagem de tratamento de exceções.

Entretanto, neste trabalho apenas parte dos elementos do diagrama é utilizada, a fim de diminuir a complexidade para a geração dos testes. Um diagrama exemplo, contendo todas as notações permitidas no formato utilizado neste trabalho é ilustrado nas Figuras 5.2a e 5.2b, na qual cada item é nomeado por um comentário. A Figura 5.2a mostra o fluxo entre ações. O fluxo de cada ação pode ser detalhado com a invocação de uma outra atividade, conforme ilustrado na Figura 5.2b.

Para o diagrama principal, as notações utilizadas neste trabalho são:

1. *Nó inicial*: determina o início do fluxo do diagrama.

2. *Ações*: unidade principal do fluxo do diagrama, recebem entradas, realizam transformações e geram saídas.

3. *Arestas*: estabelecem o fluxo do diagrama, conectando os itens que o compõe (como ações, decisões, *forks*).

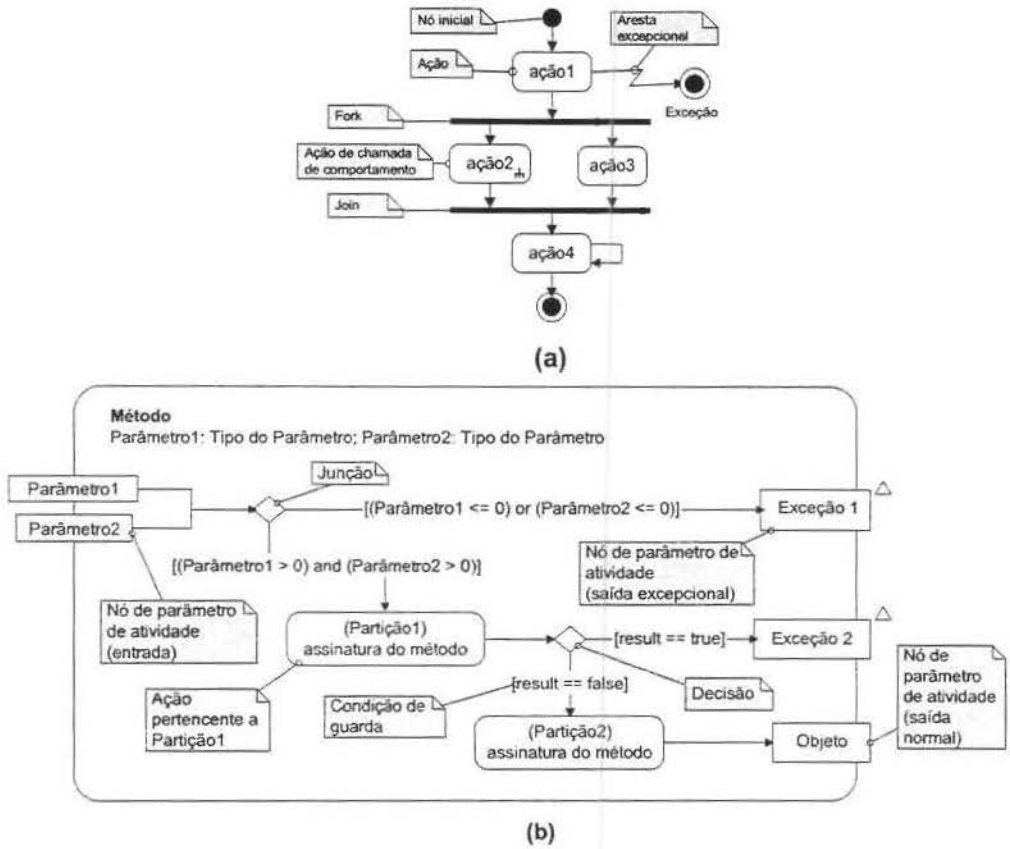

Figura 5.2: Diagrama de atividades exemplo, com as notações utilizadas para a modelagem do comportamento do componente.

4. *Arestas excepcionais*: tipo especial de aresta para fluxos de exceção.

5. *Ações de chamada de comportamento (CallBehaviorAction)*: evolução das subatividades propostas na versão 1.5, este tipo de ação possui um diagrama de atividades ligado a ela (como o da Figura 5.2b), caracterizando uma modelagem recursiva.

6. *Forks e Joins*: utilizados para a modelagem de fluxos concorrentes, o *fork* marca o início da concorrência, e o *join* marca o fim, sendo que podem abrigar dois ou mais fluxos internamente.

7. *Nó final*: determina o fim de um fluxo ou subfluxo (podem existir vários nós finais em um mesmo diagrama).

Para o diagrama de detalhamento, além das notações para a modelagem do fluxo, algumas novas notações são utilizadas:

1. *Partições*: evolução das raias propostas na versão 1.5, são utilizadas para agrupar ações com características em comum.

2. *Atividades*: nova notação da versão 2.0, serve para agrupar todas as ações de um diagrama, determinando seu contexto e fornecendo informações sobre o diagrama, como pré e pós-condições e seu título, por exemplo.

3. *Nós de parâmetros de atividades*: um tipo especial de nó para modelagem de parâmetros de entrada e saída das atividades. Podem ser normais, modelando entradas e saídas normais, ou excepcionais, para modelagem de saídas excepcionais (os excepcionais são marcados com um triângulo).

Neste trabalho, o diagrama é baseado na lógica do processo de negócios implementado pelo componente, similarmente ao proposto em [19]. O diagrama identifica possíveis seqüências de execução para os métodos das interfaces públicas do componente, e foi dividido em dois níveis: o nível principal ilustra o fluxo lógico de execução entre os métodos da interface provida do componente. O segundo (diagrama de detalhamento), ilustra o fluxo de execução ocasionado pela invocação de um método, modelando o lançamento de exceções de interface e a invocação dos métodos das interfaces requeridas. Como cada nível tem uma semântica própria, serão descritos separadamente nas subseções a seguir.

Diagrama Principal

Este diagrama representa a relação de precedência entre os métodos da interface provida pelo componente sob teste (CST). A Figura 5.3 ilustra o diagrama correspondente à interface `IAirFlowController` (Seção 4.1). O fluxo ilustra a seqüência lógica dos métodos públicos: o primeiro método é `setConfiguration()`, que realiza a configuração do componente. Duas situações podem suceder esse método: o lançamento da exceção de interface `InvalidConfigurationSetpoint`, ou a continuação do fluxo normal, marcada pelo valor válido de `O2_ref`, que pode derivar dois outros fluxos: a chamada do método `setCoalFeedRate()`, que regula a taxa de carvão da caldeira, ou `timeStep()`, que realiza o monitoramento das condições gerais da caldeira.

Neste diagrama são modeladas todas as seqüências possíveis, representando os diferentes cenários de uso dos serviços do componente. Cada ação representa um método da interface provida do componente, através de sua assinatura. Caso o método possua mais de uma saída possível, como o lançamento de uma exceção ou o retorno normal, por exemplo, é utilizada a ação de chamada de comportamento, como nos métodos do diagrama da Figura 5.3. Não deve haver duas ações representando o mesmo método neste diagrama, o que causaria inconsistência na representação.

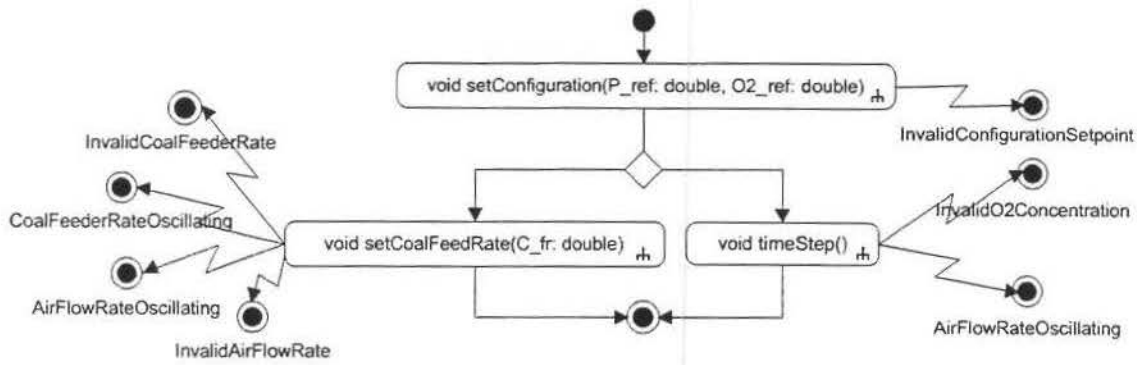

Figura 5.3: Diagrama de atividades da interface provida `IAirFlowController`.

As arestas representam a relação de precedência entre os métodos. A existência de mais de uma aresta incidente em uma ação significa que o método depende da execução de todos os que estão ligados a ele através desse tipo de aresta.

Já a aresta excepcional é utilizada somente para ligar uma ação a um nó final, quando este nó final representa uma saída excepcional. As condições para o lançamento de cada exceção são descritas no diagrama de detalhamento. Nem as arestas normais nem as excepcionais possuem condições de guarda no diagrama principal.

Todas as saídas possíveis devem ter um nó final correspondente. Caso represente um fluxo excepcional, os nós finais são marcados com o nome da exceção lançada, enquanto os nós finais representando saídas normais não trazem nenhuma marcação.

Diagrama de Detalhamento

Estes diagramas detalham as ações de chamada de comportamento. São utilizados aqui para representar o fluxo de execução ocasionado por um método da interface provida, caso esse método tenha mais de uma saída possível. Sua função é modelar o comportamento do método para que todas as suas possíveis saídas possam ser representadas.

Apesar de detalhar o comportamento de um método especificamente, a construção do diagrama não segue uma estratégia caixa branca. O comportamento modelado é relativo apenas às saídas produzidas pelo método, a partir de valores recebidos como entrada ou de interação com métodos públicos das interfaces requeridas (inclusive tratadores de exceção, caso sejam visíveis externamente ao componente). Assim, somente o comportamento que pode ser observado a partir das interfaces do componente é representado, similarmente ao representado na forma de diagrama de colaboração no processo *UMLComponents* [22].

A partir do diagrama da interface `IAirFlowController`, ilustrado na Figura 5.3, devem ser construídos três diagramas de detalhamento, já que todos os três métodos são especificados em ações de chamada de comportamento. A Figura 5.4 ilustra o diagrama

do método `setCoalFeedRate()`.

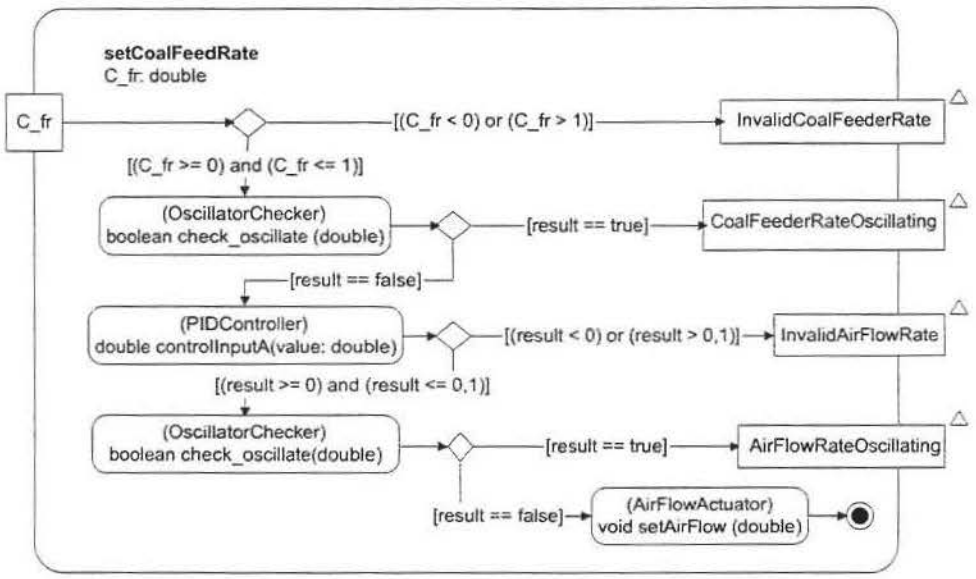

Figura 5.4: Diagrama de detalhamento do método setCoalFeedRate.

Como exemplificado na Figura 5.2b, neste diagrama é utilizada a notação de atividade, reforçando sua ligação com o método que está sendo detalhado. Seu fluxo pode ser dividido em duas partes: tratamento de valores de entrada, com o lançamento de exceções de interface, e interação com interfaces requeridas, com o lançamento de exceções externas.

A primeira parte do fluxo é referente ao tratamento dos valores de entrada. Os nós de parâmetros de entrada são diretamente ligados às saídas correspondentes, por arestas com condições de guarda especificadas na linguagem OCL - *Object Constraint Language* (descrita posteriormente neste Capítulo).

O parâmetro `C_fr` (Figura 5.4) gera dois fluxos, direcionados pelas condições de guarda: caso seu valor esteja entre zero e um, segue-se a execução do fluxo do método. Caso seu valor seja menor que zero ou maior que um, é considerado inválido, resultando no lançamento da exceção de interface `InvalidCoalFeederRate`.

Caso o método não possua parâmetros de entrada, como o método `timeStep()` (Figura 5.3), o diagrama de detalhamento começa com um nó inicial.

Após o tratamento dos valores de entrada, são tratados os retornos das interfaces requeridas. Caso não haja interação com interfaces requeridas, como o método `setConfiguration()`, o fluxo termina normalmente em um nó final.

Para a modelagem da interação com as interfaces requeridas é utilizada a notação de partição, que representa cada uma das interfaces requeridas pelo método. As ações representam a chamada a cada um dos métodos requeridos. Neste diagrama, caso um

método seja chamado mais de uma vez, deve ser utilizada uma ação diferente para cada uma das chamadas.

As ações são seguidas por arestas com condições de guarda relativas ao retorno do método requerido, que determinam o prosseguimento do fluxo. Na Figura 5.4 por exemplo, o retorno da primeira chamada ao método `check_oscillate` pode gerar o lançamento da exceção `CoalFeederRateOscillating`, caso seja igual a *true*, ou a continuação do fluxo normal, com a chamada do método `controlInputA()`.

Cada uma das saídas do diagrama de detalhamento deve ser mapeada a uma aresta de saída do método no diagrama principal.

5.1.2 Procedimento para Geração de Testes

A obtenção dos casos de teste é dividida em quatro fases. A Figura 5.5 mostra os artefatos utilizados em cada fase. A primeira consiste na transformação do diagrama de atividades com o fluxo dos métodos da interface provida em um grafo orientado. A segunda é o percurso deste grafo, obtendo um conjunto de caminhos no grafo dos quais os casos de teste serão extraídos. A terceira é a especificação dos casos de teste em uma linguagem intermediária, e a quarta e última é a implementação destes casos de teste na linguagem de programação correspondente.

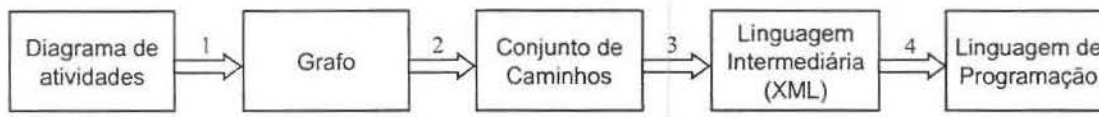

Figura 5.5: Artefatos usados na geração automática dos casos de teste.

Essas fases são descritas a seguir.

Diagrama de Atividades → Grafo

A primeira atividade é a tradução do diagrama de atividades da interface provida para um grafo orientado. A relação entre os itens do diagrama e os itens do grafo são descritas na Tabela 5.1. A Figura 5.6 mostra, à direita, o grafo correspondente ao diagrama à esquerda. O grafo de precedência é construído da seguinte forma:

1. O nó inicial é convertido no vértice com rótulo 0.

2. As ações são convertidas em vértices com rótulos correspondentes ao seu conteúdo no diagrama de atividades. Caso seja uma ação de chamada de comportamento, seu rótulo é acrescido com a palavra "detalhe". Os vértices oriundos de ações de chamada de comportamento estão designados como Di no grafo da Figura 5.6.

3. As ações entre *fork* e *join* são agrupadas em um vértice rotulado como Ci, i = 1 a k, onde k é o número de *fork-join* do diagrama.

4. Os nós finais representando saídas normais são convertidos em vértices com rótulos Ni conforme pode ser visto na Figura 5.6.

5. Os nós finais representando o lançamento de exceções são convertidos em vértices rotulados com o nome da exceção lançada. Na Figura 5.6, esses vértices são representados como Ei.

Item Diagrama	Item Grafo
Ação	Vértice Vn
Ação de Chamada de Comportamento	Vértice Dn
Nó inicial	Vértice 0
Nó final normal	Vértice Nn
Nó final com exceção	Vértice Nn
Fluxo Concorrente	Vértice Cn
Aresta	Aresta

Tabela 5.1: Correspondência entre os itens do diagrama de atividades e do grafo resultante.

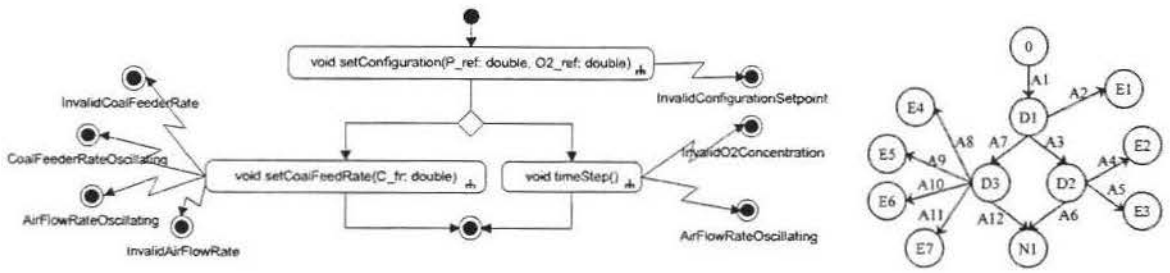

Figura 5.6: Transformação do diagrama da interface `IAirFlowController` no grafo correspondente.

Grafo → Conjunto de Caminhos

O objetivo desta etapa é a geração de caminhos que cubram todas as arestas do grafo produzido anteriormente, gerando assim casos de teste que exercitem todas as possíveis seqüências de métodos. Um caminho em um grafo é definido como uma seqüência de vértices, sendo que cada vértice da seqüência possui uma aresta que o liga ao vértice seguinte.

Os caminhos devem ser gerados a partir da execução de um algoritmo de percurso em grafos. Nesta dissertação, o algoritmo utilizado nos estudos de caso foi o algoritmo de

busca em profundidade [23], que procura traçar caminhos no grafo que atinjam todos os vértices acessíveis a partir de um vértice inicial.

O vértice inicial (0) é escolhido como o vértice de partida do algoritmo, e todos os caminhos devem terminar em algum vértice final (Ni ou Ei). Os caminhos obtidos a partir do grafo da Figura 5.6 são exibidos na Figura 5.7: todos começam no vértice 0 e terminam em algum dos vértices Ni ou Ei. Neste caso, foram gerados 9 caminhos.

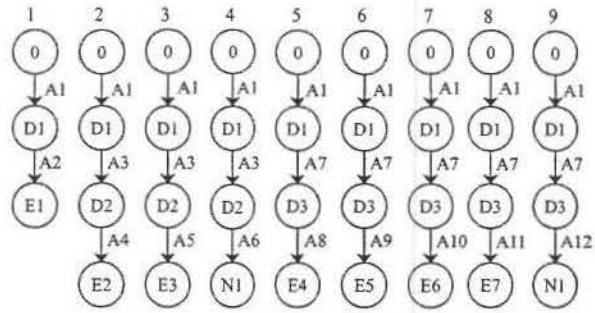

Figura 5.7: Caminhos obtidos a partir da busca em profundidade no grafo da Figura 5.6.

Considerando o rótulo real dos vértices (assinatura do método no caso de Di e nome da exceção no caso de Ei), é possível mapear cada um dos caminhos obtidos para a seqüência de métodos e resultado esperado correspondentes. Neste caso, todas as assinaturas de métodos são seguidas pela palavra "detalhe", pois possuem digramas de detalhamento. e a maioria dos resultados esperados são lançamentos de exceções, já que os caminhos terminam em vértices Ei. O mapeamento é listado na Tabela 5.2.

	Métodos	Resultado Esperado do Caso de Teste
1	void setConfiguration(..) detalhe	InvalidConfigurationSetpoint
2	void setConfiguration(..) detalhe, void timeStep() detalhe	InvalidO2Concentration
3	void setConfiguration(..) detalhe, void timeStep() detalhe	AirFlowRateOscillating
4	void setConfiguration(..) detalhe, void timeStep() detalhe	(void)
5	void setConfiguration(..) detalhe, void setCoalFeedRate(..) detalhe	InvalidCoalFeederRate
6	void setConfiguration(..) detalhe, void setCoalFeedRate(..) detalhe	CoalFeederRateOscillating
7	void setConfiguration(..) detalhe, void setCoalFeedRate(..) detalhe	AirFlowRateOscillating
8	void setConfiguration(..) detalhe, void setCoalFeedRate(..) detalhe	InvalidAirFlowRate
9	void setConfiguration(..) detalhe, void setCoalFeedRate(..) detalhe	(void)

Tabela 5.2: Casos de teste extraídos da árvore geradora.

A geração de caminhos para concorrência está além do escopo deste trabalho. Assim, os vértices que representam o fluxo entre *forks* e *joins* (Ci) são considerados como uma caixa preta, do mesmo modo que os outros vértices. Durante a realização dos testes este vértice deve ser substituído por seqüências de métodos pertencentes ao fluxo concorrente.

O teste de *loops* também será tratado em trabalhos futuros. Atualmente, este comportamento não é representado nos diagramas de atividades, e os casos de teste

que exercitam tais cenários são gerados manualmente, seguindo as diretrizes propostas por Binder, na abordagem *Round-trip Scenario Test* [16]. Esta abordagem indica que os testes devem exercitar uma iteração dos *loops*, o número máximo de iterações, e um número de iterações que represente a média de utilização ou uma saída diferente.

Especificação dos Casos de Teste

Na fase de especificação dos casos de teste, os caminhos gerados são descritos em uma linguagem intermediária. A decisão favorece a portabilidade dos casos de teste que, a partir do formato intermediário, podem ser traduzidos para qualquer linguagem de programação.

Foi escolhida a linguagem XML como linguagem intermediária, por ser amplamente difundida, facilmente entendida por pessoas e ferramentas, e por possibilitar a verificação da consistência do modelo, isto é, se o modelo foi estruturado corretamente. Para a tradução dos casos de teste obtidos a partir do conjunto de caminhos para XML, foi criado um modelo de dados baseado nas propostas de [21, 30, 45], apresentado no Apêndice B.

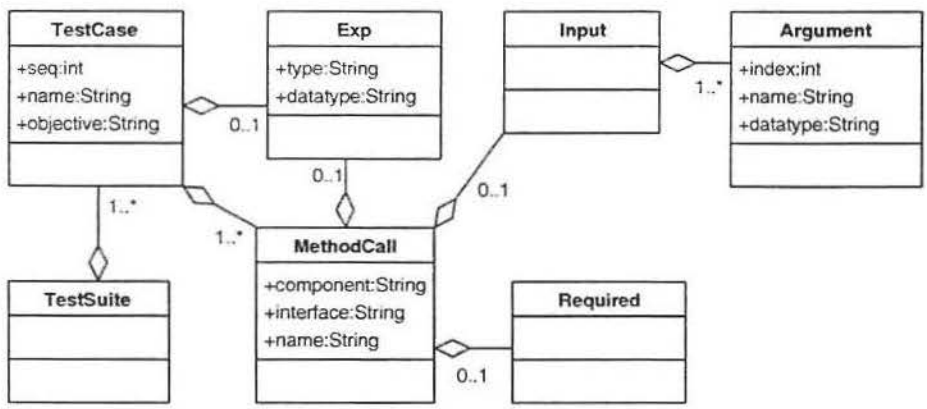

Figura 5.8: Modelo de dados para o XML de especificação dos casos de teste.

Para a criação dos casos de teste, as *tags* utilizadas são exibidas na Figura 5.8, na forma de um diagrama de classes. O conjunto de casos de teste é inicialmente formado por uma *tag* <TestSuite>, que representa uma coleção de testes de uma determinada interface, e contém todos os casos de teste gerados a partir dos caminhos. Cada um dos dos casos de teste é traduzido para um <TestCase>, formado por:

1. Uma ou mais *tags* <MethodCall>, que representam a chamada de um método do caso de teste, e são extraídas dos vértices Vi e Di;

2. Uma *tag* <Exp>, que armazena o resultado esperado do caso de teste e é obtida a partir dos vértices Ni ou Ei (por exemplo, os resultados esperados listados na Tabela 5.2).

A *tag* <MethodCall> contém as informações necessárias para a chamada do método, extraídas de sua assinatura:

1. O nome do método, da interface e do componente são armazenados como atributos de <MethodCall>;

2. A *tag* <Input> agrupa as *tags* <Argument> que, por sua vez, contêm as informações relativas a cada um dos argumentos do método (seu nome, ordem e tipo);

3. A *tag* <Exp> armazena o valor esperado de retorno deste método específico (diferentemente da *tag* contida em <TestCase>, que armazena o resultado esperado para toda a seqüência de métodos);

4. A *tag* <Required>, incluída caso o método tenha sido obtido a partir de uma ação de chamada de comportamento, isto é, de um vértice Di (cujo rótulo termine em "detalhe"). Nesta fase esta *tag* é incluída vazia, e será preenchida posteriormente com informações extraídas do diagrama da interface requerida do método.

A especificação em XML do caso de teste extraído do caminho 6 da Tabela 5.2 é apresentada na Figura 5.9.

Implementação dos Casos de Teste

A última fase da geração dos casos de teste é sua tradução para uma linguagem de programação. A portabilidade proporcionada pela geração de testes a partir dos modelos permite que a linguagem de programação seja considerada apenas na última etapa, para a tradução dos casos de teste em XML para um formato executável.

A tradução dos casos de teste é realizada com os seguintes passos:

1. Cada *tag* <TestCase> é transformada em um método, contendo um bloco protegido, o chamado *try-catch*;

2. Cada *tag* <MethodCall> é traduzida para a chamada ao método correspondente (os valores de seus parâmetros são obtidos a partir das *tags* <Argument>), dentro do bloco *try*.

3. Caso um método (*tag* <MethodCall>) possua <Exp type="normal">, isto é, o resultado esperado do método seja normal, seu valor de retorno é atribuído a uma

```
<TestSuite>
 <TestCase seq="0" name="testIP1" objective="">
  <MethodCall component="AirFlowController" interface="IAirFlowController"
                                            name="setConfiguration">
   <Input>
    <Argument index="0" name="P_ref" datatype="double"></Argument>
    <Argument index="1" name="O2_ref" datatype="double"></Argument>
   </Input>
  </MethodCall>
  <MethodCall component="AirFlowController" interface="IAirFlowController"
                                            name="setCoalFeedRate">
   <Input>
    <Argument index="0" name="C_fr" datatype="double"></Argument>
   </Input>
   <Required>  </Required>
   <Exp type="exceptional" datatype="CoalFeederRateOscillating"></Exp>
  </MethodCall>
 <Exp type="exceptional" datatype="CoalFeederRateOscillating"></Exp>
 </TestCase>
</TestSuite>
```

Figura 5.9: XML relativo ao sexto caminho extraído do diagrama da interface `IAirFlowController`.

variável temporária e comparado ao valor contido na *tag* <MethodCall>.<Exp>, mesmo que o método não seja o último da seqüência que compõe o caso de teste. Caso o resultado seja o lançamento de uma exceção, a seqüência de métodos será terminada, sendo assim o resultado esperado do método será o mesmo do caso de teste. Neste caso, o valor é verificado apenas como resultado do caso de teste.

4. Após todas a tradução de todos as *tags* <MethodCall>, isto é, da chamada de todos os métodos da seqüência do caso de teste, o resultado esperado do caso de teste é verificado:

   (a) Caso <TestCase>.<Exp type = "normal">, o resultado esperado é verificado ao final do bloco *try*, e no bloco *catch* é diretamente lançada a exceção de falha do caso de teste.

   (b) Caso <TestCase>.<Exp type = "exceptional">, o resultado esperado é verificado dentro do bloco *catch*, e ao final do bloco *try* é lançada a exceção de falha do caso de teste.

Neste trabalho foi utilizada a linguagem Java, em particular o *framework* JUnit [49], internamente ao ambiente Eclipse. O JUnit é um *framework* voltado para execução au-

tomática de casos de teste, utilizado especialmente por metodologias ágeis de desenvolvimento para a construção de testes de unidade pelos próprios desenvolvedores.

Apesar do foco em testes de unidade, o JUnit pode ser utilizado em outras fases, como o teste de componentes. O *framework* define como estruturar os casos de teste, e fornece ferramentas para executá-los [7]. Os conjuntos de casos de teste são armazenados em classes descendentes da classe `TestCase` do *framework*, em métodos cujos nomes se iniciam em "test". Os resultados recebidos são comparados aos resultados esperados através de métodos da classe `Assert`.

A tradução do caso de teste em XML da Figura 5.9 é apresentado na Figura 5.10. As linhas 4 e 5 contêm as chamadas aos métodos da seqüência do caso de teste. Os valores dos parâmetros serão completados mais adiante neste capítulo, na Seção 5.3. As *tags* <Required> também foram ignoradas nesta tradução, sendo abordadas na próxima Seção.

Como o resultado esperado do caso de teste é o lançamento de uma exceção, caso a execução atinja o fim do bloco *try*, significa que nenhuma exceção foi lançada. Assim, a linha 7 contém a chamada ao método `assertFalse` do *framework* JUnit, que sinaliza o veredicto "falhou" ao caso de teste. Na linha 10, internamente ao bloco *catch*, a biblioteca do JUnit é novamente utilizada, desta vez para comparação da exceção lançada com o resultado esperado do caso de teste.

```
1  public void testIP1() {
2    try {
3      //Chamadas aos métodos da interface provida, sem os dados de teste
4      afc.setConfiguration(..,..);
5      afc.setCoalFeedRate(..);
6      //Caso de teste falha se uma exceção não é lançada:
7      assertFalse("Excecao nao foi lancada", true);
8    } catch (Exception e) {
9      //Verificação se o resultado é o esperado (neste caso, a exceção lançada):
10     assertEquals(e.getClass().getName(),"CoalFeederRateOscillating");
11   }
12 }
```

Figura 5.10: Tradução para a linguagem Java do caso de teste em XML apresentado na Figura 5.9.

A execução dos casos de teste é coordenada pelo *driver* de testes, detalhado na Seção 5.6.

5.2 Elaboração dos *Stubs*

Para o teste dos componentes isoladamente, é necessário que suas dependências sejam resolvidas por meio de *stubs*. *Stubs* são implementações simples que substituem os componentes requeridos, simulando seu comportamento. A presença de *stubs* é particularmente importante para o teste de componentes tolerantes a falhas, pois facilita a simulação de exceções nos componentes requeridos, possibilitando a observação do comportamento do componente sob teste nestas situações.

Há diversas propostas para a implementação de *stubs*, como por exemplo os padrões *Server Stub* e *Server Proxy* [16], e a abordagem *Virtual Mock Objects* [57]. O padrão *Server Stub* propõe a substituição dos componentes requeridos por implementações simplificadas que possam ser controladas. Já a *Server Proxy* possui dois modos de operação: simulação dos componentes requeridos, similarmente ao *Server Stub*, ou apenas interceptando as chamadas aos componentes reais (particularmente útil durante testes de integração).

Por sua vez, a abordagem *Virtual Mock Objects* não propõe uma nova implementação dos componentes requeridos. Para o controle dos valores retornados, as chamadas às interfaces requeridas são interceptadas durante a execução, utilizando programação orientada a aspectos (Seção 4.4.1), e os valores são modificados de acordo com o cenário de testes desejado.

Neste trabalho, como serão realizados apenas testes de componentes isolados, foi escolhida a abordagem *Server Stub* para implementação. Pelo fato dos testes serem gerados automaticamente, a simplicidade para preparação dos *stubs* é uma característica importante. Como a estrutura dos *Virtual Mock Objects* apresenta maior complexidade, o formato *Server Stub* foi escolhido para geração automática. A implementação proposta é descrita na próxima subseção.

A subseção seguinte apresenta diretrizes para a sincronização entre *drivers* e *stubs*. A cada novo caso de teste, os *stubs* participantes são instanciados e preenchidos com os valores a serem retornados durante a execução do caso de teste. Os dados para sincronização são gerados a partir dos diagramas de detalhamento definidos na Seção 5.1.1, e incorporados ao XML de cada caso de teste.

5.2.1 Implementação dos *Stubs*

A estrutura dos *stubs* é gerada a partir das interfaces requeridas do CST. Os *stubs* gerados seguem o formato *Server Stub* [16], substituindo por inteiro os componentes requeridos. Sua função é, basicamente, retornar um valor específico quando um método da interface requerida for invocado. Os valores a serem retornados devem ser atribuídos ao *stub*, antes da chamada ao método testado, na ordem em que serão retornados.

A implementação proposta permite que os valores sejam atribuídos para os *stubs* antes da execução de cada caso de teste, ou até mesmo de cada método chamado durante o teste. Estes valores são armazenados em filas para cada método, e vão sendo consumidos a medida que o método é chamado. Esta estrutura permite que o mesmo método seja chamado várias vezes durante o teste e retorne diferentes valores. Os valores são armazenados na forma de objetos, permitindo que sejam retornados tanto valores normais quanto exceções.

Para a implementação desta estrutura, deve ser criada uma classe que implemente a interface requerida (uma ou mais interfaces podem ser opcionalmente agrupadas) e, para cada método, devem ser criados:

1. Uma fila de objetos como atributo da classe, para armazenamento de todos os valores a serem retornados pelo *stub* durante a execução do teste.

2. Um método "`setMethod`", que receba um objeto como parâmetro e o adicione à fila correspondente ao método.

3. Um método "`setMethodEmpty`", para remoção de todos os elementos da fila do método, limpando-a para o próximo teste. Outra opção é a criação de um método único de limpeza das filas, caso todas sejam reiniciadas a cada caso de teste, ou mesmo a criação de uma nova instância a cada caso de teste.

4. Implementação do método, retornando o primeiro elemento da fila (e removendo-o). Caso o elemento seja uma exceção, esta deverá ser lançada.

A Figura 5.11 ilustra a implementação na linguagem Java do *stub* referente à interface `OscillatorChecker`, requerida pelo componente `IAirFlowController`. Esta interface simula o comportamento de um sensor para monitoração de oscilações bruscas de temperatura, e retorna o valor *true* caso essas oscilações sejam constatadas.

5.2.2 Preparação dos *Stubs*

Após a implementação de cada interface requerida como *stub*, são preparados os mecanismos para sincronização entre os *stubs* e a execução dos casos de teste. Antes da execução de cada método da seqüência de teste, os *stubs* que serão invocados pelo método são preparados para retornar os valores que possibilitem a execução do cenário específico.

As informações referentes aos *stubs* são incluídas nas *tags* `<Required>` (Figura 5.9), de cada um dos métodos da seqüência de teste. Essas informações são extraídas dos diagramas de detalhamento dos métodos, que são percorridos similarmente ao diagrama principal. A tradução do diagrama para o código executável segue os mesmos passos descritos na Figura 5.5: Diagrama de Atividades → Grafo → Conjunto de Caminhos

→ XML → Linguagem de Programação. As etapas serão sumariamente descritas nas subseções seguintes, na forma de exemplos.

```
1  public class StubOscillatorChecker implements OscillatorChecker{
2
3   //Um atributo vector para cada metodo, pra guardar os valores que ele deve retornar
4   private Vector check_oscillateVector = new Vector(10);
5
6   //Um set para cada metodo, para incluir valores no Vector
7   public void setCheck_oscillate(Object ito) {
8    check_oscillateVector.add(ito);
9   }
10
11  // Reinicia os vetores
12  public void setEmpty () {
13   check_oscillateVector.removeAllElements();
14  }
15
16  //Implementação do método da interface requerida: apenas retornam os valores
17  //contidos no vector correspondente
18  public boolean check_oscillate(double value) throws Exception{
19   Object o = check_oscillateVector.firstElement();
20   check_oscillateVector.removeElementAt(0);
21   //retorno normal
22   if (o.getClass().getName().equalsIgnoreCase("java.lang.Boolean")) {
23    //Transformação do objeto para o valor primitivo boolean
24    Boolean it = (Boolean)o;
25    return it.booleanValue();
26   }
27   //lançamento de exceção
28   else if (o instanceof java.lang.Exception)) {
29     Exception e = (Exception)o;
30     throw e;
31   }
32   //Caso ocorra alguma exceção durante a execução
33   throw (new Exception("Erro interno do stub OscillatorChecker."));
34  }
35 }
```

Figura 5.11: Código do *stub* referente à interface requerida `OscillatorChecker`.

Procedimento para Obtenção das Interações

Assim como o diagrama principal, são extraídos caminhos dos diagramas de detalhamento, que representam diferentes situações de interação com os *stubs*. O diagrama é transformado em um grafo, que é posteriormente percorrido por um algoritmo.

A transformação do diagrama do método `setCoalFeederRate` em um grafo é ilustrada na Figura 5.12, que mostra à direita o grafo correspondente ao diagrama da esquerda.

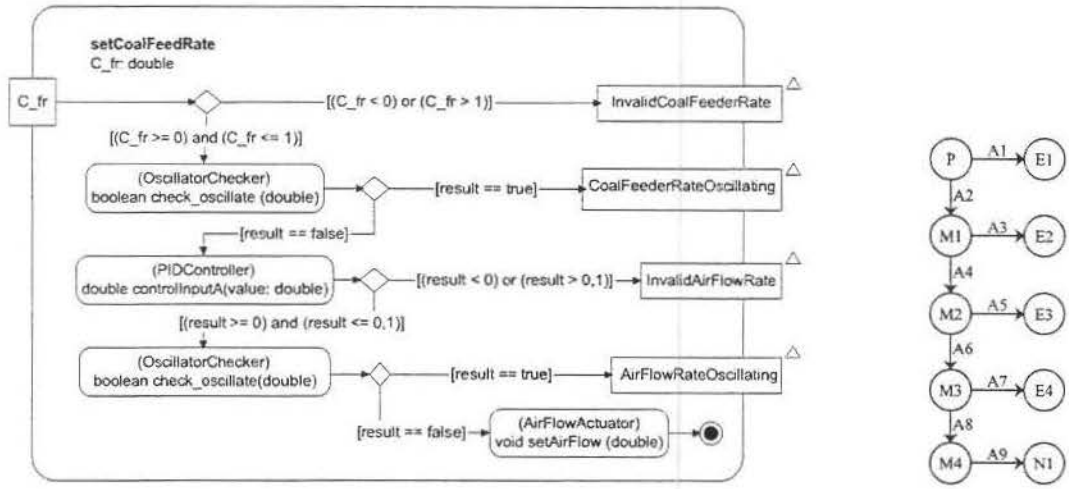

Figura 5.12: Transformação do diagrama do método `setCoalFeederRate` no grafo correspondente.

Item Diagrama	Item Grafo
Ação	Vértice Mn
Nó inicial	Vértice 0
Nó final normal	Vértice Nn
Nó de Parâmetro de Entrada	Vértice P
Nó de Parâmetro de Saída Normal	Vértice Sn
Nó de Parâmetro de Saída Excepcional	Vértice En
Fluxo Concorrente	Vértice Cn
Aresta	Aresta An
Decisão	(removida)
Junção	(removida)

Tabela 5.3: Correspondência entre os itens do diagrama de detalhamento e do grafo resultante.

A relação entre os itens do diagrama e os itens do grafo são descritas na Tabela 5.3. Os nós iniciais, finais, ações e fluxos concorrentes são traduzidos da mesma forma que no diagrama anterior (no grafo, as ações são representadas pelos vértices Mn, por questão de espaço). Os demais elementos são transformados de acordo com as seguintes diretrizes:

1. O nós de parâmetros de entrada são convertidos em um vértice rotulado com o nome do parâmetro do método. Caso haja mais de um nó, eles são agrupados, sendo os

parâmetros separados por vírgulas. Na Figura 5.12, o vértice é representado como P.

2. O nós de parâmetros de saída normal são convertidos em vértices rotulados com o valor retornado, contido no nó.

3. O nós de parâmetros de saída de exceção são convertidos em vértices rotulados com o nome da exceção lançada, assim como os nós finais excepcionais do diagrama principal (que não existem neste diagrama). Na Figura 5.12, esses vértices são representados como Ei.

4. As decisões e junções são removidas, e as condições de guarda são armazenadas junto às arestas.

Após a transformação do diagrama em grafo, são gerados caminhos que cubram todas as suas arestas, podendo ser utilizado o mesmo algoritmo do percurso do diagrama principal. Neste caso, o vértice inicial pode ser o (0) ou o (P), sendo que os caminhos devem terminar em algum dos vértices finais (Ni ou Ei). Os caminhos obtidos a partir do grafo da Figura 5.12 são exibidos na Figura 5.13, totalizando 5 caminhos.

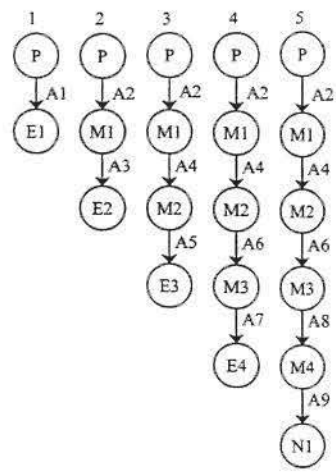

Figura 5.13: Caminhos obtidos a partir da busca em profundidade no grafo da Figura 5.12 (direita).

A Tabela 5.4 ilustra as seqüências de interação do método com as interfaces requeridas, obtidos a partir dos caminhos considerando-se o rótulo real dos vértices e arestas (assinatura do método para Mi, nome da exceção para Ei).

	Métodos e Condições de Guarda	Retorno do Método
1	[(C_fr < 0) or (C_fr > 1)]	InvalidCoalFeederRate
2	[(C_fr >= 0) and (C_fr <= 1)], (OscillatorChecker) boolean check_oscillate (double), [result == true]	CoalFeederRateOscillating
3	[(C_fr >= 0) and (C_fr <= 1)], (OscillatorChecker) boolean check_oscillate (double), [result == false], (PIDController) double controlInputA(value: double), [(result < 0) or (result > 0,1)]	InvalidAirFlowRate
4	[(C_fr >= 0) and (C_fr <= 1)], (OscillatorChecker) boolean check_oscillate (double), [result == false], (PIDController) double controlInputA(value: double), [(result >= 0) and (result <= 0,1)], (OscillatorChecker) boolean check_oscillate(double), [result == true]	AirFlowRateOscillating
5	[(C_fr >= 0) and (C_fr <= 1)], (OscillatorChecker) boolean check_oscillate (double), [result == false], (PIDController) double controlInputA(value: double), [(result >= 0) and (result <= 0,1)], (OscillatorChecker) boolean check_oscillate(double), [result == false], (AirFlowActuator) void setAirFlow (double)	(void)

Tabela 5.4: Interações do método `setCoalFeederRate` com as interfaces requeridas extraídas dos caminhos do grafo.

Especificação das Interações

Nesta fase, os caminhos gerados são traduzidos para a linguagem XML, e incorporados ao XML dos casos de teste correspondentes. O XML intermediário, relativo a geração das interações é exibido na Figura 5.14, na forma de um diagrama de classes. As regras para a construção do XML completo são apresentadas no Apêndice B.

O XML é formado por uma *tag* <Required>, que posteriormente substituirá a *tag* correspondente em <MethodCall>, e representa a interação do método com os *stubs* em um determinado cenário. Sendo assim, cada <Required> é formada por:

1. Uma ou mais *tags* <Interaction>, que representam a chamada a um método da interface requerida. Cada vértice Mi gera uma *tag* <Interaction>

2. Cada *tag* <Interaction>, por sua vez, contém uma *tag* <Stimulus>, que armazena o valor que deve ser retornado pelo método de <Interaction> quando chamado durante a execução do método de <MethodCall>.

3. Uma *tag* <Exp>, que armazena o resultado esperado de retorno do método de <MethodCall>, obtido a partir dos vértices Ni, Si ou Ei.

Os dados (contidos nas *tags* <Stimulus>)serão gerados em uma fase posterior, após a finalização do XML dos casos de teste. As regras para sua formação, porém, são obtidas

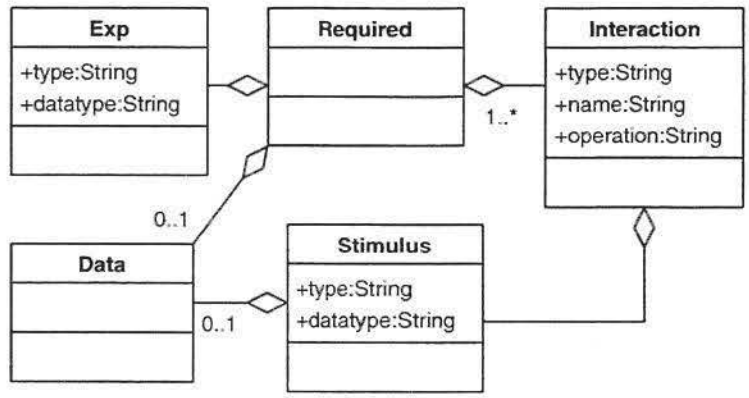

Figura 5.14: Modelo de dados para o XML de especificação das interações dos métodos.

durante o percurso a partir das condições de guarda, que devem ser armazenadas no XML. Por isso foi criada uma *tag* temporária, <Data>, que armazena a condição de guarda durante a geração do XML das interações e é substituída durante a geração dos dados de teste pelos dados correspondentes.

No XML intermediário gerado nesta fase, a *tag* <Data> pode ser incluída nas *tags* <Required> e <Stimulus>. Na *tag* <Interaction>, a *tag* <Data> armazena as condições de guarda relativas aos parâmetros de entrada, contidas nas arestas que se originam em nós de parâmetros de entrada. Na *tag* <Stimulus>, armazena as condições de guarda relativas aos valores a serem retornados pelos métodos requeridos, contidas nas arestas que se originam no método correspondente.

A especificação das interações do método `setCoalFeederRate` em XML é ilustrada na Figura 5.15. Foi incluída uma *tag* <Data> no início, que contém a regra da aresta A2; uma *tag* <Interaction> correspondente ao método `check_oscillate`; e uma *tag* <Stimulus> contendo uma *tag* <Data>, com a regra da aresta A3. A *tag* <Exp> armazena o valor do vértice E2.

Integração das Especificações

Após a geração do XML correspondente a cada um dos caminhos dos diagramas de detalhamento, as especificações dos casos de teste em XML são completadas com as informações sobre as interações. As *tags* <Required> do XML dos casos de teste são substituídas pelas *tags* preenchidas com os caminhos do diagrama de detalhamento.

O preenchimento é realizado de acordo com as *tags* <Exp> de <MethodCall> e de <Required>, que devem ser equivalentes. A equivalência é determinada a partir das *tags* <Required>.<Exp> e <MethodCall>.<Exp>. Isto é, caso os valores de ambas sejam

```
<Required>
  <Data>[(C_fr >= 0) and (C_fr <= 1)]</Data>
  <Interaction type="interface" name="OscillatorChecker" operation="check_oscillate">
    <Stimulus type="normal" datatype="boolean">
      <Data>[result == true]</Data>
    </Stimulus>
  </Interaction>
  <Exp type="exceptional" datatype="CoalFeederRateOscillating"></Exp>
</Required>
```

Figura 5.15: XML relativo ao segundo caminho extraído do diagrama do método `setCoalFeederRate`.

iguais, a *tag* <MethodCall>.<Required> pode ser substituída.

Cada *tag* <Required> extraída do diagrama de detalhamento é examinada e é buscada a *tag* <MethodCall> equivalente. A *tag* <Required> apresentada na Figura 5.15, por exemplo, é equivalente à segunda *tag* <MethodCall> do XML da Figura 5.9, já que ambas tem os mesmos valores nos atributos da *tag* <Exp>.

A integração dos arquivos XML é exibida na Figura 5.16. Após a integração, a *tag* <Required>.<Exp> é suprimida, já que seu valor é o mesmo de <MethodCall>.<Exp>.

```
<MethodCall component="AirFlowController" interface="IAirFlowController"
                                          name="setCoalFeedRate">
 <Input>
  <Argument index="0" name="C_fr" datatype="double"></Argument>
 </Input>
 <Required>
  <Data>[(C_fr >= 0) and (C_fr <= 1)]</Data>
  <Interaction type="interface" name="OscillatorChecker" operation="check_oscillate">
    <Stimulus type="normal" datatype="boolean">
      <Data>[result == true]</Data>
    </Stimulus>
  </Interaction>
  <Exp type="exceptional" datatype="CoalFeederRateOscillating"></Exp>
 </Required>
 <Exp type="exceptional" datatype="CoalFeederRateOscillating"></Exp>
</MethodCall>
```

Figura 5.16: XML relativo à chamado ao método `setCoalFeederRate` já com as informações sobre os *stubs*.

Caso a quantidade de *tags* <Required> não seja compatível à quantidade de *tags* <MethodCall>, são reutilizadas *tags* já finalizadas.

Implementação das Interações

Na tradução do XML integrado para a linguagem de programação, a única mudança em relação aos passos descritos na Seção 5.1.2 é a inclusão de comandos para a sincronização dos *stubs* antes da chamada de cada método. Para cada *tag* <Interaction> contida em um <MethodCall>.<Required>, o método "set"do *stub* correspondente é invocado, incluindo na fila do *stub* o valor a ser retornado (obtido da *tag* <Stimulus>).

Após as modificações do XML do caso de teste apresentadas na Figura 5.16, o código Java exibido na Figura 5.10 foi modificado apenas com a inclusão da chamada ao método `setCheck_oscillate(..)` (ainda sem dados), da interface `OscillatorChecker`. Esta chamada é posicionada antes da invocação ao método `setCoalFeedRate()`, preparando o ambiente para sua execução.

5.3 Criação de Dados de Teste

Após a geração do XML contendo as informações sobre os casos de teste e suas dependências, ele é finalizado com a inclusão dos dados de teste. As *tags* <Argument> e <Stimulus> devem ser preenchidas com os valores a serem utilizados durante a execução, e que obedeçam possíveis condições de guarda associadas. As *tags* <Exp> também podem ser preenchidas com dados de teste, aumentando a precisão dos oráculos.

Para a geração de dados devem ser consideradas regras como partições de equivalência e análise de valores limites [63], por exemplo. Tanto dados simples quanto complexos podem ser incluídos nas *tags* correspondentes. Os dados simples, como inteiros por exemplo, são incluídos diretamente, como em <Stimulus>true<Stimulus>. Já dados complexos, que exigem a criação de objetos, são estruturados em uma *tag* <Object>, cujo modelo de dados é exibido na Figura 5.17.

A *tag* <Object> é utilizada para a criação de objetos, cujo estado pode ser preparado tanto através de atribuição direta quanto através de chamadas de métodos. Para atributos públicos, uma *tag* <Attribute> é utilizada para cada atributo, com o respectivo valor. Para uma chamada de método, é utilizada a *tag* <Method>, cujos parâmetros recebem os valores a partir das *tags* <Parameter>. Os valores dos parâmetros devem ser incluídos nas *tags* <Parameter>.

Um exemplo da utilização da *tag* <Object> no exemplo da caldeira de vapor é apresentado na Figura 5.18, que contém a *tag* <MethodCall> referente ao método `connectTop`. Como o argumento deste método é uma interface do tipo `IComponent`, foi utilizada a tag <Object> para a instanciação. A interface foi instanciada com a classe `Conn3`, que não possui atributos internos. Neste caso, o método a ser chamado é apenas o construtor do objeto, com a utilização da *tag* <Method>. Por ser um construtor, o atributo `datatype`

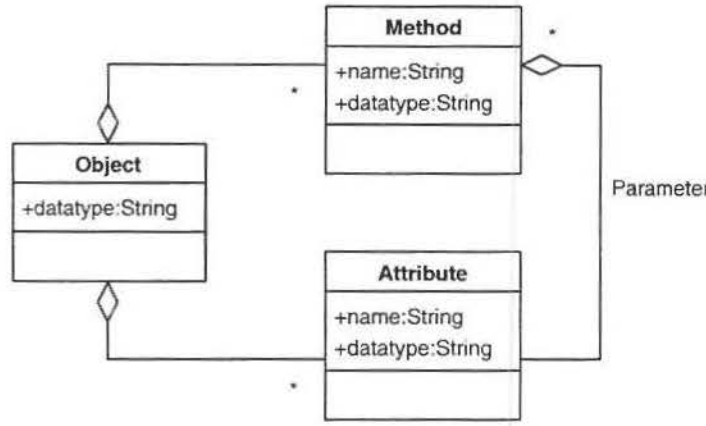

Figura 5.17: Modelo de dados para o XML de especificação dos objetos componentes dos dados de teste.

não é preenchido (para métodos sem valor de retorno, o atributo `datatype` é preenchido com o valor `void`).

```
1 <MethodCall component="AirFlowController" interface="IComponent" name="connectTop">
2   <Input>
3     <Argument index="0" name="brick" datatype="IComponent">
4       <Object datatype="IComponent">
5         <Method name="Conn3" datatype=""></Method>
6       </Object>
7     </Argument>
8   </Input>
9 </MethodCall>
```

Figura 5.18: Exemplo da utilização da *tag* <Object>.

Após a atribuição dos dados de teste, as *tags* <Data> são removidas, sendo substituídas por valores que obedeçam as condições de guarda. A Figura 5.19 apresenta o XML da Figura 5.9 finalizado, com valores de argumentos e retornos de operações requeridas.

As *tags* <Data> contidas em *tags* <Result> são substituídas diretamente pelos valores gerados. A substituição pode ser verificada na comparação entre as linhas 9 a 11 da Figura 5.16 e a linha 17 da Figura 5.19. Já as *tags* <Data> internas às *tags* <Interaction> não são substituídas diretamente. Como suas condições de guarda fazem referência a argumentos do método provido (uma condição pode relacionar diferentes

argumentos entre si), os valores gerados devem os valores devem ser distribuídos para as *tags* <Argument> correspondentes.

```
<TestSuite>
 <TestCase>
  <MethodCall component="AirFlowController" interface="IAirFlowController"
                                            name="setConfiguration">
   <Input>
    <Argument index="0" name="P_ref" datatype="double">0</Argument>
    <Argument index="1" name="02_ref" datatype="double">0</Argument>
   </Input>
  </MethodCall>
  <MethodCall component="AirFlowController" interface="IAirFlowController"
                                            name="setCoalFeedRate">
   <Input>
    <Argument index="0" name="C_fr" datatype="double">0</Argument>
   </Input>
   <Required>
    <Interaction type="interface" name="OscillatorChecker" operation="check_oscillate">
     <Stimulus type="normal" datatype="boolean">true</Stimulus>
    </Interaction>
   </Required>
   <Exp type="exceptional" datatype="CoalFeederRateOscillating">
    <!-- Apenas o tipo da exceção é verificado --!>
   </Exp>
  </MethodCall>
  <Exp type="exceptional" datatype="CoalFeederRateOscillating"></Exp>
 </TestCase>
</TestSuite>
```

Figura 5.19: XML de caso de teste da interface `IAirFlowController` com os dados das interações e os dados de teste.

5.4 Criação do Oráculo

Além da definição dos resultados esperados dentro dos casos de teste, também é incluída no componente a verificação de seu contrato durante a execução, seguindo a abordagem *Design-by-contract* (Seção 2.1.1). As assertivas são utilizadas como oráculo parcial, por não darem resultados exatos, porém servem como complemento aos resultados esperados dos casos de teste. Neste trabalho, são definidas em OCL na especificação contratual do componente, que é parte da arquitetura do componente testável.

Nos estudos de caso realizados, as assertivas foram mais detalhadas que os resultados esperados contidos nos casos de teste, pois poderiam ser utilizadas mesmo após o término

dos testes, monitorando o comportamento de todo o componente durante a execução. Além disso, facilitam a localização das falhas para posterior manutenção corretiva.

As subseções seguintes apresentam brevemente a linguagem OCL, descrevem o formato da especificação contratual do componente e como as bibliotecas do componente `Tester` podem ser geradas a partir da especificação.

5.4.1 Linguagem OCL

Um dos problemas da UML é a falta de formalismo em seus modelos, muitas vezes resultando em ambigüidades [42, 82]. Mesmo a adição de restrições em linguagem natural pode não ser suficiente para eliminação de todas as ambigüidades, motivando a construção da *Object Constraint Language* (OCL).

A OCL é uma linguagem de especificação, cujas expressões são relativas a modelos UML. Pode ser usada, por exemplo, para a especificação de condições de guarda, restrições em operações, contratos e conjuntos relativos a mensagens e ações.

Neste trabalho, a OCL será utilizada para a especificação de condições de guarda nos diagramas de atividades e dos contratos das interfaces dos componentes, através da especificação das invariantes, pré e pós-condições.

A estrutura principal de OCL é o contexto, que especifica a qual modelo a expressão se refere. É especificado pela palavra reservada `context`. Para a especificação de pré-condições, por exemplo, o contexto é o método relativo a essa pré-condição, especificado por `context` + nome do método.

A construção das expressões é favorecida pelo poder da linguagem OCL, que permite a representação desde expressões simples até expressões envolvendo exceções e a captura de mensagens, por exemplo. Pela sua extensão, neste trabalho serão explicados somente as construções utilizadas nos exemplos, comuns para a especificação de condições de guarda e contratos. Maiores informações sobre a OCL podem ser obtidas em [42, 82].

5.4.2 Especificação Contratual

A especificação contratual é parte da arquitetura do componente testável e contém os contratos das diferentes interfaces do componente, descritos em OCL. A especificação é dividida por interface, e para cada interface são descritos a sua invariante e as pré e pós-condições de seus métodos. A Figura 5.20 apresenta parte da especificação contratual da interface `IAirFlowController`.

A especificação segue a estrutura proposta pela linguagem OCL, que localiza as restrições com a determinação de contextos, utilizando a palavra reservada `context`. Nas linhas 1 e 4, respectivamente, a interface `IAirFlowController` é determinada como con-

```
1  context IAirFlowController
2   inv: true -- não há invariante
3
4  context IAirFlowController::setCoalFeedRate(C_fr: double)
5    pre: true -- não há pré-condição (programação defensiva)
6
7    norm.post: true
8      -- como o método retorna void, apenas as condições excepcionais são verificadas
9
10   InvalidCoalFeederRate.condition:
11        ((C_fr < 0) or (C_fr > 1)) -- excecao de interface
12   InvalidCoalFeederRate.post:
13        (result.oclIsTypeOf(ftBoilerSystem.InvalidCoalFeederRate) = true)
14
15   InvalidAirFlowRate.condition:
16     (let message: OclMessage = PIDController^controlInputA(?: double) in
17                          --chamada de método (o nome do atributo pode ser omitido)
18       message.hasReturned()
19       and
20       ((message.result() < 0) or (message.result() > 0.1)))
21   InvalidAirFlowRate.post:
22     (result.oclIsTypeOf(ftBoilerSystem.InvalidAirFlowRate) = true)
23
```

Figura 5.20: Especificação contratual da interface `IAirFlowController`.

texto da invariante (linha 2), e o método `setCoalFeedRate()`, como contexto das pré e pós-condições (linhas 5 a 22).

A invariante e a pré-condição são registradas como expressões booleanas após as marcas `inv:` e `pre:`. No exemplo (linhas 2 e 5), como a interface não possui invariante nem o método possui pré-condição, essas restrições são representadas pela expressão *true*, que significa que qualquer estado é válido.

As pós-condições, como as pré, também são estabelecidas no contexto de métodos em OCL, utilizando a marca `post:`. O problema é que, para implementações que seguem a programação defensiva, ou mesmo utilizam mecanismos de tolerância a falhas, o método pode retornar, além de seu resultado normal, várias exceções diferentes, relativas a entradas inválidas, retornos indevidos de interfaces requeridas ou mesmo exceções internas. Como a pós-condição deve verificar todas as possibilidades de resultados frente a cada condição de disparo, a expressão pode ser tornar extremamente complexa, aumentando a probabilidade de introdução de falhas na especificação.

A OCL não apresenta nenhuma diretriz para diminuir essa complexidade. Por isso, para melhor organização das pós-condições frente a uma grande quantidade de saídas apresentadas por um método, foi adotada uma combinação das propostas de extensão da

linguagem OCL de [22], [33] e [68]:

- As pós-condições são classificadas de acordo com cada saída prevista. A pós-condição referente à saída normal é marcada como `norm.post` (linha 7), enquanto as relativas a retornos excepcionais são marcadas com o "nome da exceção correspondente"+ ".post"(linhas 10 e 15).

- Cada pós-condição foi dividida em condição de disparo (`condition`) e resultado esperado (`post`). Ambos fazem parte da expressão completa da pós-condição, porém a divisão facilita a interpretação da especificação por ferramentas. Por exemplo, na saída equivalente ao lançamento da exceção `InvalidCoalFeederRate` (Figura 5.20, linha 10), a condição de disparo é uma entrada inválida (linha 11) e o resultado esperado é o lançamento da exceção `InvalidCoalFeederRate`, contida no pacote `ftBoilerSystem` (linha 13).

A abordagem de Chessman e Daniels [22], que separa as pré e pós-condições em pares `pre/post`, não foi totalmente seguida por divergir da teoria de "*Design-by-contract*" proposta por Meyer [56], que estabelece que pré-condições são relativas a um método, e não a uma saída correspondente. Mas, como a proposta de Chessman e Daniels favorece a legibilidade da especificação tanto para pessoas quanto para ferramentas, ela foi adotada em um formato semelhante.

Algumas das construções OCL foram utilizadas na especificação apresentada. A variável `result` (linhas 13 e 22) é definida em OCL para capturar o retorno de um método. A verificação dos tipos dos objetos contidos em uma variável é realizada com a propriedade `oclIsTypeOf()`, que retorna *true* caso o objeto seja do mesmo tipo daquele passado como parâmetro.

Os retornos das interfaces requeridas são especificados com a expressão `OCLMessage` (linha 16), possibilitando a verificação das pós-condições relativas à propagação de exceções, por exemplo. A expressão `let..in` (linha 16) permite a criação de uma variável, neste caso, `message`, do tipo `OCLMessage`, sendo atribuída com um objeto representando a mensagem de retorno do método `PIDController.controlInputA()`. Como condição de disparo da exceção, são verificados o recebimento da mensagem (linha 18) e se seu valor está de acordo com o esperado, neste caso, o lançamento da exceção `InvalidAirFlowRate`, do pacote `ftBoilerSystem` (linha 20).

O objetivo da criação deste arquivo é a geração do código executável das assertivas a ser embutido no código do componente sob teste, a partir de uma especificação cuidadosamente elaborada já durante o projeto do componente. As assertivas são geradas como bibliotecas do componente `Tester` (Seção 4.4.3), e os passos para esta geração são apresentados na próxima Seção.

5.4.3 Geração do código executável

A partir da especificação do contrato da interface em OCL, é gerado o código executável das assertivas como bibliotecas do componente `Tester`, seguindo a estrutura apresentada na Seção 4.4.3, Figuras 4.10 e 4.11, são gerados aspectos relativos a cada interface, para a verificação da invariante da interface e das pré e pós-condições de cada um dos métodos. As especificações são obtidas a partir da interface `ITesting`, com o método `getAssertions()`.

A geração pode ser dividida em duas partes, descritas nas subseções a seguir: a geração da estrutura dos aspectos e a tradução das expressões OCL para a linguagem de programação utilizada, neste caso, Java.

Estrutura dos Aspectos

O código para verificação das assertivas é separado em diferentes aspectos, conforme descritos na Seção 4.4.1, divididos de acordo com o tipo das assertivas. A estrutura dos aspectos é similar para invariantes, pré e pós-condições: são compostos por um conjunto de junção, que interceptam os métodos a serem verificados, e adendos, que contém o código executável da assertiva.

Para verificação da invariante, todos os métodos da interface são interceptados, e a expressão é verificada antes e depois da execução de cada um. Assim, caso a invariante não seja *true*, deverá ser construído um aspecto contendo:

1. Um conjunto de junção que intercepte a execução de todos os métodos públicos da interface (comando `execution + public * "nome da interface".*(..)`).

2. Um adendo anterior, para a verificação da invariante antes da execução dos métodos.

3. Um adendo posterior, para a verificação da invariante após da execução dos métodos. Em ambos os adendos, caso seja detectada a violação da invariante, esta deve ser registrada no *log* com o método `LogAssert.writeInvViolation()`.

A utilização do adendo de contorno (Seção 4.4.1)não é possível para a verificação das invariantes, pois, apesar da mesma condição ser verificada antes e depois do método, o valor das variáveis, isto é, o contexto de execução é modificado entre as verificações. Assim, a utilização do adendo de contorno impede a visualização das mudanças, já que o contexto é capturado apenas uma vez na interceptação.

Para verificação de pré-condições, o método relativo à pré-condição é interceptado, sendo a expressão verificada antes de sua execução. A especificação contratual é examinada e, para cada método que possua `pre:` diferente de *true*, é criado um aspecto para verificação de sua pré-condição, contendo:

1. Um conjunto de junção que intercepte a execução do método ao qual a pré-condição se refere (comando `execution` + assinatura do método).

2. Um adendo anterior para a verificação da pré-condição. Caso seja constatada a violação, esta deve ser registrada com o método `LogAssert.writePreViolation()`.

Assim como para a pré-condição, para a verificação da pós-condição cada um dos métodos é interceptado separadamente. As expressões relativas a todas as possíveis saídas são verificadas após a execução do método, sendo seu término normal ou excepcional. Assim, a especificação é percorrida e, caso o método possua pelo menos um par `condition/post`, é criado um aspecto para a verificação de suas pós-condições contendo:

1. Um conjunto de junção que intercepte a execução do método ao qual a pós-condição se refere (comando `execution` + assinatura).

2. Um adendo posterior de retorno normal (`after () returning()`), para a verificação da pós-condição caso o método termine normalmente.

3. Um adendo posterior de retorno excepcional (`after () throwing()`), para a verificação da pós-condição caso o método termine de forma excepcional.

A estrutura interna dos adendos para verificação das pós-condições é mais complexa que para as invariantes e pré-condições, devido a existência dos pares `condition/post`. Cada um dos adendos faz a verificação das pós-condições relativas a todas as saídas possíveis, como exemplificado nas Figuras 4.11 e 5.20. Para cada adendo, porém, os pares `condition/post` referentes a saídas normais e excepcionais são tratados de forma diferente.

No adendo para retorno excepcional, as saídas relativas a retornos excepcionais são verificadas de forma completa: as expressões contidas em `condition:` e `post:` são unificadas no formato *condition* **and not***(post)*, que pode ser entendido como "caso a condição de disparo seja satisfeita e a pós-condição não seja, ocorreu uma violação". Isso pode ser observado na verificação da exceção `InvalidCoalFeederRate` (Figura 4.11, linha 37 e 38 e Figura 5.20, linhas 10 a 13).

Para a saída normal, porém, apenas a expressão contida em `condition` é verificada, já que o lançamento da exceção capturado pelo adendo posterior excepcional representa a violação da expressão contida em `post`. Isto é, se a condição de disparo for satisfeita (configurando uma situação de normalidade), a exceção capturada pelo adendo para retorno excepcional não deveria ter sido lançada, caracterizando a violação.

Para o adendo para retorno normal, a situação se inverte: a pós-condição referente à saída normal é verificada no formato *condition* **and not***(post)*, enquanto para as saídas excepcionais apenas a expressão contida em `condition` é verificada.

As Figuras 4.11, linha 25 e 5.20, linhas 10 a 13 trazem a verificação da exceção `InvalidCoalFeederRate` como exemplo. Todas as violações são registradas através do método `LogAssert.writePostViolation()`.

Tradução das Expressões OCL

Após a criação da estrutura dos aspectos, é iniciada a tradução das expressões OCL para Java. Por exigir um estudo mais aprofundado, uma tradução genérica o suficiente para expressões OCL quaisquer está além do escopo deste trabalho. Assim, serão apresentadas diretrizes apenas sobre as expressões utilizadas nos exemplos.

Foram utilizadas três categorias de expressões OCL: relativas a parâmetros de entrada, a tipos de retorno, e a interação com as interfaces requeridas. Exemplos podem ser encontrados na Figura 5.20, linhas 11, 13 e 16 a 20, respectivamente.

As *expressões relativas a parâmetros de entrada* contêm restrições relativas aos argumentos de entrada do método. São as mais simples de serem traduzidas: basta substituir os operadores na linguagem OCL (*and* e *or*, por exemplo), pelos correspondentes na linguagem Java. A expressão OCL `((C_fr < 0) or (C_fr > 1))` foi transformada em `((C_fr < 0) || (C_fr > 1))`.

As *expressões relativas a tipos de retorno* são referentes a verificação do tipo do objeto retornado por um método. Essas expressões são muito utilizadas nas pós-condições excepcionais, para verificação do tipo da exceção lançada. Geralmente são relacionadas à variável para captura do retorno do método `result`, sendo o tipo desta variável verificado com a utilização das propriedades `oclIsTypeOf()` e `oclIsKindOf()`. A propriedade `oclIsTypeOf()` verifica se o tipo da variável é exatamente o mesmo do nome passado como parâmetro. A expressão OCL

`result.oclIsTypeOf(ftBoilerSystem.InvalidCoalFeederRate)`,

por exemplo, é traduzida para

`e.getClass().getName().equals("ftBoilerSystem.InvalidCoalFeederRate")`,

que compara o nome da classe da exceção lançada "e"ao passado como parâmetro.

Já a propriedade `oclIsKindOf()` verifica se o tipo da variável é o mesmo ou uma subclasse do tipo passado como parâmetro. Neste caso, a tradução utiliza o comando Java `instanceof`: a expressão OCL

`result.oclIsKindOf(DeclaredException)`,

por exemplo, é traduzida para

`(e instanceof DeclaredException)`,

que compara se a exceção lançada "e"é instância da classe "`DeclaredException`", mesmo que indiretamente.

A tradução das *expressões relativas à interação com as interfaces requeridas* é a mais complexa das três. São expressões que capturam o retorno de um método da interface

requerida para construção da verificação. Um exemplo pode ser observado nas Figuras 5.20, linhas 16 a 20 e 4.11, linhas 4 a 15 e 45 a 50. A expressão em OCL contida em `InvalidAirFlowRate.condition` significa que a variável `message` foi atribuída com o valor de retorno do método `PIDController.controlInputA`. Esta variável é utilizada para verificação se o método retornou algum valor, e se o valor retornado (`message.result()`) está entre 0 e 0,1.

Para a verificação deste tipo de expressão, o método da interface requerida é interceptado, para a captura de seu valor de retorno. Esse valor é armazenado em uma variável temporária, interna ao aspecto, e utilizado durante a verificação da pós-condição. Sendo assim, a tradução dessas expressões para Java possui as seguintes etapas:

1. *Obtenção do valor de retorno do método*: para a obtenção do valor, a chamada do método requerido (obtida na especificação junto à criação da variável `message`) é interceptada por um conjunto de junção, e seu valor de retorno é capturado por um adendo posterior. O adendo utilizado deve ser normal ou excepcional, dependendo do valor de retorno esperado.

   É importante que a chamada interceptada seja restrita ao método para o qual a pós-condição está sendo verificada. Por exemplo, na Figura 4.11, o método `PIDController.controlInputA` é interceptado pelo conjunto de junção `req1` (linhas 8 a 10), que limita seu alcance ao contexto capturado pelo conjunto de junção `assertion`. Foi utilizado um adendo posterior normal (linhas 13 a 15) pois o valor esperado era o retorno normal.

2. *Armazenamento*: é criado um atributo privado no aspecto, para armazenamento do valor retornado pela interface requerida. A atribuição é realizada no código do adendo posterior criado no passo anterior, como observado na linha 14 da Figura 4.11.

3. *Verificação*: após a captura do valor do método (linhas 16 a 18 da especificação), a condição relativa a este valor deve ser verificada (linha 20). Essa condição é convertida na condição de disparo da pós-condição, e verificada nos adendos para verificação da pós-condição. A tradução é basicamente a troca dos nomes das variáveis de `message.result()` para o nome do atributo criado no passo 2. No exemplo, a expressão

   ```
   ((message.result() < 0) or (message.result() > 0.1)))
   ```

   é traduzida para `((req1 < 0) || (req > 0.1)))`.

Apesar da explicação curta, as diretrizes de tradução apresentadas foram suficientes para os estudos de caso executados, possibilitando a rápida criação dos componentes `Tracker` e `Tester`, e a utilização das assertivas como oráculo dos testes.

5.5 Mecanismos de Monitoração

Após a execução dos testes, é iniciada a fase de correções, para correção dos defeitos encontrados. Para facilitar a localização das falhas, além da verificação de assertivas em tempo de execução, a arquitetura do componente testável também contém recursos para monitoração da execução do componente, indicando o fluxo executado durante os testes. O código para monitoração é embutido no CST pelo componente `Tracker`, cujas bibliotecas também são construídas com a utilização da programação orientada a aspectos, conforme descrito na Seção 4.4.3.

As bibliotecas do componente `Tracker` podem ser geradas a partir das interfaces providas e requeridas do componente. A geração é simples, já que as bibliotecas possuem uma estrutura fixa, apresentada no Capítulo 4 (Figuras 4.8 e 4.9). Assim como as bibliotecas do componente `Tester`, os aspectos de monitoração são formados por um conjunto de junção, que intercepta o código a ser monitorado, e adendos, que registram no *log* os valores durante a execução.

Para a monitoração operacional, que registra a execução dos métodos do componente, são criados aspectos para a interceptação dos métodos de cada uma das interfaces (providas e requeridas), a partir das assinaturas. Um exemplo da tradução da assinatura de um método da interface `PIDController`, requerida pela interface `IAirFlowController`, é apresentado na Figura 5.21. É criado um aspecto para cada uma das interfaces contendo, para cada um dos métodos:

1. Um conjunto de junção: se o método for de uma interface provida, o conjunto de junção deve interceptar sua execução (comando `execution` + assinatura); se for da interface requerida, o conjunto de junção deve interceptar a chamada do método (comando `call` + assinatura) durante a execução dos métodos públicos do componente (comando `cflow` + public * Interface.*(..)). Em ambos os casos, o conjunto de junção deve ainda coletar o valor dos argumentos do método (comando `args`).

2. Um adendo anterior relativo ao conjunto de junção, registrando a chamada do método e o valor dos seus argumentos com `LogTrace.writeOperationalTraceEntry()`.

3. Um adendo posterior para captura de retorno normal (*returning()*), registrando o término e o valor de retorno (se houver) com `LogTrace.writeOperationalTraceReturn()`.

4. Um adendo posterior para captura de retorno excepcional (*throwing()*), registrando a exceção lançada com `LogTrace.writeOperationalTraceReturn()`.

Assinatura:
`public double PIDController.controlInputA(double value)`

Conjunto de junção e adendos gerados (código interno dos adendos foi omitido):

```
1 private pointcut controlInputAMethod(double value):
2   call(public double PIDController.controlInputA(double))
3   && args(value)
4   && cflow(public * IAirFlowController.*(..));
5 before (double value): controlInputAMethod(value){...}
6 after (double value) returning(): controlInputAMethod(value){...}
7 after (double value) throwing (Exception e): controlInputAMethod(value){...}
```

Figura 5.21: Exemplo de geração do código de monitoração a partir da assinatura do método.

A geração dos aspectos relativos à monitoração de estados, que registra o acesso a atributos e propriedades, é bem similar à operacional. Aspectos deste tipo são mais raros, dadas as propriedades do desenvolvimento baseado em componentes que recomendam apenas a existência de métodos nas interfaces. Porém, caso existam atributos públicos e/ou propriedades, elas podem ser monitoradas com a criação deste tipo de aspecto, como ilustra o exemplo contido na Figura 5.22 da geração de código de monitoração a partir de um atributo público. É criado um aspecto por interface, contendo, para cada atributo e/ou propriedade públicos:

1. Um conjunto de junção que intercepte a leitura do atributo: comando `get` + "nome do atributo".

2. Um adendo anterior para o conjunto de junção de leitura, registrando o valor lido com o método `LogTrace.writeStateTraceRead()`.

3. Um conjunto de junção que intercepte a escrita do atributo: comando `set` + "nome do atributo".

4. Um adendo anterior para o conjunto de junção de escrita, registrando o valor antes da escrita com `LogTrace.writeStateTraceWrite()`.

5. Um adendo posterior para o conjunto de junção de escrita, registrando o valor depois da escrita com `LogTrace.writeStateTraceWrite()`.

Os aspectos de monitoramento operacional e de estados devem ser armazenados nos pacotes `operational` e `state`, respectivamente. Através da interface `ITracking`, o usuário pode escolher quais interfaces terão os métodos monitorados, utilizando o método `ITracking.operationalTrace(String)` para cada interface escolhida; e pode escolher

Atributo:
```
public double C_fr;
```

Conjunto de junção e adendos gerados (código dos adendos foi omitido):

```
1 private pointcut c_frAttributeGet(double C_fr): get(public double C_fr);
2 before(): c_frAttributeGet(value){...}
3 private pointcut c_frAttributeSet(double C_fr): set(public double C_fr);
4 before(): c_frAttributeSet(value){...}
5 after(): c_frAttributeSet(value){...}
```

Figura 5.22: Exemplo de geração do código de monitoração a partir da assinatura do atributo.

quais interfaces terão seus atributos/propriedades monitorados, invocando o método `ITracking.stateTrace(String)` para cada interface.

5.6 Elaboração do Driver

Conforme ilustrado na Figura 5.1, o *driver* coordena toda a estrutura de testes. Além da execução e avaliação dos resultados, é responsável por preparar o ambiente, incluindo a instrumentação do componente e a atribuição dos valores aos *stubs*.

Sendo assim, o *driver* é estruturado em quatro partes:

1. *Instrumentação do componente*: são inseridas as assertivas e os mecanismos de monitoração nas interfaces selecionadas. Geralmente é realizada apenas uma vez para cada conjunto de testes, já que todos os casos de teste obtidos a partir do diagrama de atividades são relativos à mesma interface.

2. *Instanciação dos componentes*: são instanciados o componente sob teste (já instrumentado) e os *stubs*, e realizadas as conexões necessárias entre as interfaces requeridas e providas. Para que os testes sempre comecem no estado inicial do componente, esta etapa deve ser repetida a cada novo caso de teste (neste caso, não é necessária a criação do método "`setEmpty()`"nos *stubs* - Seção 5.2.1).

3. *Preparação dos stubs*: durante esta etapa, já utilizando as informações geradas a partir do diagrama de detalhamento, o *driver* atribui aos *stubs* os valores a serem retornados durante a execução do caso de teste. Essa atribuição é realizada através dos métodos "`setMethod`"providos pelos *stubs*.

   A preparação dos *stubs* pode ocorrer antes da chamada de cada método que se relacione com as interfaces requeridas, conforme descrito na Seção 5.2.2, ou antes da

execução de cada caso de teste, concentrando todas as atribuições à *stubs* necessárias antes da chamada dos métodos da interface provida do CST.

4. *Execução dos testes*: após a preparação do ambiente, inicia-se a execução dos testes, conforme descrito na Seção 5.1.2. Os métodos participantes do caso de teste são chamados, e ao final o resultado obtido é comparado ao resultado esperado.

No estudo de caso realizado, a implementação do *driver* foi dividida em dois arquivos: o primeiro é responsável pela instrumentação do componente, e o segundo realiza as três últimas etapas. O segundo *driver* é carregado pelo primeiro ao final de sua execução, juntamente com o componente sob teste já instrumentado.

A Figura 5.23 mostra a estrutura do primeiro *driver*. É composto por quatro métodos executados seqüencialmente, respectivamente responsáveis pela inclusão das assertivas, inclusão dos mecanismos de monitoração, combinação dos aspectos selecionados e do CST, e execução do segundo *driver*.

O método para inclusão das assertivas é exibido em detalhes na Figura 5.23, contendo o código responsável pela inclusão das pré e pós-condições na interface `IAirFlowController`. Para cada interface a ser instrumentada, são invocados os métodos da interface `ITesting` relativos ao tipo de assertiva escolhida, e posteriormente o método `finalize()`, indicando que toda a instrumentação já foi sinalizada. O método para inclusão dos mecanismos de monitoração tem a mesma estrutura.

Para a construção do segundo *driver*, foi utilizada a estrutura `TestCase` fornecida pelo *framework* JUnit, conforme descrito na Seção 5.1.2. Além dos métodos "test", que concentram as etapas de preparação dos *stubs* e execução dos testes, são utilizados os métodos:

- Método `setUp()`: repetido antes da execução de cada caso de teste, concentrando a etapa de instanciação dos componentes.
- Método `tearDown()`: repetido após a execução de cada caso de teste, deve ser utilizado para liberação de recursos utilizados durante os testes.

5.7 Considerações Finais

Neste capítulo foram descritos todos os aspectos para implementação da estrutura necessária para o teste de componentes isolados. Foram apresentadas diretrizes para a criação dos *drivers*, *stubs* e bibliotecas de aspectos para verificação de assertivas e monitoração do componente sob teste (CST). Todos estes artefatos são obtidos a partir das especificações do CST contidas na arquitetura do componente testável (comportamental e contratual), cujos formatos também foram apresentados.

```
1  public class Instrument {
2
3   //inclusão das assertivas selecionadas
4   private void includeAssertion() {
5    ITesting t = new Testing();
6    //um bloco try/catch para cada método, que capture a exceção que sinaliza que o
7    //arquivo contendo o aspecto para instrumentação da interface não foi encontrado
8    try {
9     //verificação das pré-condições
10          t.insertPreCondition("AirFlowController");
11   }
12   catch (FileNotFoundException e) {
13    System.out.println(e.getMessage());
14   }
15   try {
16    //verificação das pós-condições
17          t.insertPostCondition("AirFlowController");
18   }
19   catch (FileNotFoundException e) {
20    System.out.println(e.getMessage());
21   }
22   //indica que todos os aspectos de verificação de contrato já foram escolhidos
23   t.finalize();
24  }
25
26  //inclusão dos mecanismos de monitoração selecionados
27  private void includeTracking() {..}
28
29  //combinação dos aspectos selecionados ao componente sob teste
30  private void weaving() {..}
31
32  //execução do segundo driver
33  private void execution() {..}
34
35 }
```

Figura 5.23: Estrutura do primeiro *driver*, responsável pela instrumentação do CST.

Em todos os passos é dada uma atenção especial ao comportamento excepcional do componente, que constitui grande parte do código de um componente tolerante a falhas. Na especificação contratual do componente, foi definido um formato para melhorar a legibilidade de pós-condições relativas ao lançamento de exceções. No diagrama comportamental, as situações excepcionais são modeladas nas condições de guarda das arestas, facilitando a geração de dados de teste que exercitem tais situações.

Todas as etapas para criação dos artefatos de teste podem ser automatizadas, seguindo as diretrizes apresentadas. A criação de ferramentas para a automação é essencial para o sucesso do processo de testes proposto, apresentado no próximo capítulo, pois possibilita a execução de um número muito maior de casos de teste e a conseqüente descoberta de um maior número de falhas. Essas ferramentas serão implementadas em trabalhos futuros.

Os passos apresentados neste capítulo compõem as atividades do método de testes, apresentado no próximo capítulo.

Capítulo 6

Um Método de Testes utilizando o Componente Testável

Este capítulo apresenta o método de testes baseado na melhoria da testabilidade dos componentes tolerantes a falhas proposto neste trabalho. Voltado para o teste funcionais de componentes isolados, o método apresenta diretrizes para a construção do componente testável pelo desenvolvedor do componente, auxiliando tanto o desenvolvedor na validação do componente quanto o usuário, na validação do componente em seu ambiente.

A principal característica do método é a preocupação tanto com o comportamento normal quanto excepcional, já que o bom funcionamento de ambos é imprescindível para garantir a confiabilidade de componentes tolerantes a falhas. Apesar do enfoque neste tipo de componente, o método é genérico o suficiente para ser utilizado em componentes quaisquer, caso o custo da construção dos artefatos de teste se justifique.

Outra característica importante é que o método proposto foi elaborado para ser executado em paralelo com um processo de desenvolvimento. O ideal para um método de testes é que este tenha início juntamente com o método de desenvolvimento, possibilitanto a criação de sistemas com melhor testabilidade e de casos de teste em maior número, que validem as reais necessidades do sistema.

Neste trabalho, o processo de desenvolvimento baseado em componentes escolhido foi o *UMLComponents*, mais especificamente o método MDCE+, uma adaptação do *UMLComponents* para construção de sistemas confiáveis. Ambos foram apresentados no Capítulo 2. O método para melhoria da testabilidade é executado paralelamente ao desenvolvimento, baseando-se na documentação produzida.

Na Seção 6.1 são apresentados os passos para os testes de componentes utilizando a arquitetura do componente testável, construída paralelamente ao desenvolvimento do componente. A Seção 6.2 traz um sumário do capítulo, incluindo uma tabela com um resumo do método proposto.

6.1 Método de Testes

Esta Seção apresenta a visão dos testes durante o desenvolvimento: a equipe executa o método de testes paralelamente ao método de desenvolvimento, neste caso, o MDCE+. As principais preocupações na elaboração do método de testes foram a minimização do custo para criação dos artefatos de teste, com o aproveitamento da documentação produzida durante o desenvolvimento, e a independência da equipe de testes em relação à equipe de desenvolvimento, com a produção de especificações completas e sem ambigüidades.

Apesar de não ser um processo completo de testes por contemplar apenas a fase de testes de componentes, as fases clássicas de um processo de testes (preparação e realização, descritas na Seção 2.2.3) foram consideradas, e suas atividades foram dispostas ao longo do ciclo. Os documentos produzidos também são baseados na norma IEEE 829 (Seção 2.2.3).

A Figura 6.1 apresenta as fases do método de testes, executadas paralelamente às fases do método de desenvolvimento. As próximas subseções descrevem as atividades de cada uma das fases do método de testes, indicando quais os artefatos necessários para o início da fase e seus produtos. Ao final da Seção é apresentado o resumo do método na Tabela 6.1.

As fases de testes de unidade, integração e sistema serão tratadas em trabalhos futuros.

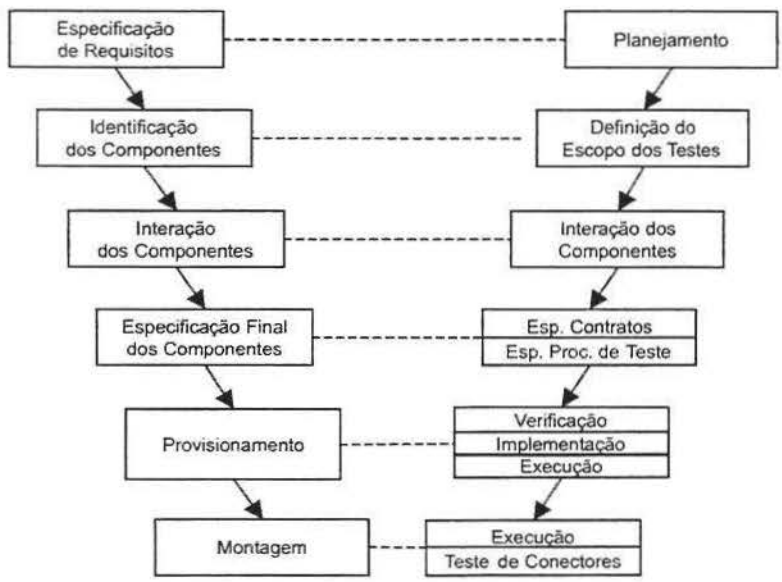

Figura 6.1: Paralelismo entre as fases dos métodos de desenvolvimento e testes.

6.1.1 Planejamento

Entradas: Plano de Desenvolvimento do Projeto, Casos de Uso.

O planejamento dos testes é realizado com base no planejamento realizado para o projeto e no seu escopo, obtido a partir dos casos de uso. Nesta fase é construído o plano de testes de acordo com a proposta da norma IEEE 829. São descritos aspectos como os recursos necessários, o cronograma de atividades, as tarefas a serem realizadas e os riscos aplicáveis aos testes.

Outro item do plano é a abrangência dos testes. Para os testes de componentes, esta parte do plano de testes é preenchida na próxima fase, já que nesta fase os componentes da aplicação ainda não foram definidos.

Saídas: Plano de Testes.

6.1.2 Definição do Escopo dos Testes

Entradas: Arquitetura inicial do sistema (componentes de negócio e de sistema).

Nesta fase são especificados quais componentes devem se tornar componentes testáveis, registrados no item "Abrangência dos Testes"no documento Plano de Testes. Como geralmente o número de componentes é alto, não é possível a criação dos artefatos para todos e, por isso, alguns são testados apenas durante os testes de integração e/ou sistema.

A partir da arquitetura inicial do sistema, são escolhidos quais componentes se tornarão testáveis. Recomenda-se que sejam escolhidos de acordo com os critérios:

- Criticidade: componentes críticos devem ter seu funcionamento garantido, e por isso devem ser priorizados para a elaboração e execução dos testes.

- Reusabilidade: como componentes reutilizáveis são testados a cada integração em um novo sistema, o custo para a automação dos testes é compensado pelo número de repetições da execução dos testes. Além disso, se o componente for reutilizado por terceiros, os artefatos contribuem para melhorar o entendimento acerca do componente, durante a reutilização.

- Volatilidade de requisitos: componentes com requisitos com alta probabilidade de serem modificados ao longo de seu ciclo de vida, exigindo a freqüente realização de testes de regressão, justificando a construção dos artefatos. É prevista a construção de ferramentas para auxiliar a atualização dos artefatos de testes.

Outra consideração a ser feita é relativa ao isolamento do componente durante os testes. Devem ser analisadas suas interfaces requeridas, e para quais dessas interfaces é vantajosa a criação de *stubs*. Caso o componente requerido seja muito simples e suas

saídas sejam facilmente controláveis, não é justificado o custo para a criação de um *stub*: é mais vantajoso que o componente requerido seja integrado ao componente sob teste durante a realização dos testes.

É importante ressaltar que as decisões tomadas durante esta fase ainda podem ser ajustadas, especialmente em relação à decisões relativas à integração de componentes, pois o comportamento detalhado de muitos componentes ainda não é conhecido nesta etapa.

Saídas: Lista de componentes a serem testados isoladamente

6.1.3 Interação dos Componentes

Entradas: Arquitetura inicial do sistema, interfaces dos componentes de sistema, e cenários dos casos de uso.

Esta fase engloba a atividade do MDCE+ de geração dos diagramas de detalhamento, das interfaces requeridas (Seção 5.1.1) para cada um dos métodos das interfaces providas. Os diagramas são construídos pela equipe de desenvolvimento para a descoberta das interfaces requeridas, são completados pelos testadores com as condições de lançamento das exceções de interface e utilizados para a geração dos dados a serem retornados pelos *stubs*. A confecção do diagrama pela equipe de desenvolvimento traz maior fidelidade ao modelo em relação aos requisitos e à futura implementação.

Os diagramas produzidos são parte do documento "Especificação do Projeto de Teste", que começa a ser produzido nesta fase. O documento contém as funcionalidades e características do sistema a serem testadas, e a descrição dos casos e procedimentos de teste de componentes.

Para os testes de componentes, os casos e procedimentos de teste são descritos através das especificações comportamentais (diagramas principal e de detalhamento), a partir dos quais os casos de teste serão gerados, e das especificações contratuais, base para as assertivas que funcionam como oráculos nos testes. Nesta fase, os diagramas de detalhamento são incluídos no documento.

Outra atividade desta fase é a revisão da abrangência dos testes (Plano de Testes), já que agora o comportamento das interfaces requeridas é conhecido. Devem ser revisados os componentes a serem substituídos por *stubs* de acordo com sua complexidade e controlabilidade.

Saídas: Documento "Especificação do Projeto de Teste", contendo as especificações comportamentais incompletas (apenas com os diagramas de detalhamento).

6.1.4 Especificação dos Contratos dos Componentes

Entradas: Casos de Uso, interfaces dos componentes a serem testados

Durante esta fase, as especificações contratuais (Seção 5.4.2) são construídas para os componentes selecionados e integradas à arquitetura do componente testável. A atividade de formalização das assertivas, opcional no *UMLComponents* e MDCE+, torna-se obrigatória para a construção do componente testável, sendo registrada no formato da especificação contratual.

A formalização pode ser realizada pela equipe de testes, com base nos contratos dos casos de uso definidos na especificação de requisitos. Além de incluída no componente testável, é também integrada ao documento "Especificação de Projeto de Testes".

Saídas: Documento "Especificação do Projeto de Teste", acrescido das especificações contratuais.

6.1.5 Especificação dos Procedimentos de Teste

Entradas: Casos de Uso, interfaces dos componentes a serem testados

Nesta fase as especificações comportamentais são concluídas, e completam o documento "Especificação do Projeto de Teste". As especificações comportamentais são finalizadas com a construção dos diagramas das interfaces providas (Seção 5.1.1), que podem ser desenvolvidos pela equipe de testes com base na descrição dos casos de uso. Devem ser considerados tanto cenários normais quanto excepcionais, para definição do fluxo de execução e das exceções de interface.

Saídas: Documento "Especificação do Projeto de Teste", acrescido das especificações comportamentais completas (diagramas de atividades principal e de detalhamento).

6.1.6 Verificação dos Modelos

Entradas: Especificações comportamentais e contratuais ("Especificação do Projeto de Teste").

Apesar do desenvolvimento e dos testes seguirem um processo paralelo, eventualmente surgem inconsistências entre as especificações e o código implementado, especialmente devido a mudanças realizadas apenas no código. Assim, para que os testes gerados sejam fiéis aos reais requisitos do sistema, é importante que, ao final da implementação de cada

componente, a equipe de testes verifique junto à equipe de desenvolvimento se os modelos realmente refletem o que foi implementado. Esta verificação evita o custo de posteriores correções dos casos de teste.

Saídas: Especificações comportamentais e contratuais em concordância com os atuais requisitos do sistema.

6.1.7 Implementação

Entradas: Interfaces providas e requeridas dos componentes, especificações contratuais e comportamentais.

A geração dos componentes testáveis e dos casos de teste acontece logo após a revisão dos documentos, com a execução dos passos apresentados no Capítulo 5. São geradas as bibliotecas dos componentes `Tracker` e `Tester` e os casos de teste a partir das especificações comportamentais. Para agilizar a execução dos testes, é recomendado que os artefatos sejam gerados separadamente para cada um dos componentes à medida que eles são implementados, agilizando a realização dos testes nos componentes já finalizados.

Os arquivos XML gerados a partir das especificações comportamentais compõem o documento "Especificação dos Casos de Teste", que traz as definições dos casos de teste, seus dados de entrada, resultados esperados, ações e condições gerais para sua execução. O documento contém as *TestSuites* geradas em XML, mais os valores incluídos manualmente no arquivo XML. Cada *TestSuite* é acompanhada de uma breve descrição sobre as condições gerais para sua execução, inclusive quais das assertivas e mecanismos de monitoração devem ser ativados.

Como os passos de execução dos casos de teste também são especificados nos arquivos XML, o conteúdo do documento "Especificação de Procedimentos de Teste"é integrado à "Especificação dos Casos de Teste".

Após a criação do XML devem ser gerados os casos de teste na linguagem do sistema em questão, também com a utilização de ferramentas de geração automática, agilizando a construção.

Saídas: Componente testável e casos de teste executáveis.

6.1.8 Execução dos Testes

Entradas: Componente testável e casos de teste executáveis.

A fase de execução dos testes pode ser executada tanto em paralelo com a fase de provisionamento quanto com a fase de montagem, dependendo dos componente escolhidos para serem testados. Caso os componentes tenham sido implementados totalmente durante o provisionamento, seus testes são executados ainda durante esta fase. Caso tenha sido decidido pela integração de alguns componentes para evitar a construção de *stubs*, a execução dos testes é realizada apenas durante a fase de montagem, na qual são construídos os conectores que integram os componentes.

Durante a execução dos testes são produzidos alguns relatórios, como indicado pela Norma IEEE 829 (Seção 2.2.3): diário de teste, com os registros cronológicos da execução; relatório resumo dos testes, com a descrição resumida das atividades de teste e uma avaliação dos resultados; relatório de incidência de testes, registrando eventos ocorridos durante os testes que mereçam análise posterior; relatório de encaminhamento do item de teste, encaminhando as correções para as equipes responsáveis.

Para melhor gerenciamento dos testes e da manutenção corretiva, é recomendado que o relatório de encaminhamento do item de teste seja construído com a utilização de ferramentas de registros de defeitos, como o Bugzilla [60], facilitando a descrição dos defeitos encontrados e a distribuição das tarefas de manutenção corretiva.

Saídas: Diário de teste, relatório resumo dos testes, relatório de incidência de testes, relatório de encaminhamento do item de teste.

6.1.9 Teste de Conectores

Entradas: Especificação dos conectores do sistema.

Apesar de todos os componentes já estarem especificados e testados, a especificação dos conectores é finalizada apenas na fase de montagem. Sua construção é iniciada na fase de interação dos componentes, na qual a especificação do comportamento excepcional do componente tem início. A forma de adaptação entre as diferentes interfaces, porém, é especificada apenas durante a fase de montagem.

Assim, a preparação e realização dos testes dos conectores tem que ser postergada até esta fase. O ciclo do método de testes deve ser executado a partir das atividades da fase de identificação dos componentes, na qual são selecionados os conectores que serão testados individualmente. É recomendado que apenas os conectores que implementam comportamentos mais complexos sejam testados isoladamente, sendo os demais exercitados durante os testes de integração.

A transformação dos conectores em componentes testáveis é simplificada pela compatibilidade entre as interfaces do conector e dos componentes ligados a ele. A compatibi-

lidade permite que os artefatos de teste dos componentes sejam reutilizados no conector, eliminando o custo de sua criação. Os testes são executados da mesma maneira que nos componentes, conforme descrito na Seção anterior.

Saídas: Documentos de teste acrescidos das informações sobre os testes dos conectores, incluindo os documentos relativos à execução dos testes.

6.2 Considerações Finais

Neste capítulo foi apresentado o método de testes proposto. A Tabela 6.1 traz um resumo do método, organizando-o por fases, trazendo suas atividades, documentos produzidos e responsabilidades. Suas fases foram elaboradas para execução em paralelo com as fases do desenvolvimento, gerando testes mais fiéis aos requisitos do sistema.

A integração dos métodos de desenvolvimento e testes foi testada em um estudo de caso realizado em um ambiente real, descrito no próximo capítulo.

Fase	Atividades	Documento	Responsabilidade
Planejamento	Escrita do Plano de Testes	Plano de Testes	Testadores
Definição da Abrangência dos Testes	Escolha dos componentes que se tornarão testáveis	Plano de Testes e Especificação do Projeto de Teste	Testadores
Interação de Componentes	Criação dos diagramas das interfaces requeridas	Especificação do Projeto de Teste	Desenvolvedores
Especificação dos Contratos	Criação especificações contratuais	Especificação do Projeto de Teste	Testadores
Especificação dos Procedimentos de Teste	Criação dos diagramas das interfaces providas	Especificação do Projeto de Teste	Testadores
Verificação dos Modelos	Verificação das especificações contratuais e comportamentais frente aos atuais requisitos	Especificação do Projeto de Teste	Testadores
Implementação	Geração dos componentes testáveis (componentes Tester e Tracker)	Código Fonte	Testadores
	Geração dos casos de teste (XML)	Especificação dos Casos de Teste	Testadores
	Geração dos casos de teste (linguagem de programação)	Código Fonte	Testadores
Execução	Aplicação dos Testes	Diário de teste	Testadores
		Rel. Resumo dos Testes	
		Rel. de Incidência de Testes	
		Rel. de Encaminhamento do Item de Teste	
Teste de Conectores	Aplicação do ciclo aos conectores	Plano de Testes	Testadores
		Esp. do Projeto de Teste	
		Esp. dos Casos de Teste	
		Código Fonte	
		Diário de Testes	
		Rel. Resumo dos Testes	
		Rel. de Incidência de Testes	
		Rel. de Encaminhamento do Item de Teste	

Tabela 6.1: Fases de teste e suas atividades, ressaltando os responsáveis e os documentos produzidos.

Capítulo 7

Estudo de Caso

A validação do método de testes proposto, em conjunto com o método de desenvolvimento MDCE+, foi realizada através de um estudo de caso em um ambiente real. O ambiente escolhido foi a Autbank Projetos e Consultoria Ltda, uma empresa brasileira de médio porte, consultoria e desenvolvimento de software para o mercado financeiro. Em conjunto com a equipe da empresa, foi construído e testado um sistema financeiro de cadastro e controle de emissão de cheques e limite de crédito, no qual a confiança no funcionamento (*dependability*) é uma das principais características.

A execução do estudo de caso ocorreu em três etapas: (i) especificação do sistema; (ii) implementação do sistema, em paralelo à geração dos casos de teste; e (iii) execução dos testes e coleta de dados. Para facilitar a execução, apenas a fase de especificação foi realizada dentro da empresa, com a presença de duas analistas, uma delas com experiência em desenvolvimento baseado em componentes e grande domínio do negócio. Durante esta fase, as analistas foram apenas orientadas pelos autores.

Durante o estudo de caso foi possível exercitar cada uma das fases de desenvolvimento e testes em conjunto, o que contribuiu para a maior integração entre os métodos. Além disso, a visão da equipe da empresa favoreceu a agilidade do método.

A partir deste estudo de caso foi produzido um relatório técnico [65] que descreve detalhadamente todas as atividades e produtos. Neste capítulo são sucintamente descritos o sistema implementado (Seção 7.1), como foram realizados os testes (Seção 7.2), e os resultados dos testes juntamente com uma avaliação crítica (Seção 7.3). A Seção 7.4 traz um sumário do capítulo.

7.1 Descrição do Sistema

O sistema implementa 6 casos de uso principais, sendo 10 no total, como ilustra a Figura 7.1. A partir destes casos de uso foram levantados os cenários normais, alternativos

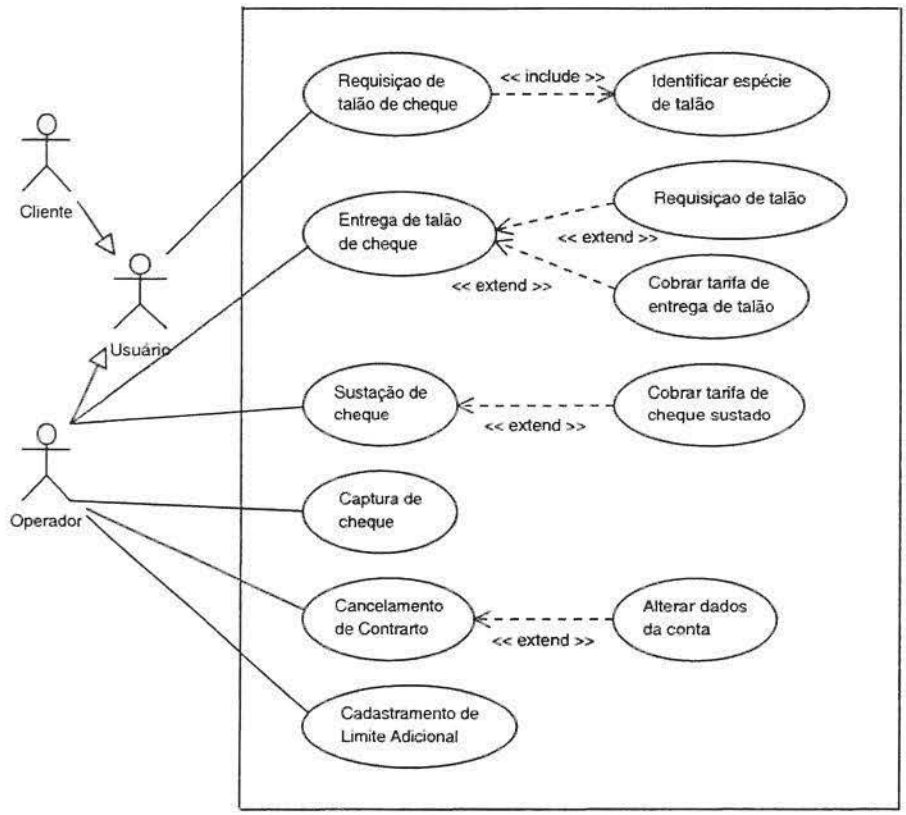

Figura 7.1: Diagrama de Casos de Uso

e excepcionais, além das assertivas e exceções (informalmente). As principais funcionalidades são:

1. **Requisição de talões de cheques.**: Solicitação de talões de cheque pelo cliente, diretamente em uma agência;

2. **Entrega de talões de cheques.**: Retirada de talões de cheque solicitados previamente;

3. **Sustação de cheques.**: Solicitação do cancelamento de um determinado número de cheques pelo cliente;

4. **Captura de cheque para compensação.**: Captura de um cheque durante um depósito, para sua posterior compensação.

5. **Cancelamento do contrato da conta.**: Perda do crédito da conta de um cliente, cancelando o contrato de crédito.

6. **Cadastramento de limite adicional.**: Aumento do limite de crédito do cliente.

A arquitetura adotada é ilustrada na Figura 7.2. Foi adotada uma arquitetura em camadas relaxada, que segue restrições impostas pelo ambiente da empresa, tais como componentes utilitários e a maneira de acesso aos dados.

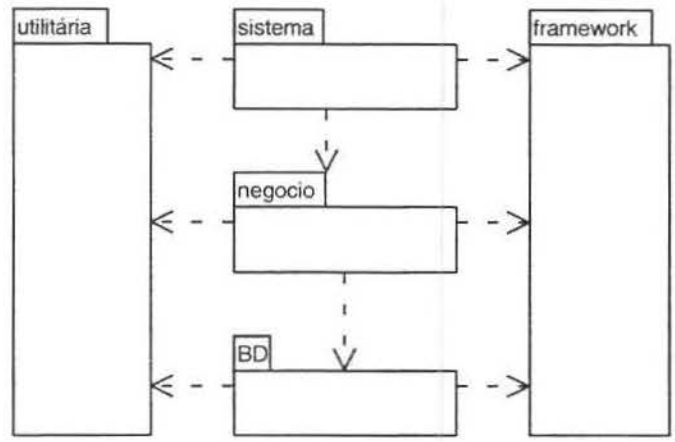

Figura 7.2: Arquitetura do Sistema Bancário

A camada `utilitária` agrupa componentes utilitários, cujos serviços são requeridos por componentes de diferentes camadas. A camada `sistema` contém os componentes que implementam as funcionalidades da aplicação especificamente, enquanto a camada `negócio` agrupa os componentes que gerenciam os dados necessários. A camada de banco de dados (`BD`) traz a implementação do acesso ao banco de dados escolhido, enquanto a `framework` agrupa as classes necessárias para a implementação de acordo com o modelo de componentes adotado.

A partir das funcionalidades especificadas nos casos de uso, foram identificados os componentes de sistema, negócio e utilitários. Os componentes de sistema e negócio são respectivamente ilustrados nas Figuras 7.3 e 7.4. Na camada de sistema, as interfaces `ISBCancelaContratoContaMgr`, `ISBCadLimiteAdcMgr`, `ISBCapturaChequeMgr`, `ISBSustarChequeMgr` definem os respectivos casos de uso, e contém apenas uma operação pública cada. A interface `ISBObtencaoTalao` define os casos de uso "Requisição de talão de cheque"e "Entrega de talão de cheque", provendo duas operações públicas.

Na camada de negócio, as interfaces agrupam as operações relacionadas, requeridas pelos diferentes componentes de sistema. Para a camada `utilitária`, foi identificado o componente `componenteUtilitario`, que fornece serviços para validação de datas.

Todos os componentes especificados têm o formato do componente ideal (Seção 2.1.2), englobando os comportamentos normal e excepcional. Cada um dos comportamentos

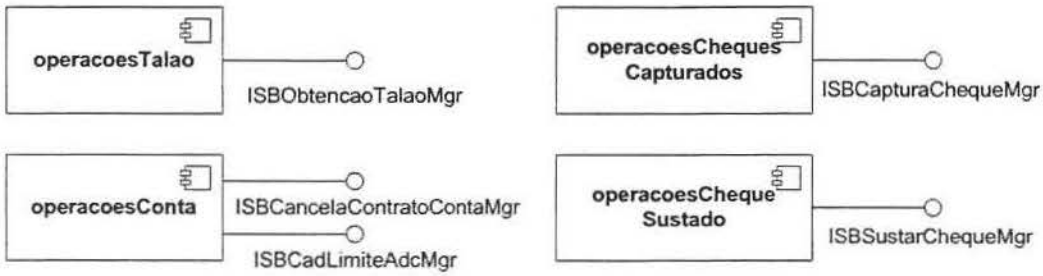

Figura 7.3: Componentes identificados para a camada de sistema.

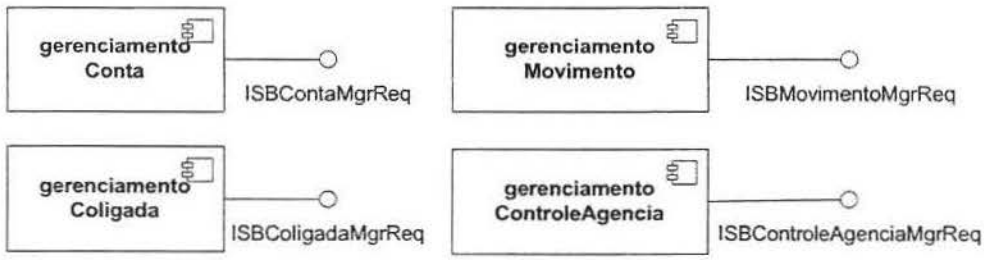

Figura 7.4: Componentes identificados para a camada de negócio.

é implementado por diferentes componentes, como apresenta a Figura 7.5, que ilustra a arquitetura interna do componente `operacoesConta`, exemplificando a estrutura do componente ideal proposta pelo MDCE+. O comportamento excepcional é implementado por vários componentes `tratadores`, ligados ao componente normal (`operacoesConta`) por um conector interno. Essa estruturação facilita tanto a reutilização dos componentes normais quanto dos tratadores.

Para a implementação dos componentes normais e excepcionais, o MDCE+ propõe a utilização do modelo COSMOS [27]. O COSMOS é um modelo independente de plataforma para a implementação de componentes de *software*, criado a partir das diretrizes de materialização de elementos arquiteturais, separação de especificação e implementação, declaração explícita das dependências entre componentes e baixo acoplamento entre as classes de implementação. A Figura 7.6 apresenta a estrutura interna do componente que implementa o comportamento normal do componente `operacoesConta` como exemplo da proposta do modelo COSMOS.

O COSMOS define três sub-modelos para implementação de diferentes aspectos do desenvolvimento baseado em componentes:

1. *Modelo de Especificação (pacote spec)*: especifica as interfaces providas e requeridas do componente, incluindo os tratadores requeridos (pacote `req.exceptional`). Os pacotes `spec` são públicos.

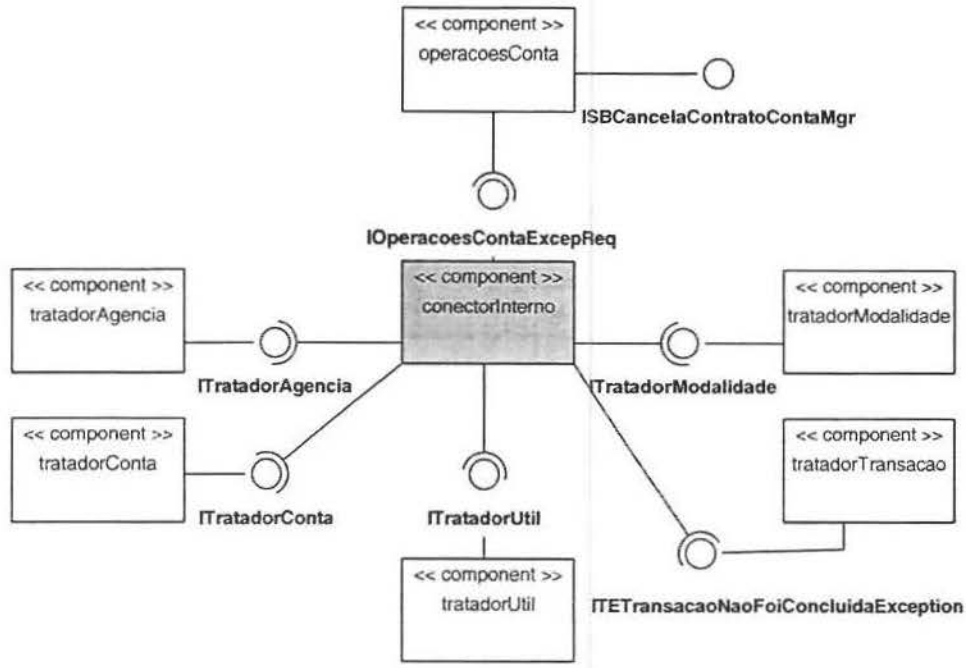

Figura 7.5: Estrutura interna do componente ideal `operacoesConta`.

2. *Modelo de Implementação (pacote impl)*: define como os serviços providos pelo componente são implementados. O pacote `impl` é privado, e formado pelas classes que implementam as funcionalidades do componente. Na Figura 7.6, as classes `FacadeISBCancelaContratoContaMgr` e `FacadeISBCadLimiteAdcMgr` implementam as respectivas interfaces. Além dessas classes, há também classes específicas do modelo COSMOS (`Manager`, `ComponentFactory` e `ObjectFactory`), que controlam a instanciação do componente.

3. *Modelo de Conectores*: especifica as conexões entre os componentes, permitindo que dois ou mais componentes sejam conectados em uma configuração determinada. Um componente sempre se conecta a outro através de um conector, que faz as ligações entre as interfaces providas e requeridas. O modelo de conectores não foi exemplificado na Figura, mas pode ser consultado em [27].

A Seção a seguir descreve a geração e execução dos testes no estudo de caso.

7.2 Testes

O primeiro passo foi a seleção dos componentes a serem testados. Foram escolhidos os componentes de sistema, já que eram o foco da aplicação em questão. Os componentes

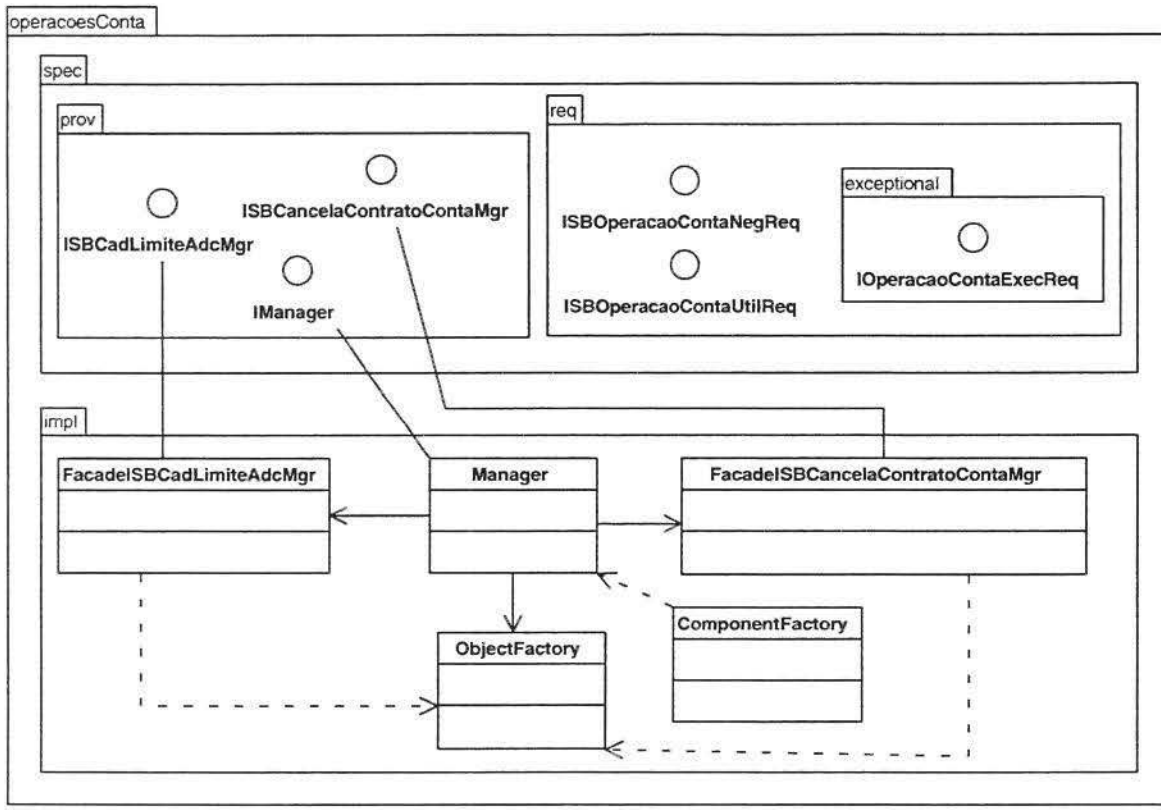

Figura 7.6: Estrutura interna do componente normal `operacoesConta`, ilustrando o modelo COSMOS.

de negócio não foram selecionados pois foram implementados parcialmente, apenas para a simulação da execução dos componentes de sistema.

Analisando cada componente de sistema, foi decidido pelo teste dos componentes na forma de componente ideal, considerando a estrutura completa como uma caixa preta. Essa decisão foi tomada devido à simplicidade dos tratadores, pois a grande maioria trata as exceções apenas com a propagação simples. Assim, foram testados de forma integrada ao componente normal. Apenas um tratador, por implementar um tratamento mais difícil de ser controlado durante os testes, foi testado isoladamente e substituído por um *stub* durante os testes do componente ideal.

Já os componentes de negócio foram todos substituídos por *stubs*, por apresentarem um comportamento mais complexo e, por isso, mais difícil de ser controlado. Os cenários foram obtidos a partir dos diagramas de atividades das interfaces requeridas, como descrito no Capítulo 5.

Os diagramas de detalhamento (exceto a parte relativa às exceções de interface) foram

criados pelos desenvolvedores como passo do MDCE+, para a descoberta das operações e das exceções lançadas pelas interfaces requeridas, que devem ser tratadas pelos componentes. O diagrama de atividades substituiu o diagrama de colaboração proposto pelo *UMLComponents* para a descoberta das operações das interfaces requeridas, trazendo vários benefícios: além de contribuírem para a geração automática dos casos de teste, os diagramas de atividades possibilitam a criação de condições de guarda para a especificação das exceções e a visualização da ordem de chamada das operações.

Aos testadores coube completar o diagrama com os fluxos relativos às exceções de interface, a partir da especificação das exceções de interface construída durante o levantamento de requisitos. Para os desenvolvedores, esta parte do diagrama não era necessária, pois dentro do MDCE+ o objetivo do diagrama é apenas representar interações com as interfaces requeridas. Durante a especificação dos diagramas, foi experimentada, sem sucesso, a criação dos diagramas de detalhamento completos pelo testador, a partir da descrição do cenário dos casos de uso. Esta experiência comprovou que, quando criados pelos desenvolvedores, os diagramas são criados mais rapidamente e sofrem menos manutenções. Para agilizar o processo, as condições de guarda foram especificadas de maneira informal pelos desenvolvedores, e depois traduzidas para OCL pelos testadores.

Já os diagramas principais foram criados facilmente pelos testadores a partir dos casos de uso. Vale ressaltar que, como o *UMLComponents* propõe a criação de uma interface por caso de uso, a criação dos diagramas de interface provida foi simplificada, já que apenas uma das interfaces tinha mais de uma operação. Caso a lógica seja mais complexa, pode ser interessante que os diagramas sejam revisados por especialistas na regra do negócio. A formalização das assertivas também foi realizada pelos testadores, a partir dos casos de uso e dos diagramas.

A geração da arquitetura do componente testável e dos casos de teste foi realizada em paralelo à implementação dos componentes, com base nos documentos produzidos durante a especificação. Este paralelismo total trouxe muitos problemas para a criação dos casos de teste relativos à discrepâncias entre implementação e especificação, gerando muitas manutenções nos testes e nas bibliotecas de aspectos. Esses problemas motivaram a criação da fase de verificação dos modelos, e o adiamento da geração das bibliotecas e dos casos de teste para o fim da implementação de cada componente.

Os casos de teste foram gerados manualmente, utilizando o *framework* JUnit. Durante a tradução do XML para casos de teste foi seguida a estrutura definida em [16] e implementada pelo JUnit: configuração do ambiente (*set up*), execução, avaliação e liberação de recursos. Para facilitar a geração manual, foi estabelecida a estrutura exibida na Figura 7.7 para o conjuto de casos de teste.

A fase de configuração, por ser similar para todos os casos de teste, foi centralizada no método `setUp()`, provido pelo *framework* JUnit e chamado automaticamente antes

```
1  public class TestISBCancelaContratoContaMgr extends TestCase {
2
3  \\Criação e instanciação do componente normal e tratadores.
4  \\Instanciação dos objetos que serão utilizados nos casos de teste.
5
6  private void assemblyComponent() { ... }
7
8  private void setStubReturn() { ... }
9
10 private void setUp() { ... }
11
12 public void testIP1 () { ... }
13 ...
14 public void testIPn () { ... }
15
16 private void tearDown() { ... }
17 }
```

Figura 7.7: Estrutura usada durante a geração manual dos casos de teste.

da execução de cada um dos casos de teste. Essa configuração foi dividida nos seguintes passos (Figura 7.7):

1. Criação e instanciação dos atributos referentes às partes do componente ideal: componente normal e tratadores.

2. Instanciação dos objetos que serão utilizados nos casos de teste (tanto como parâmetros para os métodos da interface provida quanto como retornos de *stubs*).

3. Método privado `void assemblyComponent()`: responsável pela montagem do componente ideal, o método é dividido em ligação do componente e interfaces requeridas de negócio, e ligação do componente com os tratadores.

4. Método privado `void setStubReturn()`: responsável pela atribuição de valores aos *stubs*. Neste método, os valores atribuídos seguem o caminho para término normal do caso de teste. Os retornos que levam caminhos excepcionais são atribuídos internamente ao caso de teste específico.

5. Método do JUnit `void setUp()`: realiza a criação dos objetos parâmetros dos métodos das interfaces providas (seguindo o cenário normal), e invoca os métodos `setStubReturn()` e `assemblyComponent()`.

Utilizando essa estrutura, as atividades de atribuição dos valores dos *stubs* e dos parâmetros de entrada foram suprimidas dos casos de teste, ficando restritas às mudanças necessárias para que o caminho seja percorrido.

Os casos de teste foram obtidos a partir das *tags* `TestCase` contidas no arquivo XML. Cada `TestCase` foi implementado em um método cujo nome se inicia por `test`, e que realiza as ações de execução e avaliação relativa ao caso de teste, como descrito na Seção 5.1.2. A atividade de liberação dos recursos (CST e *stubs*) foi implementada utilizando o método provido pelo JUnit, `tearDown()`, executado automaticamente após a execução de cada caso de teste. Os testes foram aplicados na fase de montagem, após a implementação dos conectores do componente ideal, e os resultados são apresentados na próxima seção.

7.3 Resultados e Avaliação

Nesta Seção é apresentada a análise tanto do impacto da intrumentação no espaço de armazenamento do CST quanto dos resultados dos testes.

7.3.1 Instrumentação

As estatísticas referentes à criação dos aspectos contidos nas bibliotecas dos componentes Tester e Tracker são apresentadas na Tabela 7.1. A tabela é dividida de acordo com as interfaces dos componentes instrumentados. Sobre os aspectos, são apresentadas informações como o número de aspectos criados nas bibliotecas Tracker e Tester, e a quantidade de linhas de código necessárias. Sobre o impacto da instrumentação sobre o CST, foram medidas as alterações no tamanho dos *bytecodes* gerados. Análises sobre impactos no desempenho do CST serão realizadas em trabalhos futuros.

O tamanho do executável foi medido apenas em relação à classe que implementa a interface, já que é o único ponto do componente modificado pela instrumentação. Foram medidos o espaço de armazenamento dos *bytecodes* da classe do CST antes da instrumentação (coluna "Sem Instr."), após a inserção de cada biblioteca separadamente, e após a inserção de ambas. A medida da classe instrumentada considera o novo tamanho da classe do CST mais dos *bytecodes* gerados a partir dos aspectos, já que o código inserido no CST faz referência a esses aspectos. Não foram consideradas as classes para registro no *log* (LogAssert - 4KB e LogTrace - 4,4KB), por serem incluídas apenas uma vez no CST, qualquer que seja a quantidade de interfaces instrumentadas. As outras classes auxiliares (Testing e Setup) também não foram consideradas pois não são incluídas na versão executável do CST, sendo utilizadas apenas para a criação desta versão.

Os dados apresentados refletem uma grande interferência da instrumentação no espaço de armazenamento do CST, aumentando em até 5 vezes o seu tamanho, em alguns casos (interface `ISBCadLimiteAdc`). Também pode ser observado que, geralmente, as bibliotecas para verificação de assertivas exercem maior impacto, já que o código contido nos aspectos é maior e mais complexo, como demonstrado pela contagem de linhas de código.

Instrumentação dos Componentes					
Componente	operacoesChequeCapturado				
Interface	ISBCapturaChequeMgr				
Biblioteca	**Aspectos**		**Tamanho executável CST***		
	Quantidade	**Linhas de Código**	**Sem Instr.**	**Instrumentado****	**Aumento**
Tracker	2	120	12,8 KB	15,3 KB + 14,8 KB = 30,1 KB	135,2%
Tester	1	325	12,8 KB	14,2 KB + 23,6 KB = 37,8 KB	195,3%
Total	3	445	12,8 KB	16,2 KB + 38,4 KB = 54,6 KB	326,6%
Componente	operacoesTalao				
Interface	ISBObtencaoTalaoMgr				
Biblioteca	**Aspectos**		**Tamanho executável CST***		
	Quantidade	**Linhas de Código**	**Sem Instr.**	**Instrumentado****	**Aumento**
Tracker	2	368	17,4 KB	21,5 KB + 34,8 KB = 56,3 KB	123,6%
Tester	2	558	17,4 KB	21,7 KB + 39,5 KB = 61,2 KB	251,7%
Total	4	926	17,4 KB	28,8 KB + 74,3 KB = 103,1 KB	492,5%
Componente	operacoesChequeSustado				
Interface	ISBSustarChequeMgr				
Biblioteca	**Aspectos**		**Tamanho executável CST***		
	Quantidade	**Linhas de Código**	**Sem Instr.**	**Instrumentado****	**Aumento**
Tracker	2	197	11,7 KB	15,8 KB + 22,6 KB = 38,4 KB	228,2%
Tester	1	255	11,7 KB	13,4 KB + 16,7 KB = 30,1 KB	157,3%
Total	3	452	11,7 KB	17,0 KB + 39,3 KB = 56,3 KB	381,2%
Componente	operacoesConta				
Interface	ISBCadLimiteAdcMgr				
Biblioteca	**Aspectos**		**Tamanho executável CST***		
	Quantidade	**Linhas de Código**	**Sem Instr.**	**Instrumentado****	**Aumento**
Tracker	2	217	9,1 KB	13,7 KB + 22,5 KB = 36,2 KB	297,8%
Tester	1	277	9,1 KB	11,3 KB + 17,4 KB = 28,7 KB	315,4%
Total	3	494	9,1 KB	15,5 KB + 39,9 KB = 55,4 KB	508,8%
Interface	ISBCancelaContratoContaMgr				
Biblioteca	**Aspectos**		**Tamanho executável CST***		
	Quantidade	**Linhas de Código**	**Sem Instr.**	**Instrumentado****	**Aumento**
Tracker	2	242	11,5 KB	16,3 KB + 25,5 KB = 41,8 KB	363,5%
Tester	1	329	11,5 KB	13,9 KB + 21,2 KB = 35,1 KB	205,2%
Total	3	571	11,5 KB	18,2 KB + 46,7 KB = 64,9 KB	464,3%

*Foi considerado como CST apenas a classe FacadeN, que implementa a interface, pois é a única a ser instrumentada pelos aspectos.

**O tamanho medido compreende o tamanho do CST (classe FacadeN) instrumentado mais o tamanho dos aspectos inseridos.

Tabela 7.1: Estatísticas relativas à instrumentação dos componentes.

Com base nas informações, recomenda-se que a inserção da instrumentação seja realizada de acordo com os testes, para que tanto o espaço de armazenamento quanto os arquivos de *log* não sejam preenchidos com informações pouco relevantes para o contexto dos testes. Outra recomendação é a remoção da instrumentação na versão operacional do componente, para não sobrecarregar o espaço de armazenamento do sistema final.

7.3.2 Casos de Teste

As estatísticas dos testes são apresentadas na Tabela 7.2. Todas as falhas encontradas são relativas à verificação de parâmetros de entrada, ou seja, algumas exceções de interface não eram devidamente tratadas. A partir desses dados, foi possível perceber que, como a parte

do diagrama de detalhamento correspondente a exceções de interface foi especificada pelo testador, algumas das situações excepcionais não foram capturadas pelo desenvolvedor ao especificar e implementar o componente, comprovando a riqueza dos testes quando elaborados por uma equipe independente.

Relatório de encaminhamento do item de teste			
Contexto	Relatório dos testes dos componentes ideais		
Participantes	Camila Rocha		
Abrangência	Componentes de Sistema operacoesConta, operacoesTalao, operacoesChequeSustado e operacoesChequeCapturado, integrados aos tratadores e ao componenteUtilitário		
Componente	operacoesChequeCapturado		
Interface	ISBCapturaChequeMgr		
Total Casos de Teste	27		
Sumário dos Defeitos	**Quantidade**	**Porcentagem**	**Casos de Teste**
Total	3	11,1%	IP4, IP5 e IP6
JUnit	2	7,4%	IP5 e IP6
Contrato	2	7,4%	IP4 e IP5
Componente	operacoesTalao		
Interface	ISBObtencaoTalaoMgr		
Total Casos de Teste	60		
Sumário dos Defeitos	**Quantidade**	**Porcentagem**	**Casos de Teste**
JUnit	2	3,3%	IP20 e IP21
Contrato	2	3,3%	IP20 e IP21
Componente	operacoesChequeSustado		
Interface	ISBSustarChequeMgr		
Total Casos de Teste	60		
Sumário dos Defeitos	**Quantidade**	**Porcentagem**	**Casos de Teste**
JUnit	4	6,7%	IP3, IP4, IP11 e IP12
Contrato	3	5,0%	IP3, IP4 e IP11
Componente	operacoesConta		
Interface	ISBCadLimiteAdcMgr		
Total Casos de Teste	20		
Sumário dos Defeitos	**Quantidade**	**Porcentagem**	**Casos de Teste**
JUnit	2	10,0%	IP9, IP10
Contrato	2	10,0%	IP9, IP10
Interface	ISBCancelaContratoContaMgr		
Total Casos de Teste	27		
Sumário dos Defeitos	**Quantidade**	**Porcentagem**	**Casos de Teste**
JUnit	0	0,0%	
Contrato	0	0,0%	

Tabela 7.2: Relatório de encaminhamento do item de teste.

Os dados também mostraram equivalência entre os oráculos das assertivas e do caso de teste, sendo que o do caso de teste foi superior em uma situação específica, a de verificação de datas, por uma particularidade da linguagem Java. Como a linguagem considera uma data como um número inteiro, mesmo quando uma data inválida é fornecida (com o parâmetro dia com o valor 32, por exemplo), a transformação para inteiro a transforma em uma data válida, porém diferente da registrada inicialmente. Assim, as assertivas consideraram como uma data válida, porém o caso de teste registrou o não lançamento da exceção prevista.

Apesar de neste estudo de caso o oráculo contido nos casos de teste ter sido superior

em algumas situações, as assertivas tinham um poder maior por verificarem mais detalhadamente os objetos recebidos e retornados. Esse poder não foi demonstrado pois, em casos como a verificação do contexto de exceções por exemplo, as exceções eram lançadas por *stubs*.

Como análise da eficácia dos testes, realizou-se uma avaliação em relação à sua cobertura. Por serem testes caixa preta, foi analisada a cobertura dos modelos, e comprovada que todas as arestas foram percorridas, alcançando o objetivo do percurso. Outra avaliação foi realizada em relação à cobertura dos testes gerados a partir do modelo, com a análise da cobertura do código. Assim, os testes foram aplicados utilizando o ambiente djUnit [2], *plug-in* (módulo) do ambiente Eclipse que aplica os testes no formato JUnit e realiza a análise de cobertura do código. As estatísticas relativas à cobertura de código são apresentadas na Tabela 7.3.

Cobertura dos Testes			
Componente*	**Linhas de Código**	**Cobertura das Linhas**	**Cobertura das Decisões**
Cobertura Geral	4583	68%	85%
operacoesChequeCapturado	109	88%	96%
operacoesChequeSustado	88	96%	97%
operacoesConta	154	95%	100%
operacoesTalao	167	98%	100%
componenteUtilitario	28	46%	58%
tratadorAgencia	7	71%	100%
tratadorChequeSustado	8	87%	100%
tratadorChequesCapturados	19	84%	100%
tratadorCliente	7	85%	100%
tratadorColigada	5	80%	100%
tratadorConta	13	76%	100%
tratadorModalidade	6	83%	100%
tratadorTalao	22	80%	100%
tratadorTransacao	5	100%	100%
tratadorUtil	8	75%	100%

*Foram consideradas apenas as classes que implementam as interfaces.

Tabela 7.3: Estatísticas relativas à cobertura de código dos testes realizados.

A cobertura foi analisada apenas nas classes que implementam as interfaces (Facade do modelo COSMOS), tanto nos componentes normais quanto dos tratadores. A cobertura foi bastante satisfatória, estando acima de 80% na maioria dos casos, e acima de 88% para os componentes normais, o principal alvo dos testes. Além disso, grande parte dos motivos pela diminuição da taxa de cobertura foram problemas na implementação.

A cobertura geral ficou em 68% devido a decisões de projeto tomadas durante a implementação, que tornaram parte do código obsoleto (mas que não foi removido do componente). Estas mudanças atingiram especialmente os tratadores e conectores internos dos componentes ideais. Outra decisão que afetou a cobertura foi a não utilização de todos os serviços providos pelas classes de controle do modelo COSMOS, deixando de invocar alguns de seus métodos, como os que listam os métodos das interfaces providas e

requeridas, por exemplo.

Um caso particular, que pode ser observado na Tabela 7.3, ocorreu com o `componenteUtilitario` que, por ter sido reutilizado de um projeto anterior, não teve todas as suas operações invocadas neste projeto, apresentando baixa cobertura.

7.4 Considerações Finais

Neste capítulo foi descrito o estudo de caso realizado em conjunto com exercício do método de desenvolvimento, em um ambiente real. A partir deste estudo de caso, aconteceram alterações importantes tanto no método de desenvolvimento quanto de testes. Estas alterações contribuíram principalmente para a agilidade e integração entre os métodos.

Também foi apresentada a análise sobre a instrumentação e os resultados dos testes. Foi constatada uma grande interferência da instrumentação sobre o espaço de armazenamento do componente, levando à recomendação de utilizá-la apenas durante os testes. Sobre a aplicação dos testes, apesar do pequeno número de falhas, foi constatado o sucesso da abordagem de criação de casos de teste a partir da análise de cobertura do código.

Capítulo 8

Conclusões e Trabalhos Futuros

Com a larga utilização do desenvolvimento baseado em componentes para construção de sistemas computacionais, por trazer redução do tempo e custo de desenvolvimento através da reutilização de código, até mesmo sistemas considerados críticos são construídos desta maneira.

A qualidade dos componentes porém, continua dependente da realização de testes a cada novo contexto. Para que a redução dos custos no desenvolvimento não seja perdida nas inúmeras repetições dos testes, a testabilidade do componente é um fator de grande importância durante o desenvolvimento, já que facilita a realização dos testes, diminuindo o custo dessa fase.

Em sistemas críticos, usualmente formados por componentes tolerantes a falhas, a realização sistemática de testes em seus componentes é imprescindível, já que a confiança no funcionamento (*dependability*) é uma característica fundamental. As dificuldades para este tipo de componente também aumentam, já que muitas situações são difíceis de serem simuladas, especialmente nos testes da parte excepcional do comportamento do componente.

Neste trabalho foi proposto um método para melhoria da testabilidade de componentes. O principal enfoque foram sistemas críticos, formados por componentes tolerantes a falhas. O método visa a melhoria da testabilidade do componente de forma caixa preta, buscando facilitar a realização de testes pelo cliente do componente, que não possui acesso ao código fonte.

Neste capítulo é apresentado um resumo das atividades realizadas (Seção 8.1), as principais contribuições do trabalho (Seção 8.2) e sugestões de trabalhos futuros (Seção 8.3).

8.1 Atividades Realizadas

A primeira etapa do trabalho foi o estudo das diversas abordagens para o aumento da testabilidade de componentes, como apresentado no Capítulo 2. Uma análise crítica dos trabalhos de Toyota [74] e Ukuma [76], que serviram de base para este trabalho, foi cuidadosamente realizada, para levantamento dos pontos a serem melhorados. Foi estudada também a teoria relativa a componentes tolerantes a falhas.

Foram então definidos os aspectos da testabilidade do componente que seriam atacados. Foi decidida pela melhoria da representação com a criação de especificações que seriam empacotadas junto com o componente, e a introdução de mecanismos BIT para monitoração da execução e verificação de assertivas.

O passo seguinte foi a definição da arquitetura do componente testável, com experimentações com as abordagens de manipulação de *bytecodes* e programação orientada a aspectos para implementação das bibliotecas. Após uma análise crítica, a programação a aspectos foi preferida pela facilidade de uso.

O próximo passo foi a definição dos modelos comportamental e contratual do componente, de forma que os casos de teste e o código executável das assertivas pudessem ser gerados automaticamente. Foram escolhidas notações UML: o diagrama de atividades, para a especificação comportamental, e a OCL (*Object Constraint Language*, para a contratual. Foram estabelecidas regras para a construção de ambas, gerando modelos formais dos quais os casos de teste e assertivas podem ser automaticamente gerados, com atenção especial ao comportamento excepcional. Juntamente com a definição do formato das especificações, foram estabelecidos os passos a serem implementados pelas ferramentas futuras de geração de casos de teste e de código executável das assertivas.

Com base na construção do componente testável, foi finalmente elaborado um método de testes para componentes, contendo os passos para a construção do componente testável, geração e execução dos casos de teste. O método foi definido juntamente com o MDCE+ (Método para Definição do Comportamento Excepcional), para construção de sistemas confiáveis. As atividades de teste de componentes foram dispostas ao longo do desenvolvimento, reutilizando os modelos construídos pelos desenvolvedores para a geração dos testes.

Ambos os métodos foram refinados em um estudo de caso realizado em um ambiente real, no qual a integração entre os métodos pôde ser melhorada. Durante este estudo de caso foi também realizada uma análise crítica do trabalho.

8.2 Principais Contribuições

Neste trabalho foi proposto um método para testes de componentes tolerantes a falhas, cuja elaboração foi guiada para o aumento da testabilidade dos componentes produzidos. As principais contribuições foram:

- A arquitetura do componente testável, permitindo a inclusão de mecanismos BIT mesmo em componentes cujo código fonte não está disponível;
- Construção de uma estrutura reutilizável para instrumentação de componentes, encapsulando as mudanças relativas aos mecanismos de um componente em bibliotecas de aspectos;
- Um formato para a especificação do contrato do componente em OCL (englobando regras para os comportamentos normal e excepcional) e os passos para geração autmomática dos mecanismos BIT para verificação de assertivas;
- Os passos para geração dos mecanismos BIT de monitoração da execução do componente a partir de suas interfaces públicas;
- Um formato para especificação comportamental, utilizando o diagrama de atividades da UML, modelando o fluxo de execução das interfaces providas do componente e relacionando-os com as interfaces requeridas, incluindo o comportamento excepcional;
- Diretrizes para geração automática de *drivers*, *stubs* e da sincronização entre eles a partir dos diagramas de atividades;
- Um método para contrução de componentes testáveis paralelamente ao seu desenvolvimento, e para a execução e avaliação dos resultados de teste;
- Refinamento do método MDCE+, diminuindo o custo da criação dos artefatos de teste.

8.3 Trabalhos Futuros

Por ser apenas o início da proposta de um método completo de testes com ênfase em automação, vários trabalhos futuros podem sugeridos:

- Medidas do impacto da inclusão dos mecanismos BIT na performance do componente;

- Implementação de ferramentas de apoio:
  - Para geração da arquitetura do componente testável;
  - Para geração das bibliotecas de monitoração a partir das interfaces dos componentes e, de verificação das assertivas, a partir das especificações contratuais;
  - Para geração dos casos de teste a partir das especificações comportamentais;
- Estudos sobre a obtenção automática de dados de teste;
- Adaptação do método de testes a outros processos de desenvolvimento;
- Definição de um método para construção da arquitetura do componente testável para componentes COTS;
- Aplicação em outros estudos de caso, para medida da eficácia da metodologia;
- Definição do processo completo de testes para sistemas confiáveis, acrescentando diretrizes para testes unitários, de integração e de sistema.

Referências Bibliográficas

[1] *ANSI/IEEE Standard 829-1983: IEEE Standard for Software Test Documentation.* The Institue of Electrical and Electronic Engineers, 1987.

[2] djunit. http://works.dgic.co.jp/djunit/, 2005.

[3] Thomas Anderson e Peter A. Lee. *Fault Tolerance: Principles and Practice.* Prentice-Hall, 2ª edição, 1990.

[4] Tom Anderson, Mei Feng, Steve Riddle e Alexander B. Romanovsky. Protective wrapper development: A case study. In: M. Hakan Erdogmus e Tao Weng, editores, *ICCBSS*, volume 2580 de *Lecture Notes in Computer Science*, pp. 1–14. Springer, 2003.

[5] Colin Atkinson, Joachim Bayer e Dirk Muthig. Component-based product line development: The kobrA approach. In: P. Donohoe, editor, *Proceedings of theFirst-Software Product Line Conference*, pp. 289–309, 2000.

[6] D. L. Barbosa, W. L. Andrade, P. D. L. Machado e J. C. A. Figueiredo. Spaces - uma ferramenta para teste funcional de componentes. In: *Anais de XVIII Simpósio Brasileiro de Engenharia de Software - XI Sessão de Ferramentas*, pp. 55–60, Outubro 2004.

[7] Kent Beck e Erick Gamma. Junit test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[8] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* John Wiley and Sons, Inc., 1995.

[9] Boris Beizer. *Sofware Testing Techniques.* International Thomson Computer Press, 2 edição, 1997.

[10] Sami Beydeda e Volker Gruhn. Merging components and testing tools: The self-testing cots components (stecc) strategy. In: *EUROMICRO*, pp. 107–115. IEEE Computer Society, 2003.

[11] Sami Beydeda e Volker Gruhn. State of the art in testing components. In: *3rd International Conference on Quality Software*, 2003.

[12] Sami Beydeda e Volker Gruhn. Black- and white-box self-testing cots components. In: Frank Maurer e Günther Ruhe, editores, *SEKE*, pp. 104–109, 2004.

[13] Sami Beydeda, Volker Gruhn, Johannes Mayer, Ralf Reussner e Franz Schweiggert, editores. *Testing of Component-Based Systems and Software Quality, Proceedings of SOQUA 2004 (First International Workshop on Software Quality) and TECOS 2004 (Workshop Testing Component-Based Systems)*, volume 58 de *LNI*. GI, 2004.

[14] Adrita Bhor. Software component testing strategies. Relatório Técnico UCI-ICS-02-06, University of California, Irvine, June 2001.

[15] Robert V. Binder. Design for testability in object-oriented systems. *Communications of the ACM*, 37(9):87–101, 1994.

[16] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[17] Grady Booch, James Rumbaugh e Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[18] L. C. Briand, Y. Labiche e H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Software - Practice & Experience*, 33(7):637–672, 2003.

[19] Lionel Briand e Yvan Labiche. A uml-based approach to system testing. *Software and Systems Modeling*, 1(1):10–42, September 2002.

[20] Alan W. Brown e Kurt C. Wallnau. The current state of cbse. *IEEE Software*, 15(5):37–46, 1998.

[21] Gary A. Bundell, Gareth Lee, John Morris e Kris Parker. A software component verification tool. In: *International Conference on Software Methods and Tools (SMT)*, pp. 137–146. IEEE Computer Society Press, 2001.

[22] John Chessman e John Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Paperback, 2002.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[24] Adalberto N. Crespo, Odair J. da Silva, Carlos A. Borges, Clênio F. Salviano, Miguel T. A. Junior e Mario Jino. Uma metodologia para teste de software no contexto da melhoria do processo. In: *III Simpósio Brasileiro de Qualidade de Software (SBQS)*, Junho 2004.

[25] Flaviu Cristian. Exception handling. In: T. Anderson, editor, *Dependability of Resilient Computers*, pp. 68–97. Blackwell Scientific Publications, 1989.

[26] Patrick Henrique da Silva Brito, Fernando Castor Filho e Cecília Mary Fischer Rubira. Um método para modelagem de exceções em desenvolvimento baseado em componentes. In: *Anais do IV Workshop de Desenvolvimento Baseado em Componentes (WDBC'2004)*, Setembro 2004.

[27] Moacir C. da Silva Jr., Paulo Astério de C. Guerra e Cecília M. F. Rubira. A java component model for evolving software systems. In: *Proc. of the ASE*, pp. 327–330, 2003.

[28] Markus Dahm. Byte code engineering with the bcel api. Relatório Técnico B-17-98, Freie Universit at Berlin, Institut für Informatik, Abril 2001.

[29] Paulo Asterio de C. Guerra, Cecília Mary F. Rubira, Alexander B. Romanovsky e Rogério de Lemos. A dependable architecture for cots-based software systems using protective wrappers. In: Rogério de Lemos, Cristina Gacek e Alexander Romanovsky, editores, *WADS*, volume 3069 de *Lecture Notes in Computer Science*, pp. 144–166. Springer, 2003.

[30] Paulo Asterio de Castro Guerra, Carolina Stephan de Araújo, Camila Ribeiro Rocha e Eliane Martins. Cbdunit - uma ferramenta para testes unitários de componentes. In: *Anais de XIX Simpósio Brasileiro de Engenharia de Software - XII Sessão de Ferramentas*, Outubro 2005.

[31] Wilson de Pádua Paula Filho. *Engenharia de Software: Fundamentos, Métodos e Padrões.* Livros Técnicos e Científicos, 1ª edição, 2001.

[32] Rodger D. Drabick. *Best Practices for the Formal Software Testing Process: A Menu of Testing Tasks.* Dorset House Publishing CO., Inc., 2004.

[33] Desmond d'Souza e Alan C. Will. *Objects, Components and Frameworks with UML: the Catalysis Approach.* Addison-Wesley, 1998.

[34] Stephen H. Edwards. A framework for practical, automated black-box testing of component-based software. *Software Testing, Verification and Reliability*, 11(2):97–111, June 2001.

[35] Carina M. Farias e Patricia D. L. Barbosa. Um método de teste funcional para verificação de componentes. In: *Anais de XVII Simpósio Brasileiro de Engenharia de Software*, Outubro 2003.

[36] Gisele R. M. Ferreira. Tratamento de exceções no desenvolvimento de sistemas confiáveis baseados em componentes. Dissertação de mestrado, IC, Unicamp, Dezembro 2001.

[37] Roy S. Freedman. Testability of software components. *IEEE Transactions on Software Engineering*, 17(6):553–564, 1991.

[38] Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[39] Jerry Gao. Component testability and component testing challenges. In: *Proc. of International Workshop on Component-Based Software Engineering*, 2000.

[40] Jerry Z. Gao, Kamal K. Gupta, Shalini Gupta e Simon S. Y. Shim. On building testable software components. In: *Proc. First International Conference on COTS-Based Software Systems*, pp. 108–121, Fevereiro 2002.

[41] Object Management Group. *OMG Unified Modeling Language Specification Version 1.5*, 2003.

[42] Object Management Group. *UML 2.0 OCL Specification*, 2003.

[43] Object Management Group. The omg's corba website. http://www.corba.org, 2004.

[44] Object Management Group. *UML 2.0 Superstructure Final Adopted Specification*, 2004.

[45] Object Management Group. *UML 2.0 Testing Profile Specification*, 2004.

[46] Daniel Hoffman. Hardware testing and software ics. In: *Proc. Pacific Northwest Software Quality Conference*, pp. 234–244, 1998. Portland, Oregon.

[47] Jon Hopkins. Component primer. *Communications of the ACM*, 43(10):27–30, 2000.

[48] Jonas Hörnstein e Hakan Edler. Test reuse in cbse using built-in tests. In: *Workshop on Component-based Software Engineering*, Abril 2002.

[49] Object Mentor Inc. Junit.org. http://www.junit.org, 2004.

[50] Sun Microsystems Inc. Core java technologies techinical tips: Updating jar files. http://java.sun.com/developer/JDCTechTips/2004/tt0727.html, 2004.

[51] Sun Microsystems Inc. Enterprise javabeans technology. http://java.sun.com/products/ejb, 2004.

[52] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier e John Irwin. Aspect-oriented programming. In: Mehmet Akşit e Satoshi Matsuoka, editores, *Proc. European Conference on Object-Oriented Programming*, volume 1241, pp. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[53] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Company, 2003.

[54] Julius C. B. Leite e Orlando G. Loques. Introdução à tolerância a falhas. In: *Anais do II Simpósio de Computação Tolerante a Falhas*, 1987.

[55] Pattie Maes. Concepts and experiments in computational reflection. In: *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pp. 147–155. ACM Press, 1987.

[56] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 2ª edição, 1997.

[57] Simon Monk e Stephen Hall. Virtual mock objects using aspectj with junit. *XP Magazine*, 2002.

[58] Glenford Myers. *The Art of Software Testing.* John Wiley and Sons, 1979.

[59] G. Neumann e U. Zdun. Xotcl, an object-oriented scripting language. In: *Proceedings of Tcl2k: The 7th USENIX Tcl/Tk Conference*, Austin, Texas, USA, February 2000.

[60] The Mozilla Organization. Bugzilla. http://www.bugzilla.org/, 2005.

[61] Alessandro Orso, Mary Jean Harrold e David Rosemblum. Component metadata for software engineering tasks. *LNCS*, 1999:129–144, 1999.

[62] Macario Polo e Alejandra Cechich. An aspect-based environment for cots component testing. In: Beydeda et al. [13], pp. 17–29.

[63] Roger S. Pressman. *Software Engineering: a Practitioner's Approach.* McGraw-Hill, 5ª edição, 2001.

[64] The Eclipse Project. Eclipse project. http://www.eclipse.org/, 2005.

[65] Camila Ribeiro Rocha, Patrick Henrique da Silva Brito, Eliane Martins e Cecília Mary Fischer Rubira. Um método de desenvolvimento e testes para sistemas confiáveis baseados em componentes: Um estudo de caso. Relatório técnico, A ser publicado.

[66] Camila Ribeiro Rocha e Eliane Martins. A strategy to improve component testability without source code. In: Beydeda et al. [13], pp. 47–62.

[67] R. M. Sitaraman e B. W. Weide. Component-based software engineering using resolve. *ACM SIGSOFT Software Enigineering Notes*, 19(4):21–67, 1994.

[68] Neelam Soundarajan e Stephen Fridella. Modeling exceptional behavior. In: Robert B. France e Bernhard Rumpe, editores, *UML*, volume 1723 de *Lecture Notes in Computer Science*, pp. 691–705. Springer, 1999.

[69] Mark Strembeck e Uwe Zdun. Scenario-based component testing using embedded metadata. In: Beydeda et al. [13], pp. 31–45.

[70] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[71] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, Jr. E. James Whitehead, Jason E. Robbins, Kari A. Nies, Peyman Oreizy e Deborah L. Dubrow. A component- and message-based architectural style for gui software. *IEEE Transactions on Software Engineering*, 22(6):390–406, 1996.

[72] The Aspect# Team. Aspect#. http://aspectsharp.sourceforge.net/index.html, 2004.

[73] Rodrigo Teruo Tomita, Fernando Castor Filho e Cecília Mary Fischer Rubira. Bellatrix: Um ambiente para suporte arquitetural ao desenvolvimento baseado em componentes. In: *Anais do IV Workshop de Desenvolvimento Baseado em Componentes (WDBC'2004)*, Setembro 2004.

[74] Cristina M. Toyota. Melhoria da testabilidade de classes usando o conceito de autoteste. Dissertação de mestrado, IC, Unicamp, Junho 2000.

[75] Yves Le Traon, Daniel Deveaux e Jean-Marc Jézéquel. Self-testable components: from pragmatic tests to design-for-testability methodology. In: *Technology of Object-Oriented Languages and Systems*, pp. 96–107. IEEE Computer Society Press, Jun. 1999.

[76] Luciano H. Ukuma. Uma estratégia para o desenvolvimento de componentes de software autotestáveis. Dissertação de mestrado, IC, Unicamp, Abril 2002.

[77] Padmal Vitharana. Risks and challenges of component-based software development. *Communications of the ACM*, 46(8):67–72, 2003.

[78] Jeffrey M. Voas. Certifying off-the-shelf software components. *IEEE Computer*, 31(6):53–59, 1998.

[79] Jeffrey M. Voas e Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12(3):17–28, May 1995.

[80] W3C. *XML Schema Part 0: Primer Second*, 2004.

[81] Yingxu Wang, Graham King e Hakan Wickburg. A method for built-in tests in component-based software maintenance. In: *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 186–189. IEEE Computer Society Press, 1999.

[82] Jos Warmer e Anneke Kleppe. *The Object Constraint Language: Getting your Models Ready for MDA*. Addison-Wesley, 2 edição, 2003.

[83] Gerson Mizuta Weiss. Adaptação de componentes de software para o desenvolvimento de sistemas confiáveis. Dissertação de mestrado, IC, Unicamp, Maio 2001.

[84] Elaine J. Weyuker. Testing component-based software: a cautionary tale. *IEEE Software*, 15(5):54–59, September/October 1998.

Apêndice A

Configuração do Componente Testável

Neste apêndice são apresentados os códigos fonte das classes Testing e Setup, responsáveis por embutir os mecanismos de teste no componente testável. A instrumentação foi implementada utilizando arquivos JAR (*Java Archive*), um formato da linguagem Java que permite o empacotamento de vários arquivos em um único arquivo de terminação .jar. Sendo assim, tanto o CST quanto os componentes Tracker e Tester devem estar empacotados como arquivos jar.

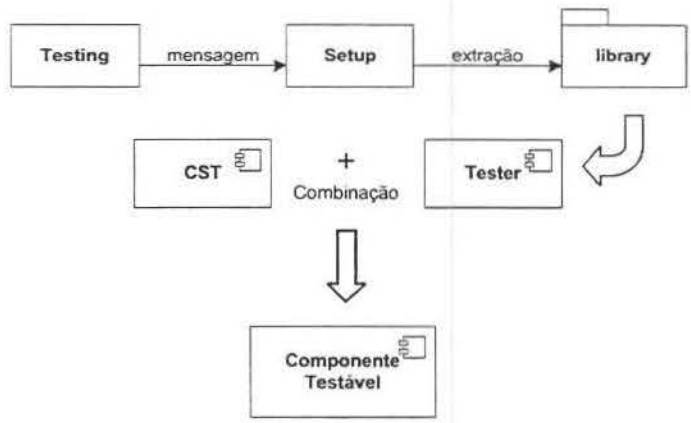

Figura A.1: Esquema da inclusão da instrumentação nos *bytecodes* do componente sob teste.

A instrumentação é realizada como ilustrado na Figura A.1: a classe Testing recebe a requisição de quais aspectos devem ser incluídos no componente, e solicita à classe Setup que realize a instrumentação. Esta, por sua vez, acessa o pacote (.jar) da biblioteca do componente Tester (`library`), e extrai os aspectos solicitados, formando um novo pacote

(tester.jar). No momento da carga da aplicação, o pacote tester.jar é combinado aos *bytecodes* do componente, e a instrumentação é carregada juntamente com o código do componente.

A.1 Classe Testing

A classe Testing é responsável pela implementação da interface ITesting. É através dela que o usuário escolhe quais aspectos serão instrumentados ao componente.

```
1
2  package tester;
3
4  import java.io.*;
5
6  public class Testing implements ITesting {
7
8   private static Setup setup = new Setup("tester\\library.jar");
9
10  public void finalize() {  //todos os aspectos já foram escolhidos, e o pacote tester.jar
11    setup.persistTempJar(); //é persistido.
12  }
13
14  public void insertInvariant(String interface_name) throws FileNotFoundException {
15   try {
16     //o aspecto correspondente é procurado dentro da estrutura do pacote library
17     File dir = new File ("tester\\library\\" + interface_name + "\\");
18     File [] dir_file = dir.listFiles();
19     int i = 0;
20     while (i < dir_file.length) {
21       if (dir_file[i].getName().endsWith("Inv.class")) {
22         System.out.println("Adding new file : " + dir_file[i].getName());
23         //requisita a classe Setup a adição do aspecto
24         setup.addNewFile(dir_file[i].getPath());
25       }
26     i++;
27     }
28   } catch (FileNotFoundException f) {
29     //se o aspecto nao foi encontrado, significa que a invariante daquela interface
30     //não existe ou nao foi implementada
31       throw new FileNotFoundException("Monitoracao das invariantes dos metodos de " +
32                                       interface_name + "nao foi implementada.");
33   }
34  }
35
```

```
36  public void insertPostcondition(String interface_name) throws FileNotFoundException {
37   try {
38     File dir = new File ("tester\\library\\" + interface_name + "\\");
39     File [] dir_file = dir.listFiles();
40     int i = 0;
41     //as pós condições de todos os métodos são inseridas, isto é, todos os arquivos
42     //terminados em "Post" contidos no pacote da interface
43     while (i < dir_file.length) {
44       if (dir_file[i].getName().endsWith("Post.class")) {
45         System.out.println("Adding new file : " + dir_file[i].getName());
46         setup.addNewFile(dir_file[i].getPath());
47       }
48       i++;
49     }
50   } catch (FileNotFoundException f) {
51     throw new FileNotFoundException("Monitoracao das pos-condicoes dos metodos de " +
52                                     interface_name + "nao foi implementada.");
53   }
54  }
55
56  public void insertPrecondition(String interface_name) throws FileNotFoundException {
57   try {
58     File dir = new File ("tester\\library\\" + interface_name + "\\");
59     File [] dir_file = dir.listFiles();
60     int i = 0;
61     //as pré condições de todos os métodos são inseridas, isto é, todos os arquivos
62     //terminados em "Pre" contidos no pacote da interface
63     while (i < dir_file.length) {
64       if (dir_file[i].getName().endsWith("Pre.class")) {
65         System.out.println("Adding new file : " + dir_file[i].getName());
66         setup.addNewFile(dir_file[i].getPath());
67       }
68     i++;
69     }
70   } catch (FileNotFoundException f) {
71     throw new FileNotFoundException("Monitoracao das pre-condicoes dos metodos de " +
72                                     interface_name + "nao foi implementada.");
73   }
74  }
75
76 }
```

A.2 Classe Setup

O código da classe Setup foi baseado em [50]. Esta classe recebe as requisições da classe Testing e cria um novo pacote contendo os aspectos determinados.

```
1  package tester;
2
3  import java.io.*;
4  import java.util.*;
5  import java.util.zip.*;
6  import java.util.jar.*;
7
8  public class Setup {
9
10  File tempJar = null;      //pacote com os novos aspectos (tester.jar)
11  JarOutputStream newJar = null;
12  libraryJar jar = null;    //arquivo temporario igual a library.jar
13  boolean success = true;
14
15  //Inicialização de variáveis
16  public Setup(String oldJar){
17   try {
18     tempJar = File.createTempFile("tester.jar", null);
19   } catch (IOException e) {
20     success = false;
21     System.err.println("Unable to create intermediate file.");
22     System.exit(-2);
23   }
24   try {
25     jar = new libraryJar(oldJar);  //libraryJar recebe o library.jar original
26   } catch (IOException e) {
27     success = false;
28     System.err.println("Unable to access original file.");
29     System.exit(-3);
30   }
31   copyOldFiles();
32  }
33
34  //Copia library.jar e o arquivo da classe LogTrace para o libraryJar
35  private void copyOldFiles(){
36   try {
37     newJar = new JarOutputStream(new FileOutputStream(tempJar));
38     byte buffer[] = new byte[1024];
39     int bytesRead;
40     try {
41       // Adiciona os arquivos originais
```

```
42        Enumeration entries = jar.entries();
43        while (entries.hasMoreElements()) {
44          JarEntry entry = (JarEntry) entries.nextElement();
45          InputStream is = jar.getInputStream(entry);
46          newJar.putNextEntry(entry);
47                 while ((bytesRead = is.read(buffer)) != -1)
48                  newJar.write(buffer, 0, bytesRead);
49        }
50      } catch (IOException ex) {
51        success = false;
52        System.err.println("Copy: Operation aborted due to : " + ex);
53      }
54    } catch (IOException ex) {
55      success = false;
56      System.err.println("Can't access new file");
57      try {
58        jar.close();
59      } catch (IOException e) {
60        success = false;
61        System.err.println(e.getMessage());
62      }
63    }
64  }
65
66  //adiciona o arquivo passado como parâmeto ao tester.jar
67  public void addNewFile (String fileToAdd) throws FileNotFoundException{
68   byte buffer[] = new byte[1024];
69   int bytesRead;
70   FileInputStream fis = null;
71   // Add new file last
72   try {
73     fis = new FileInputStream(fileToAdd);
74     JarEntry entry = new JarEntry(fileToAdd);
75     newJar.putNextEntry(entry);
76     while ((bytesRead = fis.read(buffer)) != -1)
77       newJar.write(buffer, 0, bytesRead);
78   } catch (IOException ex) {
79     if (ex.getClass().getName().equalsIgnoreCase("java.io.FileNotFoundException")) {
80       throw new FileNotFoundException(fileToAdd);
81     }
82   } finally {
83     try {
84       if (fis != null)
85         fis.close();
86     } catch (IOException e) {
87       success = false;
```

```
88      }
89    }
90   }
91
92   //Finaliza as variáveis e grava o arquivo tester.jar final
93   public void persistTempJar(){
94    try {
95      jar.close();
96      newJar.close();
97    } catch (IOException ignored) {
98    }
99    if (success) {
100     File origFile = new File("..\\dist\\tester.jar");
101     origFile.delete();
102     tempJar.renameTo(origFile);
103   }
104   else tempJar.delete();
105   }
106  }
```

Apêndice B

Estrutura Lógica do XML dos Casos de Teste

Os casos de teste gerados a partir dos diagramas de atividades são inicialmente armazenados em um formato independente de linguagem de programação, aumentando a portabilidade da abordagem. Para isso, foi definido um modelo de dados para armazenamento dos casos de teste em formato XML. O formato foi influenciado por outros esquemas já definidos [21], incluindo o UML Testing Profile [45]. Esta parte da UML é utilizada para a definição dos artefatos de sistemas de teste. Parte da terminologia proposta pela UML foi utilizada no meta modelo proposto.

Este apêndice apresenta a estrutura lógica dos XML correspondentes ao modelo de dados exibido na Figura B.1. A estrutura é registrada na linguagem XSD (*XML Schema Definition*) [80], criada para definição da estrutura e dos tipos de dados de documentos XML. Um documento XSD é composto por um elemento <xs:schema> contendo declarações de atributos e elementos, e definição de novos tipos de dados, que podem ser simples (*simpleType*) ou complexos (*complexType*), isto é, que contém outros atributos e elementos. O documento XSD completo, correspondente ao modelo de dados, é apresentado a seguir.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
3
4   <!-- Class: Argument  -->
5   <xs:element name="Argument" type="Argument"/>
6   <xs:complexType name="Argument">
7    <xs:sequence>
8     <xs:element name="index" type="xs:int"/>
9     <xs:element name="name" type="xs:string"/>
10    <xs:element name="datatype" type="xs:string"/>
```

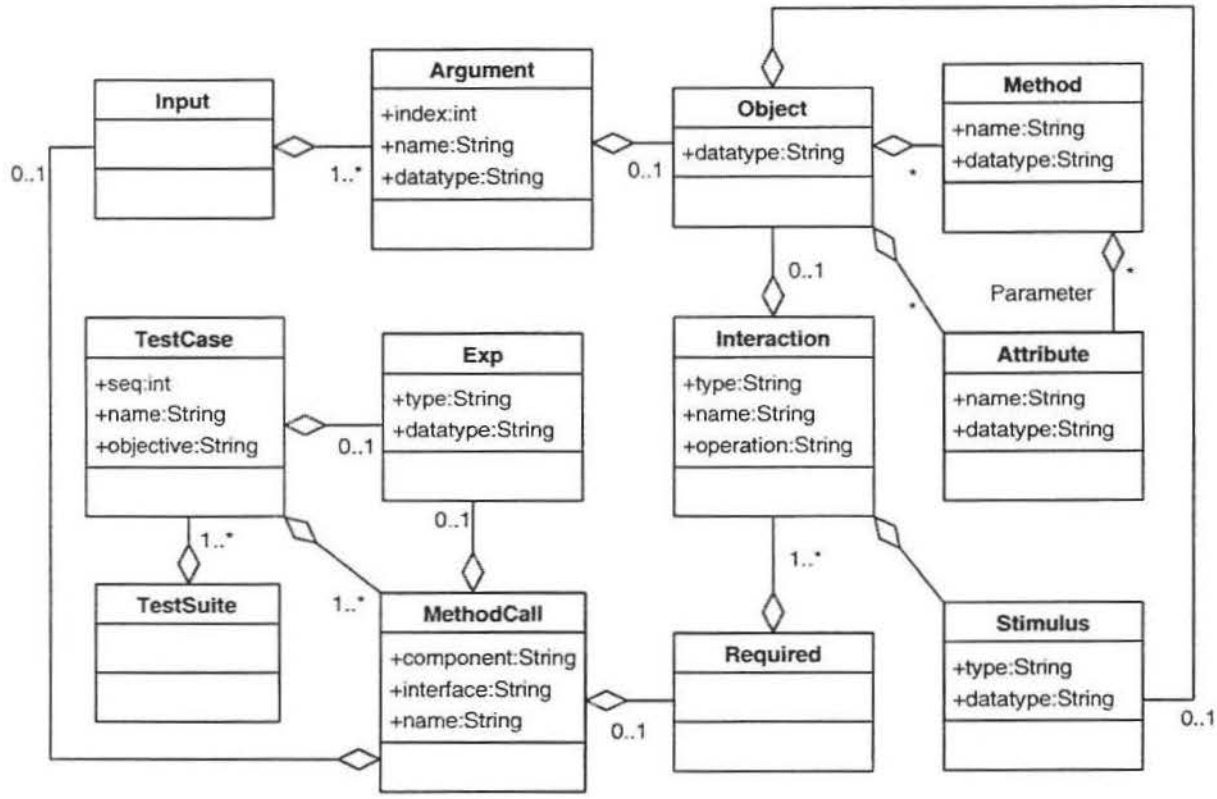

Figura B.1: Modelo de dados para o XML de especificação dos casos de teste.

```
11     <xs:element ref="Object" minOccurs="0"/>
12    </xs:sequence>
13   </xs:complexType>
14
15   <!-- Class: Attribute  -->
16   <xs:element name="Attribute" type="Attribute"/>
17   <xs:complexType name="Attribute">
18    <xs:sequence>
19     <xs:element name="name" type="xs:string"/>
20     <xs:element name="datatype" type="xs:string"/>
21    </xs:sequence>
22   </xs:complexType>
23
24   <!-- Class: Exp  -->
25   <xs:element name="Exp" type="Exp"/>
26   <xs:complexType name="Exp">
27    <xs:sequence>
```

```
28      <xs:element name="type" type="xs:string"/>
29      <xs:element name="datatype" type="xs:string"/>
30      <xs:element ref="Object" minOccurs="0"/>
31     </xs:sequence>
32    </xs:complexType>
33
34    <!-- Class: Input  -->
35    <xs:element name="Input" type="Input"/>
36    <xs:complexType name="Input">
37     <xs:sequence>
38      <xs:element ref="Argument" minOccurs="1" maxOccurs="unbounded"/>
39     </xs:sequence>
40    </xs:complexType>
41
42    <!-- Class: Interaction  -->
43    <xs:element name="Interaction" type="Interaction"/>
44    <xs:complexType name="Interaction">
45     <xs:sequence>
46      <xs:element name="interface" type="xs:string"/>
47      <xs:element name="operation" type="xs:string"/>
48      <xs:element name="type" type="xs:string"/>
49      <xs:element ref="Stimulus"/>
50     </xs:sequence>
51    </xs:complexType>
52
53    <!-- Class: Method  -->
54    <xs:element name="Method" type="Method"/>
55    <xs:complexType name="Method">
56     <xs:sequence>
57      <xs:element name="name" type="xs:string"/>
58      <xs:element name="datatype" type="xs:string"/>
59      <xs:element ref="Attribute" minOccurs="0" maxOccurs="unbounded"/>
60     </xs:sequence>
61    </xs:complexType>
62
63    <!-- Class: MethodCall  -->
64    <xs:element name="MethodCall" type="MethodCall"/>
65    <xs:complexType name="MethodCall">
66     <xs:sequence>
67      <xs:element name="component" type="xs:string"/>
68      <xs:element name="interface" type="xs:string"/>
69      <xs:element name="name" type="xs:string"/>
70      <xs:element ref="Exp"/>
71      <xs:element ref="Required" minOccurs="0"/>
72      <xs:element ref="Input" minOccurs="0"/>
73     </xs:sequence>
```

```
74  </xs:complexType>
75
76  <!-- Class: Object  -->
77  <xs:element name="Object" type="Object"/>
78  <xs:complexType name="Object">
79   <xs:sequence>
80    <xs:element name="datatype" type="xs:string"/>
81    <xs:element ref="Method" minOccurs="0" maxOccurs="unbounded"/>
82    <xs:element ref="Attribute" minOccurs="0" maxOccurs="unbounded"/>
83   </xs:sequence>
84  </xs:complexType>
85
86  <!-- Class: Required  -->
87  <xs:element name="Required" type="Required"/>
88  <xs:complexType name="Required">
89   <xs:sequence>
90    <xs:element name="Interaction" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
91   </xs:sequence>
92  </xs:complexType>
93
94  <!-- Class: Stimulus  -->
95  <xs:element name="Stimulus" type="Stimulus"/>
96  <xs:complexType name="Stimulus">
97   <xs:sequence>
98    <xs:element name="type" type="xs:string"/>
99    <xs:element name="datatype" type="xs:string"/>
100   <xs:element ref="Object"/>
101  </xs:sequence>
102 </xs:complexType>
103
104 <!-- Class: TestCase  -->
105 <xs:element name="TestCase" type="TestCase"/>
106 <xs:complexType name="TestCase">
107  <xs:sequence>
108   <xs:element name="seq" type="xs:int"/>
109   <xs:element name="name" type="xs:string"/>
110   <xs:element name="objective" type="xs:string"/>
111   <xs:element ref="MethodCall" minOccurs="1" maxOccurs="unbounded"/>
112   <xs:element ref="Exp" minOccurs="0"/>
113  </xs:sequence>
114 </xs:complexType>
115
116 <!-- Class: TestSuite  -->
117 <xs:element name="TestSuite" type="TestSuite"/>
118 <xs:complexType name="TestSuite">
119  <xs:sequence>
```

```
120      <xs:element ref="TestCase" minOccurs="1" maxOccurs="unbounded"/>
121     </xs:sequence>
122    </xs:complexType>
123
124    <!-- DataType: void  -->
125    <xs:simpleType name="void">
126     <xs:restriction base="xs:string"/>
127    </xs:simpleType>
128
129  </xs:schema>
```