Uma Estratégia para o Desenvolvimento de Componentes de Software Autotestáveis

Luciano Hayato Ukuma

Dissertação de Mestrado

Instituto de Computação

Universidade Estadual de Campinas

Uma Estratégia para o Desenvolvimento de Componentes de Software Autotestáveis

Luciano Hayato Ukuma[1]

Abril de 2002.

Banca Examinadora

- Profa. Eliane Martins (Orientadora)
- Profa. Selma Shin Shimizu Melnikoff
  Escola Politécnica (USP)
- Profa. Cecília Mary Fischer Rubira
  Instituto de Computação (UNICAMP)
- Profa. Maria Beatriz Felgar de Toledo (Suplente)
  Instituto de Computação (UNICAMP)

[1] Financiado pela CAPES

Uma Estratégia para o Desenvolvimento de Componentes de Software Autotestáveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Luciano Hayato Ukuma e aprovada pela banca examinadora.

Campinas, 08 de Abril de 2002.

Eliane Martins (Orientadora)

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

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Ukuma, Luciano Hayato

Uk9e Uma estratégia para o desenvolvimento de componentes de software autotestáveis / Luciano Hayato Ukuma -- Campinas, [S.P. :s.n.], 2002.

Orientadora : Eliane Martins

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

1. Software – Testes. 2. Engenharia de software. 3. Software - Desenvolvimento. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 08 de abril de 2002, pela Banca Examinadora composta pelos Professores Doutores:

Profª. Drª. Selma Melnikoff
EPUSP

Profª. Drª. Cecília Mary Fischer Rubira
IC - UNICAMP

Profª. Drª. Eliane Martins
IC - UNICAMP

Agradecimentos

Agradeço primeiramente a meus pais, Sadao Ukuma e Yoshiko Ohara Ukuma, por tudo que eles têm feito por mim. Por todo amor que tenho recebido.

Aos meus irmãos, Walter e Tânia, por todo o incentivo e carinho.

Especialmente a Thaise, por todo amor, carinho e apoio.

Aos amigos de república Gerson, Uber, Danillo, Junior e Sandro pelos anos de convivência e amizade.

Agradeço principalmente a minha orientadora, Profa. Eliane Martins, pela dedicação e ensinamentos.

A Deus, por ter me dado forças e saúde para superar os momentos de tristeza e por orientar meu caminho e pensamentos.

Resumo

A atividade de teste de software tem-se mostrado de extrema importância para o desenvolvimento de software com qualidade. No caso do desenvolvimento de software baseado em componentes, tal atividade tem ainda maior importância devido a característica da reusabilidade, pois para que o reuso seja bem sucedido é fundamental que o componente reutilizável seja confiável. Desta forma, componentes devem ser testados durante todo seu desenvolvimento e a cada vez que haja alguma alteração. Neste trabalho é apresentada a descrição de uma estratégia para o desenvolvimento de componentes autotestáveis que tem como objetivo construir componentes que possuam mecanismos de teste embutidos afim de melhorar sua testabilidade. O objetivo final é reduzir os custos com a realização dos testes sem comprometimento da confiabilidade. Mais especificamente nossa estratégia está baseada nos seguintes elementos: especificação do componente, especificação de teste, mecanismos de teste embutido e geração de casos de teste.

Abstract

The software test activity has revealed to be of extreme importance for the development of good quality software. In particular, in the case of the component-based development, such an activity has an even greater importance due to reusability potential. However, for the reuse to be successful it is fundamental that the reusable component be reliable. Thereby, components should be tested during it development time and every time that the component undergoes some alteration. This work presents the description of a strategy for the development of self-testing components that aims to construct components that have built-in test mechanisms to improve the testability in order to reduce the costs with test realization, without compromising reliability. More specifically, our strategy is based on: component specification, test specification, built-in test mechanisms and test case generation.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Contexto e Motivação

Atualmente existe muita pesquisa sobre arquiteturas de software, padrões de projeto, *frameworks* e desenvolvimento de software baseado em componentes (do inglês *Component-Based Software Engineering*, CBSE). Todas estas técnicas têm como principal objetivo permitir a reutilização de soluções já existentes. A reutilização de componentes tem sido uma das abordagens mais promissoras exatamente pelos benefícios que ela proporciona como a redução de custos e prazos para o desenvolvimento de software.

Todavia para que o reuso seja bem sucedido é fundamental que se possa ter confiança de que o componente reutilizável forneça o serviço desejado. Por exemplo em 1996, durante o lançamento do veículo espacial Ariane5, houve uma mudança na trajetória e o veículo explodiu menos de um minuto após o lançamento. Depois do acidente foram realizadas investigações que determinaram que a explosão resultara de uma insuficiência de testes sobre um software reutilizado da Ariane4. Desenvolvedores teriam certamente reutilizado componentes de software da Ariane4 no projeto da Ariane5 sem retestá-los substancialmente, assumindo que não existiam diferenças significativas entre os dois sistemas [49] [15].

Segundo [49], componentes reutilizáveis devem ser testados várias vezes: durante o desenvolvimento, toda a vez que forem utilizados em um novo contexto e a cada vez que sofram alguma alteração. Portanto, melhorar a testabilidade de um componente reutilizável é essencial para alcançar alta qualidade. Inicialmente as abordagens para a melhoria da testabilidade foram desenvolvidas na área de hardware com o objetivo de introduzir mecanismos para melhorar o acesso a pontos internos do circuito durante os testes, além de poder-se evitar a necessidade de utilizar um testador externo através da incorporação de mecanismos para a geração de testes e análise de resultados.

Em software, [23] [7] propuseram trabalhos que mostraram que também é possível embutir mecanismos de testes para melhorar a testabilidade de componentes de software. Isto foi realizado fazendo uma analogia entre circuitos integrados e software. Por exemplo em [7] foi proposto o conceito de classes autotestáveis, em que da mesma forma que no hardware, em que circuitos extras foram embutidos exclusivamente com funções de teste, mecanismos similares também foram acrescentados a uma classe. Com base nesse trabalho foi desenvolvido um protótipo de uma ferramenta de apoio à construção e uso de componentes autotestáveis constituídos de uma classe escrita em C++ [43].

Neste trabalho busca-se estender o conceito de autoteste para componentes de software. A extensão desta técnica se mostrou interessante porque apesar de um componente de software poder conter uma única classe, este, na maioria das vezes, é composto por um conjunto de classes relacionadas. Desta forma, com a aplicação desta estratégia poderá se construir componentes reutilizáveis que sejam fáceis de testar afim de reduzir os custos com a realização dos testes .

A definição de componentes de software utilizada neste trabalho é especificada como sendo uma unidade de composição com interfaces contratualmente especificadas e com dependências de contexto explícitas [41]. Portanto, é importante salientar que um componente deve possuir uma interface bem definida, apresentando todas as funcionalidades que este pode fornecer. Além disso, apesar de se considerar que um componente não necessariamente utiliza-se de uma tecnologia específica [12], neste trabalho serão considerados apenas componentes oriundos da tecnologia de orientação a objetos.

1.2 Objetivos

Os principais objetivos desta dissertação são:

1. Estudar e propor uma estratégia para a construção de componentes de software autotestáveis, tendo como base a técnica desenvolvida por [43].

2. Analisar as dificuldades e vantagens da utilização da estratégia proposta.

1.3 Organização da Dissertação

Os outros capítulos desta dissertação estão organizados da seguinte forma:

Capítulo 2 apresenta os conceitos de teste de software, tanto para sistemas convencionais quanto para sistemas orientados a objetos. São descritos os objetivos da atividade de teste, as fases, a classificação dos testes e processo envolvido nesta atividade. Também são apresentados os conceitos de orientação objetos e o impacto das características deste paradigma sobre a atividade de teste.

Capítulo 3 apresenta uma visão geral da abordagem de desenvolvimento baseado em componentes. São apresentados os principais conceitos, características, benefícios e uma classificação das fases de teste para esta abordagem.

Capítulo 4 descreve as principais características da testabilidade de um software, apresentando os fatores que afetam a testabilidade, algumas métricas de testabilidade, características do projeto para testabilidade aplicadas ao software e alguns trabalhos que abordam problemas relacionados com a melhoria da testabilidade.

Capítulo 5 apresenta a estratégia proposta para a construção de componentes de software autotestáveis, sendo descritos os passos que devem ser executados para a sua aplicação.

Capítulo 6 apresenta dois estudos de caso realizados para ilustrar a estratégia proposta e avaliar as dificuldades e vantagens de sua utilização.

Capítulo 7 apresenta as conclusões dessa dissertação, as principais contribuições e os planos para a continuidade do trabalho.

Capítulo 2

Teste de Software

Um dos principais critérios na medição da qualidade de sistemas de software é a verificação de sua funcionalidade. Embora o conceito de qualidade não esteja definido somente com base neste fator, raramente um sistema será considerado de boa qualidade quando este apresentar falhas na realização de suas funções.

Desta forma, a confiabilidade do sistema somente será atingida e comprovada quando técnicas de teste de software forem aplicadas ao sistema oferecendo evidências de que este atende as funcionalidades desejadas.

Neste capítulo são apresentados conceitos de teste de software, tanto para sistemas convencionais quanto para sistemas orientados a objetos, que foram importantes para o desenvolvimento e compreensão deste trabalho, sendo que o conceito de testabilidade de software será tratado no Capítulo 4.

Na Seção 2.1 é apresentada uma terminologia que é aplicada durante todo o trabalho. Na Seção 2.2 são apresentados os objetivos da atividade de teste. Na Seção 2.3 são descritas as fases do teste de software. Na Seção 2.4 é apresentado o processo de teste. Na Seção 2.5 é apresentada a classificação dos testes segundo os critérios de seleção. Na Seção 2.6 são apresentados os conceitos de orientação objetos e o impacto das características deste paradigma sobre a atividade de teste. Esta seção apresenta ainda como estão classificadas as fases de teste para este tipo de sistema.

2.1 Terminologia

Primeiramente é necessário definir alguns termos que serão usados durante o texto, em particular os termos falha, erro e defeito. As definições usadas neste trabalho seguem o

padrão IEEE (IEEE STD. *Glossary of Software Engineering Terminology*, padrão 610.12/1990) e em português, seguiu-se a terminologia definida em [28].

A manifestação de um engano realizado por um desenvolvedor é uma falha (*fault*) no software (uma especificação ou código incorretos). Quando uma falha é ativada (por exemplo, a instrução com falha é executada), é gerado um erro, o qual pode se propagar e levar a um defeito (*failure*). Diz-se que um software tem um defeito quando o serviço que ele fornece a outros sistemas ou ao usuário viola a sua especificação.

2.2 Objetivos

Testes constituem uma atividade de verificação dinâmica que consiste em exercitar o software fornecendo-lhe entradas com o objetivo de encontrar falhas [32], e não provar que o software está correto.

Portanto, um teste bem sucedido é aquele que detecta uma falha que ainda não havia sido descoberta. Caso a execução dos testes não detecte nenhuma falha, isso não significa que o software não contenha falhas.

Em geral, não é possível testar exaustivamente um programa, ou seja, fornecer-lhe todas as entradas possíveis. Portanto, é imprescindível escolher um conjunto finito de entradas que tenham boa capacidade de detectar falhas. Na seção 2.5 são apresentados alguns critérios de seleção que são utilizados para estabelecer conjuntos de casos de teste.

A realização de testes é importante para complementar outras formas de verificação e validação pois é a única que permite exercitar o comportamento operacional do programa [30].

2.3 Fases de Teste

A atividade de teste é composta de diversas fases, cada uma com o objetivo de validar diferentes aspectos do software. Portanto, os casos de teste para cada fase são especificados em uma determinada etapa do desenvolvimento de software. Para sistemas convencionais são consideradas as fases de teste apresentadas a seguir:

Teste de Unidade

Antes de tratar dos aspectos referentes a essa fase de teste, é útil definir o conceito de unidade. Segundo o padrão IEEE 610.12-1990, uma unidade é um componente de

software que não pode ser dividido. Portanto, uma unidade é desenvolvida para ser parte de um programa maior, não constituindo um programa completo por si mesma.

O teste de unidade tem como objetivo verificar cada unidade que compõe o software isoladamente, para determinar se cada uma delas realiza o que foi especificado. Para que a unidade possa ser testada é necessário que se construa um programa que permita a execução e teste da unidade. Para cada unidade sendo testada, um *driver* e um ou vários *stubs* podem ser necessários.

O *driver* é um elemento que coordena o teste da unidade, sendo responsável por receber os dados de teste fornecidos pelo testador, passar esses dados na forma de parâmetros para a unidade, coletar os resultados relevantes produzidos pela unidade, e apresentá-los para o testador.

Um *stub* é um elemento que substitui, na hora do teste, uma função usada pela unidade. Desta forma, um *stub* simula o comportamento de outra unidade necessária, com o mínimo de computação ou manipulação de dados. O uso conjunto do *driver* e *stubs* torna possível o teste da unidade de maneira independente, e isolada do restante do programa. A figura 1 apresenta o relacionamento entre o *driver*, a unidade de teste e os *stubs*.

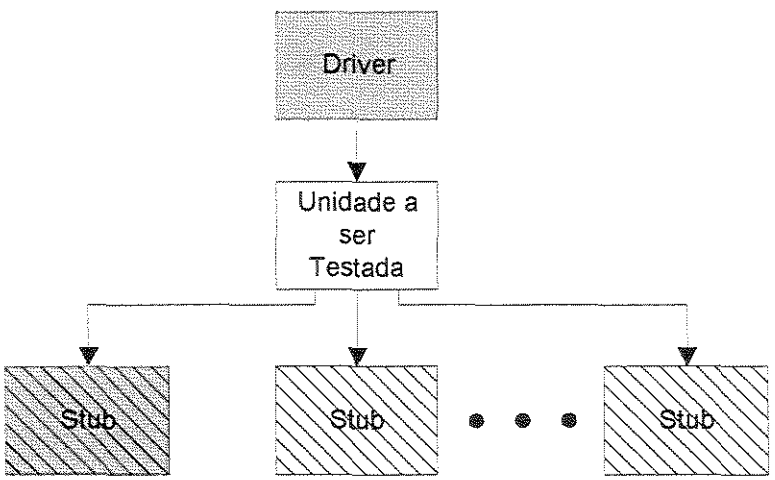

Figura 1 – Relacionamento entre *Driver* – Unidade de Teste – *Stubs*

Teste de Integração

Nesta fase a ênfase está nos testes das interfaces das unidades e suas interações. Conjuntos de teste devem ser construídos com o objetivo explícito de exercitar as interações e não a funcionalidade individual das unidades.

Pode-se perguntar porque os testes de integração são necessários, quando cada unidade já foi testada isoladamente. O problema está justamente no interfaceamento entre elas.

Várias estratégias têm sido propostas para a integração de um software. Estratégias não incrementais, também conhecidas como *Big-bang*, combinam todos os módulos do software para então aplicar o teste. Já estratégias incrementais constróem o software por partes. Cada parte é testada, integrada com outras partes, e novamente testadas, até que o sistema completo seja construído. As principais estratégias incrementais são a *Top-Down*, a *Bottom-Up* e *Sandwich*.

Na primeira, a estratégia exige a utilização de *stubs* que devem substituir temporariamente os módulos de mais baixo nível enquanto estes não são integrados. A vantagem dessa estratégia é que os pontos principais de decisão ou controle, que em geral encontram-se nos módulos de mais alto nível, são testados antes, fazendo com que possíveis problemas sejam revelados mais cedo.

A estratégia *Bottom-Up*, ao contrário, inicia a integração pelos módulos mais baixos, neste caso *drivers* são necessários para coordenar o teste de cada subsistema construído. A estratégia *Sandwich* combina as duas anteriores. O programa é integrado de baixo para cima e de cima para baixo e eventualmente a integração se encontra em algum ponto no meio da estrutura.

Teste de Validação

O objetivo do teste de validação é determinar se o software funciona da maneira esperada, conforme consta na especificação de requisitos. Um aspecto não menos importante desse teste diz respeito à revisão da configuração, cujo objetivo é determinar se todos os elementos da configuração do software foram devidamente desenvolvidos e documentados, afim de que a manutenção possa ser realizada posteriormente.

Teste de Sistemas

Consiste na realização de vários tipos de testes que visam determinar se os componentes de um sistema computacional (envolvendo outros componentes de software e/ou hardware) se integram bem e realizam as funcionalidades que lhes foram especificadas.

Portanto o teste de sistemas tem como objetivo validar comportamentos que são expostos somente testando o sistema completo, incluindo hardware e software. Testes de

sistemas visam também validar algumas propriedades do sistema, tais como desempenho e segurança [30].

Em geral sugere-se que as fases de teste sejam realizadas concorrentemente com as fases de desenvolvimento, como ilustrado na figura 2.

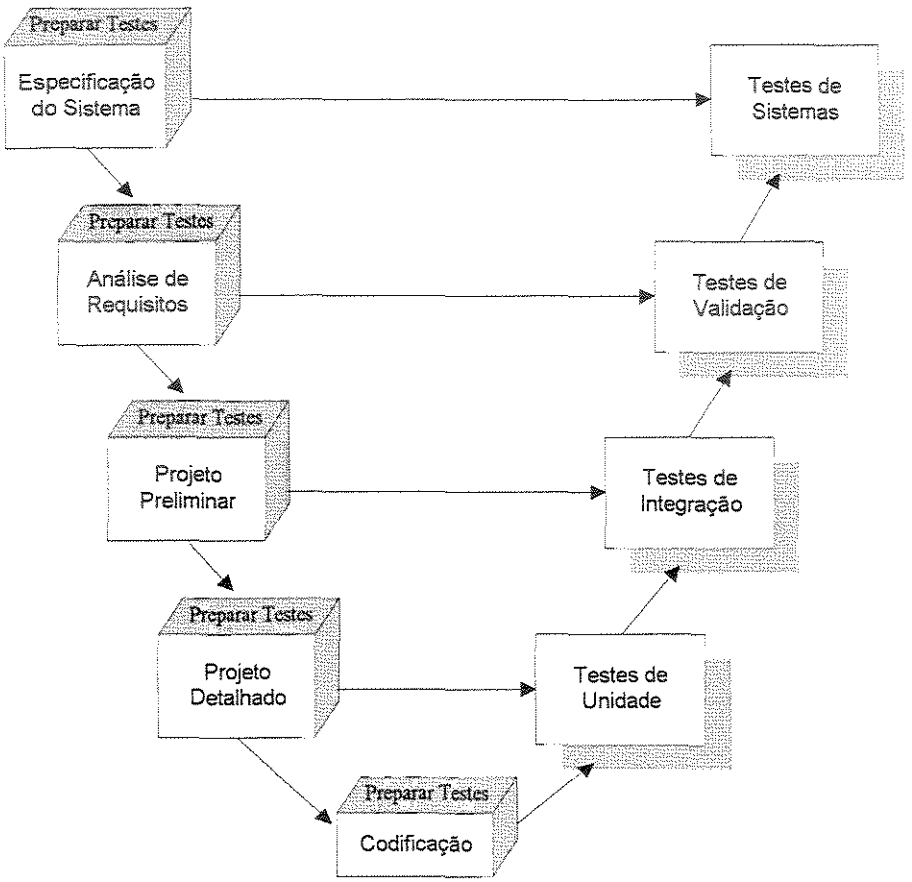

Figura 2 – Fases de Teste no Processo de Desenvolvimento

Em suma, a atividade de teste inicia com os testes de unidade, que visam verificar se cada módulo ou unidade satisfaz à sua especificação, estabelecida no Projeto Detalhado. Após testar separadamente cada módulo, estes são agrupados para compor subsistemas, de acordo a arquitetura do sistema definida no Projeto Preliminar, sendo esta a fase de testes de integração.

Em seguida, são realizados os testes de validação afim de determinar se o software satisfaz aos requisitos especificados na fase de análise de requisitos e finalmente os testes de sistema são aplicados para exercitar o sistema como um todo, incorporando todos os

componentes (hardware e software) para determinar se o sistema completo satisfaz a sua especificação.

2.4 O Processo de Teste

Para cada fase de teste deve-se aplicar um processo englobando desde a geração dos testes até análise dos resultados e a correção do sistema. Esse processo, como ilustrado na figura 3, é constituído por uma seqüência de atividades [35].

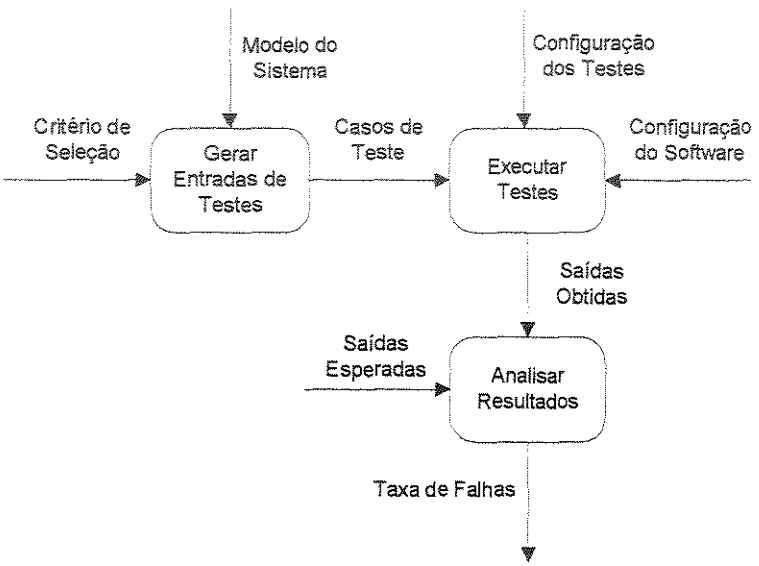

Figura 3 – Processo de Teste

O processo inicia com a geração e seleção de casos de testes, formados por dados de teste e pelas saídas esperadas. Os casos de teste são derivados a partir de um modelo de base, que pode ser uma especificação, o próprio código ou uma representação das falhas na especificação ou no código, decorrentes de erros cometidos ao longo do desenvolvimento. Um problema que está associado a esta atividade é a dificuldade de se selecionar um conjunto de testes finito, pois geralmente o conjunto de entradas possíveis para um programa é muito grande ou mesmo infinito. Este problema será tratado com mais detalhes nas seções seguintes.

A próxima atividade é a execução dos testes. Para isso deve ser construído um ambiente que permita executar o software com os casos de teste selecionados e coletar as saídas observadas. Um problema relacionado a esta atividade é a quantidade de casos de

testes que devem ser executados, pois freqüentemente o número de testes é muito grande, sendo imprescindível a automação dos testes, afim de facilitar e melhorar a execução dos casos de testes, pois aplicá-los manualmente além de não ser prático é altamente propenso a erros.

Após a execução temos a análise dos resultados que compreende tanto a avaliação da qualidade do software quanto da qualidade dos testes aplicados. Nesta etapa é necessária a aplicação de um oráculo. Um oráculo é um mecanismo que determina se uma saída obtida é correta ou não para uma dada entrada. O problema é que muitos programas são construídos para calcular resultados que não são possíveis de se obter manualmente e portanto, torna-se difícil determinar se os resultados estão corretos. O fato é que oráculos são necessários, todavia oráculos automatizados são raros e oráculos humanos estão propensos a erros.

2.5 Classificação dos Testes

Uma das grandes dificuldades nos testes é selecionar um conjunto de entradas com a maior probabilidade de detectar o maior número possível de falhas [32]. Os critérios de seleção de testes tentam solucionar este problema estabelecendo condições que devem ser satisfeitas durante os testes.

Como mencionado na seção anterior, os casos de testes são derivados a partir de um modelo de base. Esses modelos muitas vezes não são testáveis, desta forma deve-se desenvolver uma abstração dos mesmos que facilite descobrir quais são as possíveis falhas e quais os casos de testes que podem ser aplicados para descobrí-las. Esse modelo intermediário entre o modelo de base e os testes é denominado modelo de teste [39]. De acordo com esses modelos, os critérios de seleção podem ser classificados como: critério funcional, critério estrutural e critério baseado em falhas. As próximas seções apresentam mais detalhadamente cada um desses critérios.

2.5.1 Critério Funcional

Também conhecido como caixa preta pois o código em teste é tratado como uma caixa da qual não se conhece o conteúdo, levando-se em consideração somente as entradas que o

programa aceita e as saídas que o programa deve produzir, sem considerar detalhes de implementação.

O objetivo desse critério é determinar se todos os requisitos foram implementados, onde um requisito não implementado corresponde a um caminho ou código faltando no software [35]. As fontes usadas por este critério são as especificações de requisitos e de projeto. Alguns exemplos que podem ser citados são:

Particionamento em Classes de Equivalência

Procura dividir o domínio de entrada em classes que têm como característica o fato de que a execução do programa com qualquer elemento de uma classe produz o mesmo resultado. Assim se um caso de teste revela uma falha, todos os elementos da mesma classe devem também revelar a mesma falha.

Análise de Valor Limite

Acrescenta algumas características ao critério de Particionamento em Classes de Equivalência. Em vez de selecionar qualquer elemento de uma classe, os casos de teste são escolhidos nas fronteiras das classes pois é aí que se concentra um grande número de erros.

Grafo de Causa-Efeito

Nesse critério são primeiramente levantadas as possíveis condições de entrada (causas) e as possíveis ações (efeitos) do programa. Em seguida, é construído um grafo relacionando as causas e efeitos levantados. Esse grafo é convertido em uma tabela de decisão que é então utilizada na construção dos casos de teste.

Teste de Transição de Estados

Nesse critério os testes são baseados em modelos baseados em transição de estados. Esses modelos são úteis para representar aspectos dinâmicos (que variam com o tempo) de um sistema. O objetivo do teste de transição de estados é exercitar todas as transições existentes ao menos uma vez. Existem alguns métodos para a derivação dos testes, como por exemplo o método de varredura de transições (também chamado de método T ou TT, do inglês *Transiction Tour*) que tem o objetivo de percorrer o modelo baseado em transições de estados de forma a satisfazer o critério dado.

2.5.2 Critério Estrutural

Utiliza informações obtidas a partir do código do programa para a derivação dos testes, assim, este critério também é chamado de teste caixa branca ou caixa de vidro significando que a estrutura da implementação é considerada para a escolha dos testes. Alguns exemplos que podem ser citados são [3] [32]:

Critérios baseados na estrutura

Visam exercitar os diferentes componentes estruturais do programa. Basicamente são três os componentes estruturais visados: instruções (comandos), decisões e dados.

Critérios baseados no fluxo

O objetivo desses critérios é exercitar as diferentes seqüências de execução de comandos ou de definição-uso de dados (As ocorrências de variáveis em um programa podem ser classificadas em: "definição – def" quando um valor é atribuído à variável e "uso" quando a referência à variável não a está definindo). Estes critérios consideram os componentes estruturais do programa no seu contexto de execução, ou seja, eles consideram seqüências de comandos ou de definição/uso de dados. Este critério é dividido em critérios baseados no fluxo de controle e critérios baseados no fluxo de dados.

Critérios baseados no estado

Visam exercitar os diferentes estados do programa. O estado de um programa é determinado pelos valores de suas variáveis em um momento específico da execução. Este critério tem por objetivo fazer com que nos locais onde eventualmente existam falhas o estado dos dados seja infectado com valores incorretos.

2.5.3 Critério baseado em falhas

Utiliza informações sobre a classe de falhas que são cometidas no processo de desenvolvimento para a seleção de testes. Duas abordagens comumente utilizadas são baseadas na Análise de Mutantes e Injeção de Falhas.

Análise de Mutantes

A análise de mutantes consiste em introduzir falhas no programa, criando assim os chamados mutantes [16]. Os mutantes são produzidos aplicando-se ao programa operadores

de mutação, os quais introduzem pequenas falhas no programa, uma por vez, como mostra o exemplo a seguir.

P: Programa Original	**Mutante 1:**	**Mutante 2:**	
...	...	...	...
if a <= b then	if a < b then	if a > b then	
...	...	...	

Os mutantes são executados e espera-se que eles se comportem de forma diferente do programa original. Quando isso acontece é dito que o mutante foi morto pelo caso de teste. Um conjunto T de casos de teste é dito adequado a um programa P e a um conjunto M de mutantes se para cada mutante $m \in M$ que não é equivalente a P existir um caso de teste $t \in T$ tal que t mata m com relação a P.

Se o conjunto M de mutantes modelasse todas as falhas possíveis em um programa P, um conjunto T de casos de teste adequado para M poderia ser utilizado para provar que o programa não possui falhas e, conseqüentemente, provar que P está correto. No entanto, o conjunto de mutantes seria infinito e a adequação de T não poderia ser avaliada. Portanto, para tornar factível a tarefa de avaliação da adequação, M deve ser finito, escolhendo-se apenas alguns tipos de falhas. A adequação de um conjunto de testes T é dado pelo escore de mutação definido como:

$$m(T,P) = |K(T,P)| \ / \ |M(P)| - |E(P)|$$

onde,

K(T,P) - conjunto de mutantes mortos pelos casos de teste de T.

M(P) - conjunto de mutantes de P.

E(P) - conjunto de mutantes equivalentes.

Injeção de Falhas

Esta técnica tem como objetivo acelerar a ocorrência de falhas em um sistema, ou seja, gerar falhas dentro do sistema sob teste afim de observar seu comportamento. As falhas que afetam um sistema podem ser classificadas segundo diferentes pontos de vista, de forma geral, pode-se considerar uma divisão em falhas de hardware e falhas de software.

As falhas de hardware representam as falhas decorrentes de fenômenos físicos adversos. As falhas de software representam as falhas resultantes de imperfeições cometidas durante as fases de desenvolvimento do sistema ou nas modificações feitas em fase de manutenção. As principais modalidades de injeção de falhas são [29]:

- Injeção de falhas por hardware: Utiliza-se um circuito especial para produzir as falhas no sistema sob teste.

- Injeção de falhas por software: Tem como objetivo modificar o estado do hardware e/ou software usando-se ferramentas de software.

- Injeção de falhas por simulação: Essa forma de injeção é realizada ainda na etapa de projeto do sistema sob teste, em oposição às formas anteriores, que são feitas durante o teste do sistema. Nesta técnica utiliza-se o projeto do sistema para uma simulação do que aconteceria caso ocorresse uma falha. Através disso, pode-se determinar se o projeto do sistema é adequado para tratar das possíveis falhas que podem ocorrer.

Essas diferentes técnicas de teste possuem características complementares e por isso devem ser utilizadas em conjunto, buscando aumentar a qualidade do sistema.

2.6 Testes em Sistemas Orientados a Objetos

O paradigma de orientação a objetos apresenta-se como uma abordagem promissora para o desenvolvimento de software robusto e confiável devido a características pertencentes ao próprio paradigma, tais como, abstração de dados, encapsulamento, herança e reutilização de objetos [37].

A utilização desse paradigma facilita o controle da complexidade do sistema porque promove uma melhor estruturação de seus componentes e consequentemente o desenvolvimento de sistemas bem projetados, todavia isso não assegura a produção de sistemas corretos [8]. Portanto é essencial testar sistemas orientados a objetos, principalmente considerando o fato de que este paradigma favorece a reutilização de componentes de software em contextos diferentes, o que implica que tais componentes devem ser altamente confiáveis e desta forma devem ser bem testados.

Atualmente existe um número considerável de técnicas de teste procedimental que foram desenvolvidas, reconhecidas e implementadas através de ferramentas. Entretanto, tais técnicas necessitam de adaptações quando sistemas orientados a objetos são considerados. Isto ocorre porque apesar de serem importantes e poderosas, as características deste

paradigma introduzem novos desafios para se testar esse tipo de sistema, como será visto a seguir.

2.6.1 Conceitos e Características que Afetam os Testes em Sistemas Orientados a Objetos

Nesta seção serão apresentados os aspectos de orientação a objetos que influenciam na realização de testes. Desta forma, não será feita uma apresentação detalhada de todos os conceitos do assunto, pois fogem do escopo desse texto. Serão apresentados somente aqueles conceitos que tenham algum interesse do ponto de vista dos testes.

A seguir temos uma breve definição de alguns conceitos de orientação a objetos e sua influência nos testes. Os conceitos são baseados nas referências de [35] [37] e as dificuldades levantadas para os testes em [8] e [30].

Encapsulamento

O encapsulamento descreve o empacotamento de uma coleção de itens (ocultação da informação), permitindo que detalhes internos da implementação sejam escondidos do mundo externo. Os dados estão inseridos no objeto e somente os métodos do objeto têm acesso a eles, assim um objeto pode ter sua implementação alterada sem que isso afete as aplicações que o utilizam. Do ponto de vista dos testes, o encapsulamento dificulta o controle e observação do estado interno de um objeto, pois os detalhes internos da implementação de atributos e métodos são escondidos do mundo externo.

Herança

É o mecanismo que permite derivar novas classes a partir de classes existentes através de um processo de refinamento. O maior benefício da herança é a reutilização de software, pois cada classe derivada herda todas as propriedades e operações de sua classe base, e pode adicionar novas operações, estender ou redefinir a implementação de operações já existentes. O mecanismo de herança pode ser usado para formar hierarquias de implementação e hierarquia de tipos. Na hierarquia de implementação a herança é usada somente para facilitar a implementação, não sendo garantido que a subclasse tenha o mesmo comportamento que a superclasse. Na hierarquia de tipos (ou herança de comportamento), a subclasse é vista como um subtipo da superclasse e apresenta no

mínimo o mesmo comportamento que esta. Neste caso, pode-se dizer que há um relacionamento do tipo "é-um" entre a subclasse e a superclasse. Em relação aos testes, a herança apresenta as seguintes dificuldades:

- A linguagem utilizada afeta a estratégia a ser adotada para os testes, pois a implementação do mecanismo de herança varia de uma linguagem para outra.

- As características herdadas devem ser testadas a cada reutilização, devido a possíveis mudanças no novo contexto.

- Se a herança não for usada de forma disciplinada (por exemplo, herança de implementação), os casos de teste precisam ser regerados para a subclasse.

Polimorfismo e Ligação Dinâmica

Polimorfismo significa a capacidade de tornar mais de uma forma. No contexto de orientação a objetos um atributo pode ter mais de um conjunto de valores e uma operação pode ser implementada por mais de um método. Assim, dois ou mais objetos podem realizar de formas diferentes a mesma operação. A ligação dinâmica é uma forma de implementar o polimorfismo. Este mecanismo permite a determinação em tempo de execução de qual operação será executada dependendo do nome da operação e do tipo do objeto. Em relação aos testes o polimorfismo e a ligação dinâmica apresentam as seguintes dificuldades:

- A determinação de todos os usos polimórficos de um objeto é impossível.

- É difícil antecipar todas as possibilidades de ligações e estas aumentam a possibilidade de ocorrência de falhas de interfaces.

A tabela a seguir apresenta um resumo das principais dificuldades existentes para a realização de testes em sistemas orientados a objetos, apresentando além das dificuldades descritas anteriormente, outras como a necessidade de ferramentas de teste especializadas para este tipo de sistema.

Característica	Dificuldade
Encapsulamento	Dificulta o controle e observação do estado interno de um objeto, pois os detalhes internos da implementação de atributos e métodos são escondidos do mundo externo.
Herança	A implementação do mecanismo de herança varia de uma linguagem para outra, assim a estratégia a ser adotada para os testes está intimamente associada a linguagem.
	Mudanças em uma ou mais classes bases podem causar efeitos inesperados nas classes derivadas, tendo como implicação a necessidade de retestá-las.
	Na herança de implementação a classe derivada não precisa herdar todas as características da classe base. Assim os casos de teste já definidos para a classe base devem ser alterados para suprimir as referências às propriedades não herdadas.
Polimorfismo e Ligação Dinâmica	É difícil antecipar todas as possibilidades de ligações e estas aumentam a possibilidade de ocorrência de falhas de interfaces.
	A determinação de todos os usos polimórficos de um objeto é impossível.
	Existem classes não instanciáveis, ou seja, classes abstratas (que contêm métodos virtuais) e classes parametrizadas, que portanto não podem ser testadas diretamente, pois só objetos podem ser testados.
Reuso	O reuso de software implica na necessidade de re-teste, pois mesmo que as características herdadas permaneçam inalteradas, elas devem ser testadas no novo contexto.
Relacionamentos	É difícil entender uma classe se ela depende de muitas outras classes.
	O testador pode encontrar problemas na identificação de onde iniciar os testes em uma biblioteca.
	A construção de *stubs* se torna mais complexa.
Problema de Ferramenta de Suporte	A maior parte das ferramentas comerciais existentes implementam métodos de teste convencionais.

Tabela 1 – Características que Afetam o Teste de Sistemas Orientado a Objetos

Estes problemas introduzidos pelas características do paradigma orientado a objetos não são triviais. Por outro lado existem características que facilitam os teste em relação aos sistemas procedimentais, tais como [30]:

- As interfaces de classes e métodos são bem definidas e explícitas, isto reduz o risco de que alterações em características de uma classe afetem outras classes, desde que a interface permaneça inalterada.

- Os métodos são em geral pequenos, com algoritmos menos complexos devido à alta coesão existente dentro de uma classe, reduzindo a probabilidade de ocorrência de falhas de caminhos de execução.

- A herança sugere uma forma natural de reutilizar casos de teste, por exemplo a técnica incremental hierárquica [22] é uma abordagem de teste que explora a natureza hierárquica da relação de herança para testar grupos de classes

relacionadas, reutilizando a informação de teste de uma classes base para orientar o teste de uma classe derivada.

2.6.2 Fases de Teste

Assim como nos testes convencionais, a atividade de teste em sistemas orientado a objetos também é composta de diversas fases, podendo se classificar em [35]:

Teste de Classe

O teste de classes é equivalente ao teste de unidades em sistemas convencionais, e enfocam os métodos encapsulados na classe e o seu comportamento. Apesar de que os métodos não podem ser tratados separadamente da classe a que pertencem, algumas abordagens sugerem o teste de cada método isoladamente antes de se testar a classe.

Teste Intraclasse e Interclasse

Estes testes são equivalentes ao teste de integração em sistemas convencionais. As estratégias incrementais não são aplicáveis a sistemas orientados a objetos, pois estes não apresentam uma estrutura de controle hierárquica como os sistemas procedimentais. O teste intraclasse tem o objetivo de testar as interações entre os métodos pertencentes a uma classe, enquanto que o teste interclasse refere-se aos testes sobre um grupo de classes que interagem. O teste interclasse é classificado como teste de *cluster* (um *cluster* é um conjunto de classes que interagem entre si) e são apresentadas duas abordagens para o teste de integração:

- Testes de *threads* - visam integrar as classes necessárias para responder uma determinada entrada ou a um determinado evento do sistema. Cada *thread* vai sendo integrada e testada.

- Testes de usos - começam a integração do sistema a partir das classes independentes, isto é, aquelas que não se relacionam com outras classes. Em seguida são integradas as classes que se relacionam diretamente com essas classes independentes, e prossegue-se dessa forma até que todas as classes do sistema tenham sido integradas.

Teste do Sistema

A funcionalidade é enfatizada neste nível. Nessa fase, da mesma forma que nos testes convencionais, a estrutura interna do sistema não é mais de interesse: o enfoque está nas respostas do sistema às entradas fornecidas por seus usuários. Neste caso não haveria grandes diferenças com relação aos testes de sistemas convencionais.

Quanto ao processo de teste, apesar das diferenças significativas entre os dois tipos de sistemas, as atividades que devem ser realizadas para testar um sistema são as mesmas, ou seja, deve-se gerar os casos de teste, executá-los e por fim analisar os resultados. Os testes baseados na especificação são muito úteis nesse contexto. Cenários de uso e outros modelos construídos na análise e/ou projeto são usados nos testes, com algumas modificações, pois geralmente estes são muito informais para permitir a derivação semi-automática dos testes.

Os testes de transição de estados, em particular, são muito usados tanto nos testes de classes quanto nos testes de integração e de sistemas. Por exemplo em [9] é apresentada uma técnica que pode ser usada nos testes de classes, de integração e de sistemas, onde o modelo de estados é utilizado para representar o comportamento de classes para as quais seja possível determinar seqüências válidas de invocação de métodos.

2.7 Considerações Finais

Neste capítulo foram apresentados os principais conceitos relacionados à atividade de teste de software, tanto para sistemas convencionais quanto para sistemas orientados a objetos, que são importantes para a compreensão do trabalho. Entre os vários conceitos apresentados neste capítulo é importante destacar que o objetivo da atividade de teste é detectar falhas no software e não provar que este está correto, ou seja, um teste bem sucedido é aquele que detecta uma falha que ainda não havia sido descoberta, sendo que, caso a execução dos testes não detecte nenhuma falha, isso não significa que o software não contenha falhas. No próximo capítulo serão apresentados os principais conceitos e características de componentes de software, pois o objetivo deste trabalho é definir uma estratégia de teste para facilitar e melhorar a realização de testes para este tipo de artefato.

Capítulo 3

Componentes de Software

A crescente complexidade do software e a constante preocupação com o desenvolvimento de produtos de qualidade atendendo aos requisitos básicos de engenharia de software podem ser considerados como fatores chave para a pesquisa por novas tecnologias que possam ser aplicadas no desenvolvimento de software.

Entre os principais requisitos de qualidade segundo [35] estão a reusabilidade, manutenabilidade, portabilidade, confiabilidade, usabilidade e interoperabilidade. Dentre essas, a reusabilidade representa uma alternativa significativa para incrementar a produtividade e qualidade no desenvolvimento e manutenção de software.

Neste contexto, várias pesquisas na área de orientação a objetos, padrões de projeto, *frameworks* e componentes de software estão sendo desenvolvidas. Todas essas abordagens visam aumentar a produtividade no processo de desenvolvimento de software através da reutilização de soluções já existentes. Neste capítulo é apresenta uma visão geral da abordagem de desenvolvimento baseado em componentes (*Component-Based Software Engineering*, CBSE). O desenvolvimento baseado em componentes permite que uma aplicação seja construída através da reutilização de componentes de software que já foram bem especificados e testados, diminuindo os custos do processo de desenvolvimento [13].

Na seção 3.1 são apresentados os principais conceitos e características dessa abordagem. A seções seguintes apresentam mais detalhes sobre interfaces e contratos (Seção 3.2), componentes de software e orientação a objetos (Seção 3.3), processo de desenvolvimento (Seção 3.4), benefícios da CBSE (Seção 3.5) e características da atividade de teste para esta abordagem (Seção 3.6). Na seção 3.7 temos um sumário do capítulo.

3.1 Conceitos e Características

A abordagem de desenvolvimento baseado em componentes tem sido vista como uma tecnologia promissora para o desenvolvimento de software. Segundo [13] o estilo das aplicações têm sofrido algumas alterações.

As aplicações que eram baseadas em *mainframes* e com redes de comunicação proprietárias passaram a ser distribuídas e acessadas por *intranets* e pela *Internet*. As organizações que trabalhavam com poucos mas grandes projetos começaram a gerenciar um número maior de projetos menores. Essas mudanças geraram a necessidade de se reutilizar artefatos na construção desses projetos.

Todavia, apesar da atual euforia em relação ao desenvolvimento baseado em componentes, existem algumas questões que ainda são muito discutidas como a definição de componentes de software e de um processo de desenvolvimento [12]. Por exemplo, algumas definições existentes para componentes de software na literatura são:

"Um componente é uma parte não trivial, quase independente e substituível de um sistema que executa uma função específica no contexto de uma arquitetura bem definida. Um componente está de acordo e provê a implementação de um conjunto de interfaces" [13]

"Um pacote de artefatos de software que podem ser desenvolvidos independentemente e disponibilizados como uma unidade e que podem ser compostos com outros componentes para construir um sistema complexo" [18]

Apesar dessas definições descreverem o mesmo conceito, existem algumas diferenças entre elas, por exemplo a primeira enfatiza a existência de interfaces sendo que o mesmo não ocorre na segunda. Neste trabalho, a definição utilizada para um componente de software é a seguinte:

"Um componente de software é uma unidade de composição com interfaces contratualmente especificadas e com dependências de contexto explícitas. Um componente de software pode ser desenvolvido independentemente e está sujeito a composição de terceiros" [41]

Para que um artefato possa se tornar um componente de software não é necessária a utilização de uma tecnologia específica desde que este forneça uma interface bem definida [12]. Esses componentes podem ser construídos por pessoas diferentes e em contextos diferentes. Além disso, de acordo com [2] componentes de software podem ser visualizados em duas perspectivas: componentes como implementação e componentes como uma abstração arquitetural.

O conceito de componentes como implementação é freqüentemente usado para referenciar-se a componentes de prateleira ou COTS (do inglês, *Components Off-The-Shelf*). Os COTS podem implementar funcionalidades (o que o componente faz) e coordenação (como um componente interage com o mundo externo) que são específicas para o produto.

Vistos como abstrações arquiteturais, componentes de software são referenciados como componentes arquiteturais. Estes componentes são obrigados a implementar uma ou mais interfaces que determinam como componentes podem interagir ou outras restrições arquiteturais [2]. Estas restrições são estabelecidas por infra-estruturas conhecidas como modelos de componentes ou padrões para componentes.

Atualmente, os principais modelos de componentes existentes são: OMG CORBA (*Common Object Request Broker Arquitecture*) [33], Microsoft COM/DCOM [31] e Sun JavaBeans/ EJB (*Enterprise Java Beans*) [25]. Essas infra-estruturas provêm um conjunto de serviços comuns de gerenciamento de componentes, definindo algumas regras/restrições no projeto e implementação de componentes de software.

A especificação de todas as funcionalidades de um componente de software pode ser representada através de suas interfaces. Uma interface é uma coleção de operações que são utilizadas para especificar um serviço de um componente [11]. Uma interface resume como um cliente deve interagir com o componente, escondendo detalhes de implementação, fornecendo ao cliente todas as informações das quais depende para utilizar o componente, permitindo que o projetista apenas se preocupe com detalhes semânticos da interface e não com detalhes de implementação do componente como um todo [40].

Com esse conceito, o desenvolvimento baseado em componente diferencia-se de outras técnicas devido a separação entre a especificação (interface) e a implementação de um componente de software, e também pelo fato de um componente poder implementar várias interfaces, sendo que a mesma interface pode ser implementada por diversos componentes [2].

A figura 4 apresenta uma abstração de um componente de software como uma caixa preta onde suas interações ocorrem através de sua interface.

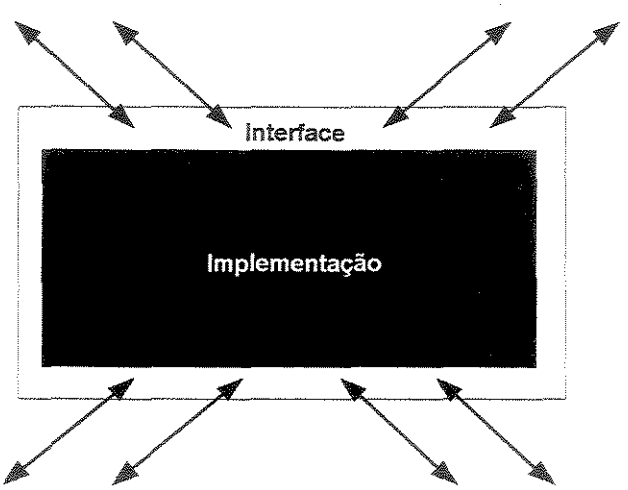

Figura 4 – Ilustração que representa um componente como uma caixa preta

A figura 5 ilustra um componente de software e suas interfaces, bem como as suas dependências. O componente descrito realiza as operações contidas em suas interfaces (Interface A e Interface B) e depende dos serviços descritos na Interface C. Essa representação foi modelada utilizando a linguagem UML.

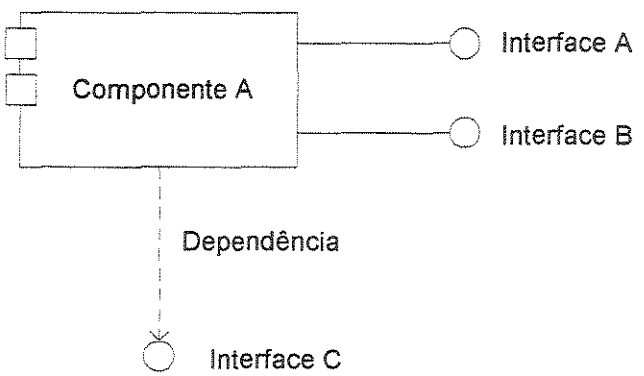

Figura 5 – Componentes e Interfaces

A separação da especificação em diferentes interfaces facilita a manutenção do sistema, pois um componente de software pode ser atualizado ou substituído por outro,

desde que este último apresente as mesmas interfaces do componente original, sem grandes impactos sobre os clientes do componente.

Além disso, a utilização de interfaces conduz a construção de um sistema baseado em componentes para um projeto baseado em interfaces. Desta forma, modelos de componentes como CORBA e COM estão diretamente associados com linguagens de definição de interface (do inglês *Interface Definition Language*, IDL). Assim como as especificações das infra-estruturas baseadas em Java, como EJB, enfatizam o projeto baseado em interfaces.

3.2 Interfaces e Contratos

Como descrito anteriormente a interface de um componente define um conjunto de operações que podem ser usadas por outros componentes, ou seja, a interação entre componentes é realizada através das operações de suas interfaces, sendo que, para que esta interação seja realizada é necessário que ambas as partes (cliente e componente) respeitem algumas regras.

Desta forma, a especificação de uma interface deve incluir, além da lista de operações que a interface oferece (incluindo suas assinaturas), todas as condições que devem ser respeitadas entre o cliente e o componente que irá implementar esta interface, ou seja, deve apresentar uma parte "requer" (*requires*) e uma parte "fornece" (*provides*). Por exemplo, um componente como o da figura 6 que *fornece* uma operação que calcula a soma de dois números inteiros positivos (CalculaSomaInteiroPos) *requer* que o cliente forneça dados que estejam dentro do escopo para que a operação seja realizada.

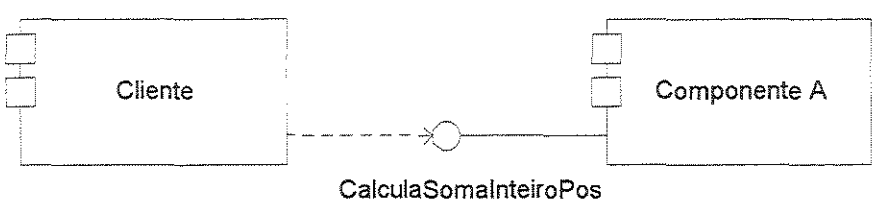

Figura 6 – Exemplo

Uma maneira de descrever uma interface é através de um contrato. Um contrato especifica os serviços fornecidos pelo componente, as responsabilidades do cliente e o ambiente necessário para que o componente ofereça estes serviços [2]. Contratos podem ser

descritos utilizando pré e pós condições [14]. Uma pré-condição especifica as condições que devem ser verdadeiras para a execução de uma operação da interface do componente e uma pós-condição especifica as condições que devem ser verdadeiras no momento em que a operação terminar sua execução, mais precisamente antes de que a mensagem/valor resultante seja retornado para o cliente. No caso do exemplo da figura anterior pode-se definir as seguintes pré e pós condições (figura 7):

```
PROCEDURE CalculaSomaInterfacePos (x,y:INTEGER; OUT result:INTEGER)
        Precondition: (x>=0) & (y>=0)
        Poscondition: result = x +y
```

Figura 7 – Contratos especificados como pré e pós condições

3.3 Componentes de Software e Orientação a Objetos

Alguns pesquisadores vêem o desenvolvimento baseado em componentes como uma evolução da abordagem de orientação a objetos, onde um componente corresponde a um conjunto de classes inter-relacionadas com visibilidade externa limitada pois, similarmente aos objetos, os componentes de software têm uma interface externa que fornece operações que manipulam estados internos [26].

Muitos princípios fundamentais do desenvolvimento baseado em componentes são os mesmos que formam a base do paradigma de orientação a objetos, como a unificação de dados e funções (operações) e encapsulamento, onde o cliente de um objeto não tem conhecimento de como as operações são implementadas.

Segundo [11], componentes de software assemelham-se em muitos aspectos com classes em orientação a objetos. Ambos possuem nomes, podem implementar um conjunto de interfaces, podem participar em relacionamentos de dependência de associação, podem ser aninhados, possuem instâncias e podem participar em interações. Todavia, apresentam algumas diferenças significativas:

- Classes representam abstrações lógicas, componentes de software representam elementos físicos que existem na forma de *bits*.

- Componentes representam um agrupamento físico, ou seja, um componente é uma implementação física de um conjunto de elementos lógicos, como classes.

- Classes podem possuir atributos e operações diretamente. Em geral, componentes de software possuem operações que são alcançáveis somente através de suas interfaces.

A figura 8 mostra o relacionamento entre um componente de software (Componente A) e as classes que ele implementa. O relacionamento está modelado em UML através de uma relação de dependência.

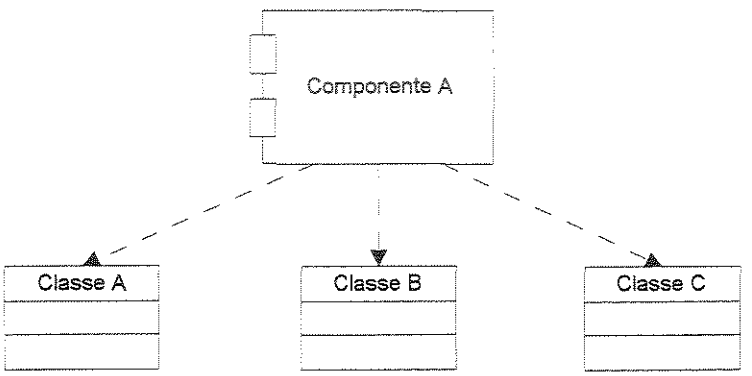

Figura 8 – Componente e Classes

Um componente de software existe, em tempo de execução, através de suas instâncias. A figura 9 mostra uma instância do componente A, essa instância é responsável pela criação e controle das instâncias (objetos) das classes que o componente A implementa.

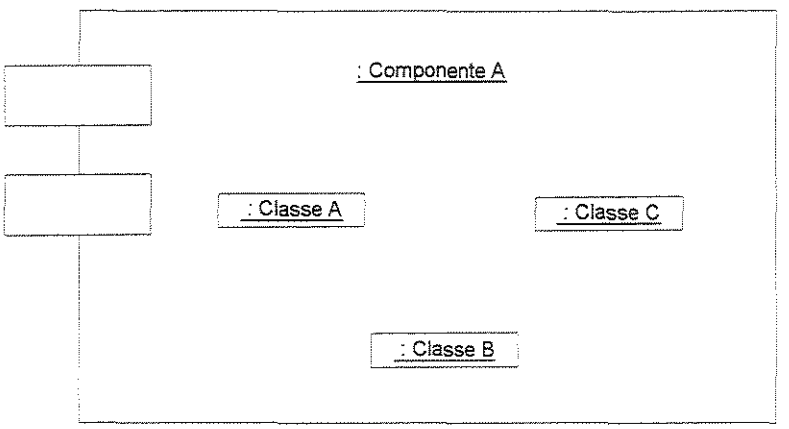

Figura 9 – Instância de um Componente

3.4 Processo de Desenvolvimento

Para que a utilização de componentes de software se torne uma prática consolidada é necessário o desenvolvimento de novos métodos de desenvolvimento e gerenciamento de componentes de software. Em [13] sugere-se que para construir componentes de software deve-se aplicar as tarefas de seleção, testes e adaptação de componentes.

A etapa de seleção visa a pesquisa por componentes apropriados ao projeto. Ao término da etapa de seleção deve-se aplicar testes afim de verificar se o componente realmente realiza as funcionalidades descritas. Geralmente os componentes selecionados necessitam de algumas modificações para se adequarem integralmente aos requisitos descritos no projeto da aplicação. Neste caso deve-se aplicar técnicas de adaptação de componentes. Esta etapa também é importante para a fase de manutenção, onde novos requisitos podem ser adicionados ocasionando a necessidade de adaptar os componentes inseridos na aplicação.

Além disso, para que seja possível reutilizar componentes de software também se faz necessário a existência de catálogos de componentes, através do qual desenvolvedores possam realizar pesquisas à componentes que atendam aos requisitos do projeto. Assim, a documentação de um componente se torna um elemento imprescindível para que esta seleção seja realizada. A figura 10 ilustra a seqüência de atividades realizadas durante um processo de desenvolvimento de software, envolvendo seleção, testes e adaptação de componentes.

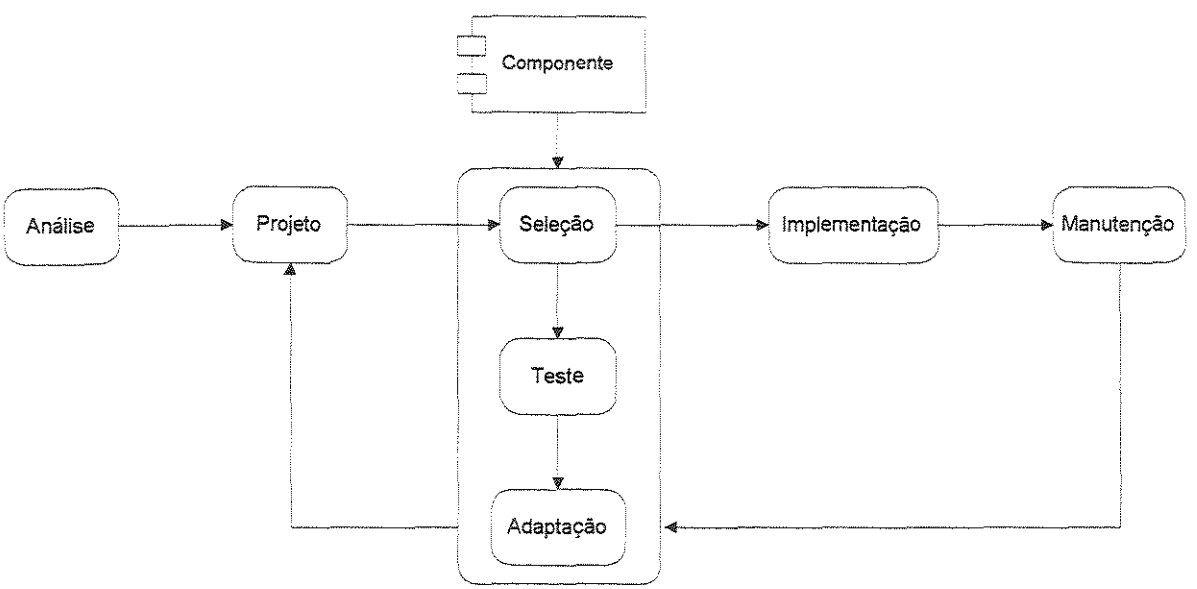

Figura 10 – Seqüência de Atividades para o CBSE

De acordo com esta figura, durante a transição entre o projeto e a implementação, deve-se realizar a identificação dos componentes que respondem aos requisitos especificados no projeto. A fase de implementação torna-se um processo de composição e integração de componentes, além de envolver a construção dos componentes de projeto que não foram selecionados [50].

3.5 Benefícios

De acordo com alguns autores [13] [41], o desenvolvimento de software baseado em componentes apresenta os seguintes benefícios:

- Redução no tempo de desenvolvimento – criar uma aplicação através de componentes já existentes, mesmo que seja necessário desenvolver a algumas novas características, torna-se mais rápido do que desenvolver uma aplicação completa. Além disso, uma aplicação desenvolvida a partir de componentes de software tem maior flexibilidade para se adequar as mudanças estabelecidas pelo ambiente na qual será utilizada.

- Aumento na qualidade – a construção de software através da reutilização de componentes de software que já foram bem especificados e testados, proporciona o aumento da produtividade e da qualidade dos produtos.

- Melhoria na manutenção do software – no desenvolvimento baseado em componentes a especificação é separada da implementação. Para isso o componente necessita encapsular sua implementação e interagir com o seu ambiente externo (outros componentes) somente através de sua interface [3]. Com essa separação, diminui-se o impacto e as dificuldades das modificações no sistema, visto que um componente de software pode ser substituído por outro desde que este especifique as mesmas interfaces do componente original.

- Redução dos custos – a medida que as aplicações se tornam mais seguras e desenvolvidas em menos tempo implicará na redução de custos.

- Alternativa de desenvolvimento – melhor do que adquirir uma solução completa ou um pacote que talvez não seja o desejado, uma organização pode comprar apenas os componentes desejados, combinando-os em uma solução própria. Dessa forma, componentes podem ser escolhidos de acordo com critérios como desempenho, preço e confiabilidade. Além disso, alguns componentes podem ser produzidos para cumprir determinados requisitos.

- Desenvolvimento simultâneo – As tarefas de desenvolvimento e seleção de componentes podem ser realizadas por equipes diferentes. Desta forma, enquanto uma equipe preocupa-se com o desenvolvimento de alguns componentes a outra responsabiliza-se pela aquisição de componentes existentes.

3.6 Testes de Componentes de Software

Na seção anterior foram apresentados os principais benefícios da reutilização de componentes de software. Todavia para que o reuso seja bem sucedido é fundamental que o componente reutilizável seja confiável.

Segundo [49] componentes reutilizáveis devem ser testados várias vezes: durante o desenvolvimento, toda a vez que forem utilizados em um novo contexto e a cada vez que sofram alguma alteração. Por exemplo, como mencionado anteriormente, o problema com o veículo espacial Ariane 5 ocorreu devido a reutilização de componentes de software da Ariane 4 no projeto da Ariane 5 sem retestá-los substancialmente, assumindo que não existiam diferenças significativas entre os dois sistemas.

Algumas atividades que podem facilitar a realização de testes em componentes reutilizáveis e consequentemente melhorar a qualidade do software consiste em armazenar alguns artefatos [21] [49]:

- Especificação do componente, incluindo informações sobre as modificações realizadas durante as atualizações do software.

- Casos de teste, incluindo informações entre cada caso de teste e as funcionalidades que este deve exercitar. Estas informações auxiliam a determinar quais casos de teste devem ser reaplicados (teste de regressão) quando o software é modificado.

Além disso, com estas informações é possível determinar quais partes da especificação tem sido testadas.

Em relação a classificação das fases de teste, a atividade de teste para a abordagem de desenvolvimento baseado em componentes pode ser classificada nas seguintes fases [49]:

- *Teste de Componente* – Similar ao teste de classes para sistemas orientados a objetos, esta etapa tem como objetivo testar cada componente separadamente. Para isso, é necessário que se construa um programa que permita a execução e teste deste componente. Para cada componente sendo testado, um driver e um ou vários stubs podem ser necessários. O uso conjunto do driver e desses stubs torna possível o teste do componente de maneira independente, isolada do restante do programa. Uma diferença básica entre o teste de classes e o teste de componente é a de que um componente geralmente é composto por mais do que uma classe.

- *Teste de Integração de Componentes* – Semelhante ao teste de cluster para sistemas orientados a objetos, todavia ao invés de verificar-se a interação entre um conjunto de classes, verifica-se a interação entre componentes. Segundo [49] existem dois fatores que afetam a complexidade da integração dos componentes. O primeiro é o número de componentes envolvidos e o segundo é capacidade de customização dos componentes.

- *Teste de Sistemas* – O objetivo deste teste é determinar se todas as funcionalidades que foram especificadas são realizadas de forma esperada.

Apesar da classificação não incluir a fase de teste de classes esta também deve ser aplicada durante a fase de teste do componente afim de validar as funcionalidade das classes que constituem o componente.

3.7 Considerações Finais

Este Capítulo apresenta as principais características do desenvolvimento de software baseado em componentes, onde constatou-se que componentes de software assemelham-se

em muitos aspectos com classes em orientação a objetos. Ambos possuem nomes, podem implementar um conjunto de interfaces, podem participar em relacionamentos de dependência de associação, podem ser aninhados, possuem instâncias e podem participar em interações. Todavia, apresentam algumas diferenças significativas como por exemplo classes representam abstrações lógicas, enquanto que, componentes de software representam elementos físicos que existem na forma de *bits*.

Também verificou-se que componentes de software devem ser testados várias vezes: durante o desenvolvimento, toda a vez que forem utilizados em um novo contexto e a cada vez que sofram alguma alteração. Dessa forma, é essencial que tais componentes sejam fáceis de testar para que os custos com a realização dos testes seja reduzida. Segundo [10] o projeto de componentes reutilizáveis melhora quando a testabilidade é considerada durante o processo de desenvolvimento. Assim, o próximo Capítulo irá apresentar os principais conceitos e características relacionados com a testabilidade de software, para que estes possam ser aplicados no projeto de componentes de software.

Capítulo 4

Testabilidade de Software

No capítulo anterior foram apresentados os conceitos e benefícios do desenvolvimento de software baseado componentes, podendo-se destacar a importância da realização de testes para que o reuso seja bem sucedido.

Portanto é essencial que componentes reutilizáveis sejam testáveis, ou seja, construir um software visando ser mais testável ou visando a sua testabilidade pode ser uma boa alternativa para se diminuir os custos com a realização dos testes, pois segundo [35] normalmente uma organização chega a gastar 40% do esforço de desenvolvimento para testar um software.

Dessa forma, é importante verificar as características que devem ser analisadas para melhorar da testabilidade de componentes de software. Entre as principais questões existentes em relação a testabilidade estão:

- O que é testabilidade?
- Quais são os fatores que afetam a testabilidade de um componente de software?
- Como construir componentes de software testáveis?

Este capítulo tem como objetivo discutir estas questões, assim na Seção 4.1 são abordados os principais conceitos sobre testabilidade. Nas seções seguintes são apresentados os fatores que afetam a testabilidade (Seção 4.2), algumas medidas de testabilidade (Seção 4.3), características do projeto para testabilidade aplicadas ao software (Seção 4.4) e alguns trabalhos que abordam problemas relacionados a melhoria da testabilidade (Seção 4.5). A Seção 4.6 apresenta um sumário do capítulo.

4.1 Definições

A crescente importância da testabilidade como fator da qualidade de um software se justifica porque quanto maior a testabilidade do sistema, menores serão as dificuldades para a realização do processo de teste. Segundo o padrão IEEE 610-12/90, a testabilidade pode ser definida como:

- Em que medida um sistema (ou um componente) facilita o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos;

- Em que medida um requisito é representado de forma a permitir o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos.

Portanto, a testabilidade de um software pode ser definida através de quão fácil é satisfazer uma meta particular de teste através de critérios de testes estabelecidos, como por exemplo, exercitar todas as instruções do programa.

Além da definição tradicional, [45] apresenta uma outra definição para testabilidade que difere por não se preocupar em verificar a facilidade com que algum critério de seleção de entrada pode ser satisfeito durante os testes, mas verificar a probabilidade que um programa possa produzir saídas incorretas caso existam falhas. Desta forma, a testabilidade é definida como sendo a possibilidade de ocorrer um defeito caso existam falhas, ou seja, um componente com baixa testabilidade tende a produzir saídas corretas para a maioria das entradas que executam uma falha.

Segundo [19] um software com boa testabilidade deve possuir duas características: observabilidade e controlabilidade. A observabilidade indica a facilidade de observar um software em relação aos valores de entrada e saída, ou seja, refere-se a capacidade de se determinar o quanto as entradas fornecidas afetam as saídas obtidas. A controlabilidade indica a facilidade de controlar um software em relação aos valores de entrada e saída, ou seja, refere-se a capacidade de se produzir uma saída desejada a partir de uma entrada fornecida.

Assim, para se testar um componente deve ser possível controlar (controlabilidade) suas entradas, bem como observar (observabilidade) a sua saída. Além dessas

características existem outras que também contribuem para a melhoria da testabilidade. A próxima seção apresenta estas características.

4.2 Fatores que afetam a Testabilidade

Segundo [7], uma boa testabilidade é resultado de uma boa prática de engenharia e um processo de desenvolvimento de software bem definido. Para se construir componentes de software visando testabilidade, alguns aspectos devem ser considerados pois afetam diretamente a melhoria da testabilidade, sendo eles [21] [7]:

Representação

Refere-se a qualidade das informações fornecidas para um software, ou seja, quais informações são fornecidas e como estas são apresentadas. A especificação do sistema, seja em linguagem natural ou especificação formal, é essencial para a realização dos testes. É praticamente impossível testar um software sem uma representação pois não se pode determinar o comportamento desejado para validar os testes realizados. Se o software possui requisitos bem definidos e uma documentação atualizada do projeto, isto contribui para a testabilidade de software. Entre os tipos de informações que podem ser apresentadas estão as informações destinadas para os usuários (manual de uso, manual de referência e a especificação da interface do componente) e as informações relacionadas com os desenvolvedores do componente (especificações de análise e projeto, documentação de testes e manutenção). Outros documentos importantes para um bom entendimento do software são: código fonte e elementos de suporte como manuais de instalação e relatórios de qualidade.

Capacidade de teste embutido (do inglês, *Built-in Test* - BIT)

A capacidade de teste embutido fornece uma separação explícita entre os testes e a funcionalidade da aplicação através da inserção de mecanismos de teste no componente. Os mecanismos de teste que podem ser adicionados são métodos *set/reset*, assertivas e métodos relatores [7]. A adição de métodos *set/reset* possibilita colocar um sistema em um estado pré-definido, não importando seu estado corrente. Uma assertiva é um teste sobre todo ou parte do estado de um programa em execução, dado pelo valor de suas variáveis. Assertivas são úteis para assegurar que uma variável tem o valor correto ou dentro de um intervalo em algum ponto da execução. Quando não são satisfeitas podem indicar que as computações realizadas previamente e das quais o valor da variável é dependente, podem

estar incorretas. Métodos relatores observam e armazenam o estado interno de um programa durante a execução dos testes. Desta forma, estes mecanismos podem contribuir para aumentar a testabilidade através da adição da capacidade de observação embutida no componente em teste.

Implementação

A estrutura do sistema é um aspecto importante para a testabilidade do software. Por exemplo, um módulo que contenha funcionalidades da interface e da aplicação dificulta o teste de somente uma delas, ou seja, módulos com funcionalidades bem específicas facilitam a realização dos testes pois facilitam o controle do escopo dos testes, podendo-se isolar e detectar mais rapidamente as falhas.

Seqüência de testes

A existência de uma seqüência de teste é muito importante. Uma seqüência é composta de uma coleção de casos de testes e a estratégia usada para aplicá-los. Além disso, a existência de um oráculo e uma boa documentação dos testes são características que facilitarão a aplicação dos testes.

Reutilização dos casos de teste

Considerando a evolução dos componentes de software deve-se considerar a possibilidade de reutilizar também os casos de teste, pois caso ocorra alguma mudança no software, os desenvolvedores podem aproveitar os casos de teste existentes, facilitando uma possível aplicação de testes de regressão. O teste de regressão tem como objetivo verificar se as modificações introduzidas no software não produziram comportamentos indesejados e novas falhas [35], sendo úteis durante o desenvolvimento (testes de integração) e principalmente na fase de manutenção. A chave para isto é desenvolver métodos sistemáticos e ferramentas para gerenciar e armazenar os casos de teste e as informações geradas pelos testes.

Ferramentas de teste

A não utilização de ferramentas automatizadas dificulta a realização dos testes, pois menos testes serão realizados e portanto, maiores serão os gastos para alcançar a confiabilidade requerida, diminuindo assim a testabilidade.

Processo de desenvolvimento de software

O processo em que a construção do software é conduzido tem influência significativa na sua testabilidade. Um processo bem definido que integre o processo de teste é importante, ou seja, os testes devem ser utilizados de forma balanceada com outras práticas para garantir a qualidade do produto.

Analisando estas características, pode-se concluir que um componente testável deve apresentar uma boa documentação, incluindo planos e casos de teste, e também mecanismos para melhorar e facilitar o controle dos valores de entrada e saída do componente, como por exemplo mecanismos BIT. Além disso um processo bem definido e a existência de ferramentas automatizadas de apoio ao desenvolvimento de software são importantes.

4.3 Medidas de Testabilidade

A realização de testes em grandes sistemas é uma tarefa considerada consumidora de tempo e de recursos no processo de desenvolvimento de software [35]. Portanto, é importante que se possa identificar os pontos do sistema que estão mais propensos a falhas, podendo-se concentrar os testes nestas regiões.

A utilização de métricas de software tem sido considerada como boa indicadora dos pontos mais críticos e complexos de um sistema. Além disso, através do uso de métricas de software, pode-se avaliar o grau de testabilidade de um programa. Nesta seção serão apresentados brevemente algumas métricas de testabilidade existentes na literatura.

Os trabalhos apresentados por Jeffrey Voas estabelecem algumas medidas matemáticas para estimar a testabilidade do software com base na capacidade do software esconder falhas. Isto é realizado através da análise da sensibilidade. Esta análise estima a probabilidade de um defeito acontecer no programa através da execução de uma falha. Segundo [44] um defeito pode ser separado em três eventos: execução de uma falha, criação de um erro e propagação do erro para um saída observável. Estas três fases são referenciadas como PIE (*Propagation, Infection and Execution*). Assim, a técnica de análise de sensibilidade estima a probabilidade destes eventos ocorrerem, estando divididas em: análise de execução, análise de infecção e análise de propagação.

A análise de execução é um método baseado na estrutura do programa, que serve para estimar a probabilidade da execução de uma região do programa. Isto é realizado

através da análise do número de casos de teste que executam esta determinada região. A análise de infecção estima a probabilidade da execução de uma região do programa gerar um estado incorreto para um dado (erro). Esta análise é feita através da criação de uma série de mutantes para uma determinada região do programa. A análise de propagação estima a probabilidade de um erro se propagar até alguma saída, sendo que a infecção é simulada através da mudança do estado de um dado durante a execução do programa e comparando o resultado obtido com o do programa original. Desta forma, quanto maior o número de saídas diferentes do programa original, maior é o grau de sensibilidade do programa. Portanto, a baixa sensibilidade tende a mostrar maior potencial para esconder falhas e consequentemente indicar que mais testes deverão ser realizados.

Em [7] são citados um conjunto de métricas para sistemas orientados a objetos que fornecem informações úteis que auxiliam no projeto de testes, como por exemplo métricas que analisam a herança existente nas classes (NOC – Número de classes derivadas), métricas que analisam o encapsulamento (LCOM – falta de coesão dos métodos) e métricas que analisam o polimorfismo (DYN – percentual de invocações dinâmicas).

No padrão IEEE 1061 1992 são apresentadas algumas métricas para a melhoria da qualidade do software [34]. Entre as métricas citadas para a testabilidade pode-se destacar a cobertura de requisitos funcionais, cobertura de instruções e cobertura de ramos. A aplicação destas métricas melhora a testabilidade e portanto aumenta a qualidade do software.

4.4 Projeto Visando a Testabilidade Aplicado ao Software

Inicialmente as abordagens de projeto para a melhoria da testabilidade foram desenvolvidas na área de hardware com o objetivo de introduzir mecanismos para melhorar o acesso a pontos internos do circuito durante os testes, além de poder-se evitar a necessidade de utilizar um testador externo através da incorporação de mecanismos para a geração de testes e análise de resultados. Em software, [23] e [7] propuseram trabalhos que mostraram que também é possível embutir mecanismos de testes para melhorar a testabilidade de componentes de software. Isto foi realizado fazendo uma analogia entre circuitos integrados e software.

Em [7] foi proposta a aplicação do projeto visando a testabilidade (do inglês *Design For Testability*, DFT) para sistemas orientados a objetos, em que da mesma forma que no hardware, em que circuitos extras foram embutidos exclusivamente com funções de teste,

mecanismos similares também foram acrescentados a uma classe. O projeto visando a testabilidade é uma estratégia para organizar o processo de desenvolvimento de software de modo a maximizar a testabilidade do software produzido.

Robert Binder propõe algumas abordagens de DFT para a construção de classes. Entre as mais importantes pode-se destacar a abordagem BIST (*Built-in Self Test*). Nesta abordagem são adicionadas à classe sob teste funções padrões BIT (*Built-in Test*) de modo a criar capacidades de observação e controle, tais como funções para relatar o estado interno da classe. Desta forma, com base nos conceitos utilizados nesta abordagem surgiu o conceito de classe autotestável.

A figura 11 descreve os componentes necessários para a inserção de mecanismos BIST:

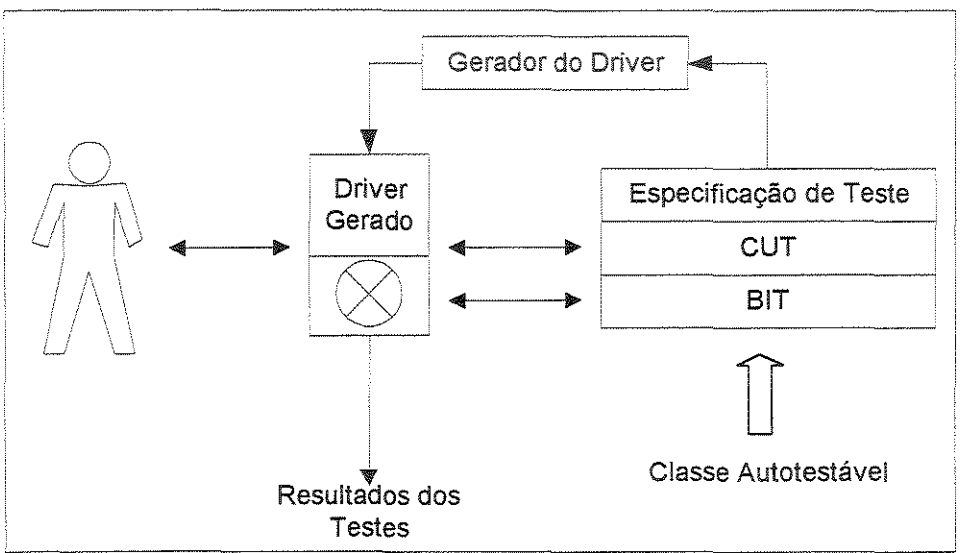

Figura 11 – Abordagem de DFT BIST para classes

- Classe autotestável – Artefato composto dos seguintes elementos: (a) classe sob teste (do inglês *class under test*, CUT), (b) especificação de teste e (c) mecanismos BIT (métodos relatores e assertivas)

- Gerador do driver – Responsável pela geração do driver (específico) que ativará e controlará a CUT durante a execução dos testes.

- Driver gerado (específico) – Corresponde à seqüência executável, gerada de acordo com o critério de teste implementado pelo gerador do driver, aplicado para um modelo de teste representado pela especificação de teste. Além da ativação e controle dos teste, este driver é responsável pela avaliação dos testes.

Baseado no conceito de classe autotestável, [43] propôs uma estratégia para a construção e utilização de classes autotestáveis combinando o conceito de autoteste com a técnica incremental hierárquica (descrita no Capítulo 2) com o objetivo de melhorar a testabilidade de classes, permitindo a reutilização de casos de teste sempre que possível. Além disso, foi construído um protótipo de uma ferramenta – ConCAT (de Construção de Classes AutoTestáveis) – que implementa parte da estratégia proposta. A figura 12 apresenta uma visão geral da metodologia de teste usando a ConCAT.

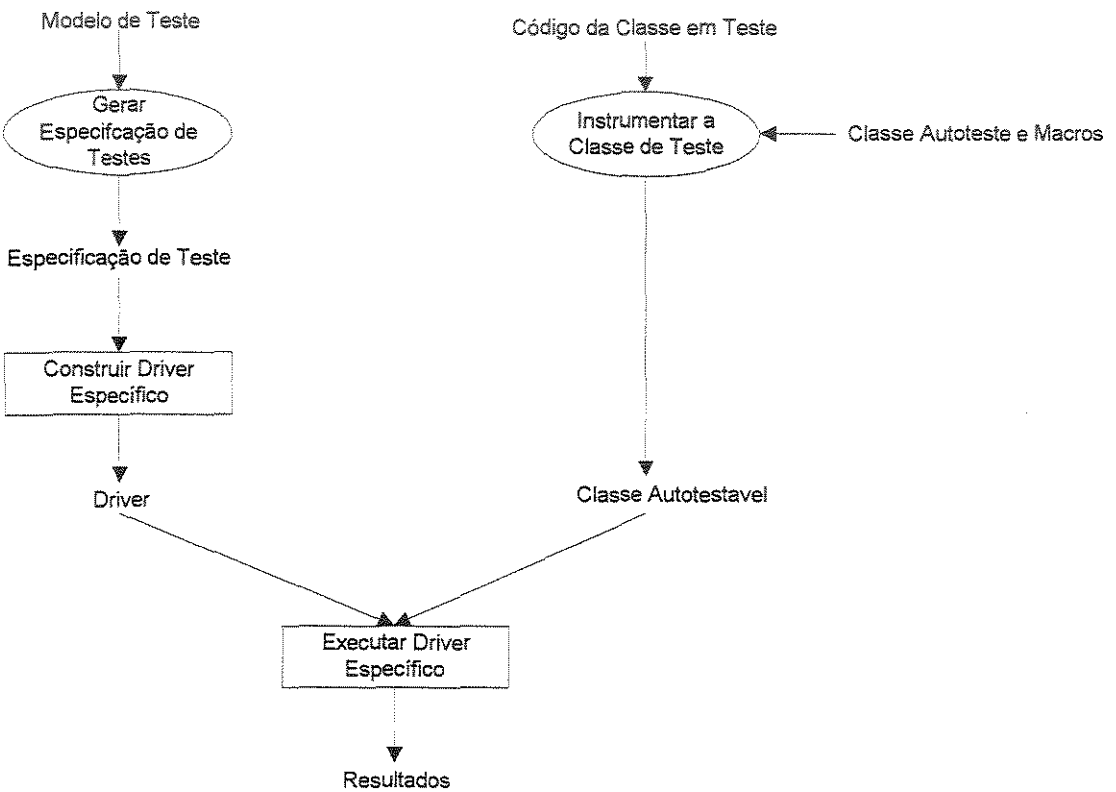

Figura 12 – Visão Geral da Ferramenta ConCAT

As atividades dentro das elipses são realizadas pelo usuário. Inicialmente o usuário deve construir um modelo da classe e em seguida representar esse modelo na forma de uma especificação de teste que descreve o modelo. Uma vez fornecida a especificação de teste, esta deve ser associada a classes em teste, para que toda vez que a classe seja reutilizada

sua especificação esteja disponível. Para isto foi proposto um padrão para nomear os arquivos contendo a especificação: o nome da classe e a extensão ".mt" indicando modelo de teste. Por exemplo, um arquivo contendo a especificação de teste de uma classe *A* se chamaria *A.mt*.

Outro passo a ser realizado é a instrumentação da classe, afim de introduzir capacidade de teste embutido: métodos de teste e assertivas. Após a criação da especificação de teste e da instrumentação da classe, o driver genérico é utilizado para construir o driver específico, ou seja, a seqüência de testes. Este driver ao ser executado gera um arquivo de resultados, contendo os casos de teste fornecidos e os resultados observados.

4.5 Trabalhos Relacionados

Esta Seção apresenta uma breve descrição de alguns trabalhos que abordam problemas relacionados a melhoria da testabilidade em sistemas orientados a objetos. Também é apresentada uma tabela comparativa entre os trabalhos descritos, avaliando algumas características que são relevantes no contexto de testabilidade. Os trabalhos que estão descritos são os de Jeffrey Voas (Seção 4.5.1), Yingxu Wang (Seção 4.5.2) e Yves Le Traon (Seção 4.5.3). Na Seção 4.5.4 é apresentada uma comparação dos trabalhos.

4.5.1 Jeffrey Voas et. al

Como citado anteriormente, a testabilidade abordada por [45] difere da definição tradicional, pois verifica a probabilidade de um programa "esconder" falhas. Normalmente existem partes do código de um programa que são descritas como "não testáveis", ou seja, regiões de código onde falhas não podem ser detectadas facilmente.

Para se construir um software visando a testabilidade definida por [45] deve-se projetar o software de tal forma que um defeito tenha grande probabilidade de ocorrer caso existam falhas. Desta forma, a preocupação está em sempre poder observar a ocorrência de um defeito. Esta ocorrência acontece quando o sistema produz uma saída diferente da saída esperada. Assim, é importante que após a execução de uma falha, o erro seja criado e principalmente, não seja mascarado antes de chegar a uma saída, pois isto pode encobrir a

existência da falha, diminuindo a testabilidade do sistema. Em [36] é apresentado um trabalho abordando a testabilidade da seguinte forma:

1. Verificar empiricamente as diferenças entre os paradigmas orientado a objetos e procedimental em termos de testabilidade.

2. Pesquisar como a testabilidade de sistemas orientados a objetos poderia ser melhorada.

3. Determinar as principais características para o desenvolvimento de um protótipo para melhorar a testabilidade de sistemas orientados a objetos.

O primeiro objetivo visava verificar se a escolha do paradigma de programação influenciava na testabilidade de um sistema. Para realizar esta tarefa um mesmo programa foi desenvolvido em C e C++, sendo que a partir desses programas foi realizada a análise de sensibilidade (descrita na Seção 4.3) utilizando a ferramenta *PiSCES* para fazer a comparação entre a testabilidade das duas implementações. Como resultado, verificou-se que o encapsulamento e as informações escondidas possuíam impacto negativo sobre a testabilidade do programa codificado em C++.

No segundo estudo procurou-se melhorar a testabilidade de sistemas orientados a objetos através do uso de assertivas. Para isto, uma análise de propagação foi realizada utilizando a ferramenta *PiSCES* para determinar onde as assertivas poderiam ser inseridas, ou seja, os autores apresentam uma abordagem que possibilita a melhoria da testabilidade através da localização das regiões que podem "esconder" as falhas e da inserção de assertivas que aumentem o potencial dos casos de teste em encontrar as falhas.

Após a execução desses estudos, foi proposta a construção de um protótipo para melhorar a testabilidade de sistemas orientados a objetos com as seguintes características:

1. Habilidade de ler as saídas geradas pela ferramenta *PiSCES* para determinar onde as assertivas poderiam ser inseridas.

2. Uso de uma meta-linguagem para a especificação das pré e pós condições.

3. Habilidade de ler as condições escritas nesta meta-linguagem e converter em código executável, inserindo as assertivas automaticamente no código fonte.

Baseado nestas características, foi desenvolvida a ferramenta Assert++ [46], que utiliza os escores de testabilidade obtidos através da análise de sensibilidade do programa para indicar os locais onde as assertivas devem ser inseridas no código e também oferecer suporte para a inserção das mesmas no código. Esta ferramenta trabalha com as linguagens C++ e Java.

4.5.2 Yingxu Wang et. al

Em [47] é apresentada uma abordagem para a utilização de mecanismos de teste embutido (BIT) em sistemas orientados a objetos. Segundo os autores um dos principais problemas existentes durante a fase de teste está relacionado ao fato de que, normalmente, o projeto, implementação e os testes do software são desenvolvidos por equipes diferentes e são descritos em documentos diferentes, ou seja, como mencionado na Seção 2.3, recomenda-se fases de teste sejam realizadas concorrentemente com as fases de desenvolvimento.

Para tentar minimizar este problema, foi proposto que mecanismos de teste sejam incorporados a classe sob teste (diretamente no código fonte) através de funções membros, como mostra a figura 13.

```
Classe nome_da_classe {
        //Interface
         declaração de dados;
         declaração do construtor;
         declaração do destrutor;
         declaração das funções;
         declaração dos testes;     //Mecanismos BIT

        //Implementação
         construtor;
         destrutor;
         funções;
         casos de teste;            //Casos de teste BIT
}
```

Figura 13 – Mecanismo de Teste

A estrutura descrita na figura 13 é uma extensão da estrutura convencional utilizada para definir uma classe através da associação de declarações de teste (interface) e casos de teste (implementação).

A principal característica desta técnica é que os casos de teste podem ser herdados e reutilizados da mesma forma que o código em sistemas orientados a objetos convencionais. Desta forma, além de possibilitar o reuso destes mecanismos toda vez que houver a necessidade de reteste e manutenção, existe ainda a possibilidade de reuso e herança destes mecanismos pelas classes derivadas. Isto porque os mecanismos são embutidos nas classes como funções membros, assim se uma nova classe for desenvolvida, ela pode herdar diretamente os mecanismos BIT como funções membros normais. Existe apenas a necessidade de incorporar novos requisitos de testes BIT à classe derivada para testar as alterações realizadas.

Além disso, os mecanismos BIT possuem dois tipos de execução: normal e de manutenção/teste. Na execução normal, o mecanismo BIT deve apresentar o mesmo comportamento do componente original, e isto acontece quando apenas funções membros normais são chamadas. Na execução em modo de manutenção e teste, os mecanismos BIT são ativados através de chamadas de funções membros BIT incorporadas ao componente.

Segundo os autores o mecanismo de autoteste proposto pode ser estendido para componentes e sistemas. Para isto é necessário que as classes que constituem o componente implementem os mecanismos BIT. Esta técnica foi aplicada na manutenção de componentes de software como pode ser visto em [48].

4.5.3 Yves Le Traon et. al

Em [42] é apresentada uma abordagem para testes de integração e testes de regressão de sistemas, através da aplicação do conceito de autoteste. Neste trabalho é proposto que um componente (classe) autotestável deve ser composto por uma especificação, uma implementação e um conjunto de casos de teste, ou seja, esta abordagem baseia-se no princípio de que o projeto e o teste de um componente devem ser integrados.

A figura 14 apresenta um modelo de ciclo de vida mostrando este princípio, onde a especificação construída pode ser utilizada tanto para os testes quanto para a implementação do componente.

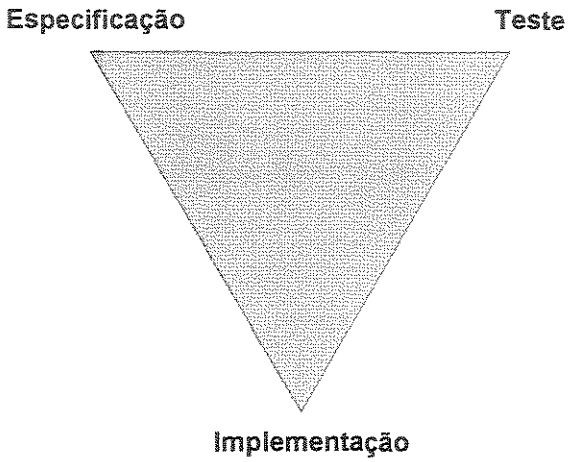

Figura 14 – Processo em "V" para desenvolver um componente

O autoteste é introduzido no componente através de uma relação de herança feita entre este e uma classe autoteste criada, de forma a introduzir rotinas de teste que deverão exercitar todos os seus métodos. Desta forma os casos de teste são definidos como sendo parte do componente de software.

Estas rotinas têm o objetivo de testar se a implementação de cada método corresponde à sua especificação. Para a geração do oráculo, a solução encontrada foi determinar manualmente o resultado esperado para cada requisito de teste criado. Assim, o testador fica responsável por criar e verificar os resultados dos testes.

Em [17] é descrito que um componente autotestável é composto dos seguintes elementos (para as linguagens C++, Java e Perl):

- Configurador – Mecanismo de configuração de parâmetros que controla se os elementos de teste são executáveis ou não.
- Contrato – Especifica o contrato do componente
- Autoteste – Métodos de teste (driver embutido)
- Classe em Teste

Um detalhe que pode ser ressaltado em relação ao mecanismo de configuração é que em Eiffel, uma opção de compilação permite retirar a instrumentação dos testes quando se

gera a versão final. Em C++ o código das funções (métodos) de teste ficam no código final, todavia este não pode ser acessado. Isto é realizado através do Configurador.

Além disso, sugere-se o uso de métricas de qualidade para verificar a eficácia do conjunto de teste gerado. Esta estimativa pode ser associada ao componente fornecendo um critério importante principalmente durante a fase de seleção de componentes. Para realizar a obtenção dessas estimativas foi utilizada a técnica de análise de mutantes (descrita no Capítulo 2).

Na realização dos testes, um modelo de dependência entre os componentes do sistema é proposto, baseado na herança e associação existente entre as classes. Este modelo serve para guiar a realização dos testes, usando para estes o autoteste introduzido nas classes. Além disso, dependendo do contexto de validação (testes de integração ou de regressão) critérios de testes são propostos.

4.5.4 Comparação dos Trabalhos

Esta Seção apresenta uma comparação dos trabalhos descritos nas seções anteriores. A tabela 2 foi definida com base nas propriedades que afetam a testabilidade (capacidade teste embutido, geração de teste, reutilização dos casos de teste e ferramentas de teste) descritas na Seção 4.2 e também nos principais objetivos de cada trabalho.

Trabalhos	Capacidade de teste embutido	Geração de testes	Reutilização dos casos de teste	Ferramentas de teste	Contribuição para questões de teste
Jeffrey Voas et al	Assertivas	---	---	*PiSCES* e Assert ++	Indicam que o uso de assertivas é útil para melhorar a testabilidade e fornecem ferramentas que auxiliam na inserção de assertivas
Yingxy Wang et al	Casos de teste embutidos diretamente no código fonte	---	Os casos de teste podem ser herdados e reutilizados da mesma forma que o código fonte	---	Mecanismos de teste são embutidos no componente, facilitando o reuso dos mesmos
Yves Le Traon et al	Rotinas de teste no código fonte	Propõe alguns critérios para teste de integração e regressão mas os testes não são gerados automaticamente	Os casos de teste podem ser herdados e reutilizados da mesma forma que o código fonte	---	Apresenta uma abordagem para o uso do autoteste (casos de teste embutidos no componente) e também na análise de mutantes como métrica de qualidade

Tabela 2 – Comparação dos Trabalhos

4.6 Considerações Finais

Neste capítulo foram apresentados o conceito de testabilidade e os aspectos de software que influenciam na testabilidade, tais como representação, implementação, seqüência de teste e seu processo de desenvolvimento. Também foram descritas algumas métricas de testabilidade e a técnica de análise de sensibilidade proposta por Voas. Além disso, foram descritos o conceito de DFT e a aplicação da abordagem BIST para a construção de classes autotestáveis.

Da aplicação da abordagem BIST (estratégia aplicada na ferramenta ConCAT), surge a idéia de componentes autotestáveis utilizada neste trabalho. Desta forma, pretende-se estender a estratégia utilizada na ferramenta ConCAT para testar-se componentes de software compostos por um conjunto de classes, com o intuito de construir componentes reutilizáveis que sejam fáceis de testar afim de reduzir os custos com a realização dos testes. O capítulo a seguir apresenta esta estratégia.

Capítulo 5

Estratégia Proposta para a Construção de Componentes de Software Autotestáveis

Com base nos estudos apresentados nos capítulos anteriores, pôde-se verificar a importância da melhoria da testabilidade para a construção de componentes de software com boa qualidade. Desta forma, foi proposta uma estratégia que tem como objetivo estender o conceito de autoteste utilizado no trabalho de [43] para a construção de componentes de software autotestáveis. A extensão desta técnica se mostrou interessante porque apesar de um componente de software poder conter uma única classe, este, na maioria das vezes, é composto por um conjunto de classes relacionadas.

Além disso, é importante ressaltar que neste trabalho serão considerados testes pela visão do usuário (consumidor) do componente, ou seja, o objetivo dos testes será verificar as funcionalidades oferecidas por um componente independente da forma como estas foram implementadas. Isto foi realizado visando auxiliar a etapa de teste descrita na figura 10 (Seqüência de Atividades da CBSE) do Capítulo 3, pois o desenvolvimento de componentes autotestáveis poderá oferecer aos usuários uma forma de realizarem testes que verifiquem as funcionalidades desses componentes.

Este capítulo tem como objetivo descrever os elementos que compõem a estratégia proposta para a construção de componentes de software autotestáveis, sendo que primeiramente serão apresentadas algumas considerações feitas para a sua definição (Seção 5.1). Em seguida, são descritos os elementos da estratégia: especificação do componente (Seção 5.2), especificação de teste (Seção 5.3), mecanismos de teste embutido (Seção 5.4) e geração dos casos de teste (Seção 5.5). Na Seção 5.6 temos um sumário do capítulo. Exemplos de utilização e uma avaliação das vantagens e desvantagens dessa estratégia são apresentados no Capítulo 6.

5.1 Considerações Iniciais

Um aspecto que deve ser considerado para estender o conceito de autoteste para componentes de software (compostos por um conjunto de classes) é a estrutura do componente. A figura 11 (Seção 4.4) mostra a estrutura de um componente autotestável constituído de uma única classe. Para o caso do componente possuir mais do que uma classe, pode-se aplicar duas soluções distintas:

- Tornar todas as classes pertencentes ao componente autotestáveis - cada classe deverá conter a sua própria especificação de teste através da qual serão gerados os casos de teste. Por exemplo a figura 15 apresenta um componente composto de 3 classes e suas respectivas especificações de teste.

- Criar uma única especificação de teste para o componente - desenvolver apenas uma especificação de teste que represente o comportamento do componente, e inserir os mecanismos embutidos de teste em todas as classes. A figura 16 ilustra esta solução.

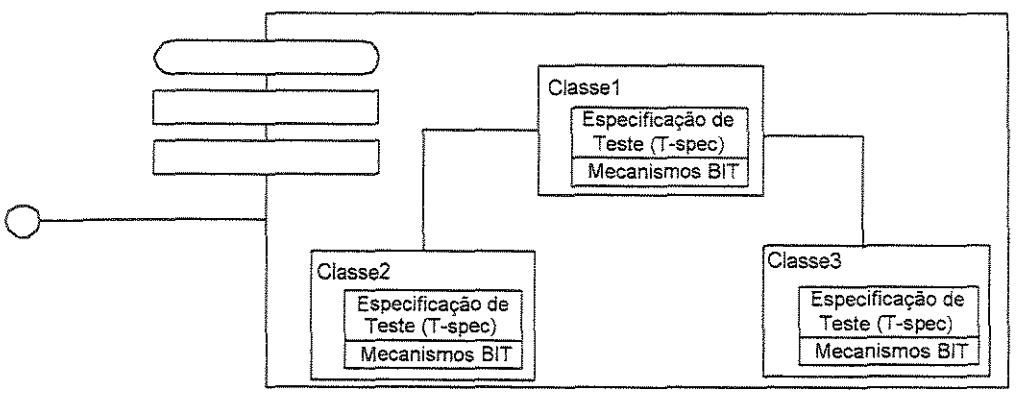

Figura 15 – Arquitetura da Primeira Solução

A solução adotada neste trabalho foi a da figura 16 devido aos seguintes fatores. A solução proposta na figura 15 pode ser útil para testar objetos individualmente dentro do componente, o que pode ser útil para o seu desenvolvedor, mas não para o consumidor, que está mais interessado em determinar se o componente provê os serviços que ele deseja.

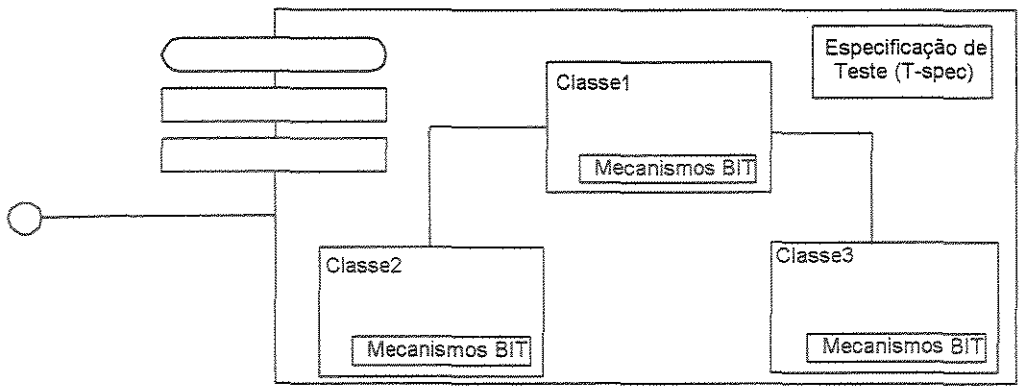

Figura 16 – Arquitetura da Segunda Solução

Além disso, a geração dos casos de teste para a primeira solução pode possivelmente resultar em uma explosão combinatória caso o consumidor não realize testes de unidades, pois as especificações de teste de cada classe devem ser combinadas para a criação da especificação do componente. Portanto é mais interessante construir uma única especificação de teste descrevendo os métodos da interface do componente, todavia, como construir tal modelo e uma especificação de teste a partir da qual os casos de teste possam ser gerados?

Outro aspecto importante na construção de componentes autotestáveis consiste na inserção de mecanismos de teste embutidos: assertivas e métodos relatores. Em [43] as assertivas foram utilizadas para descrever o contrato entre a classe e seus clientes, sendo que os métodos relatores forneciam visibilidade sobre o estado do objeto. No contexto de componentes de software o que os mecanismos BIT irão representar? As próximas seções discutem estas questões.

5.2 Especificação do Componente

Neste trabalho foram utilizados dois modelos para representar as funcionalidades do componente. Isto foi realizado para que diferentes aspectos do componente (seqüências de execução válidas e contrato do componente) fossem cobertos, afim de que estes possam ser explorados durante os testes. As próximas seções apresentam estes modelos e os aspectos que se deseja representar através dos mesmos.

5.2.1 Modelo de Fluxo de Transação

Definido por Boris Beizer, o modelo de fluxo de transação (MFT) foi projetado inicialmente para uso em testes de sistemas [4], visando descrever todas as etapas do processamento de uma transação, representando desta forma o seu fluxo (ou ciclo de vida). Em [1] uma transação é definida como:

(1) Um comando, mensagem ou registro de entrada que é chamado explicitamente ou implicitamente para o processamento de uma ação, tal como a atualização de um arquivo.

(2) Uma interação entre um usuário final e um sistema.

(3) Em um SGBD (Sistema de Gerenciamento de Banco de Dados), uma unidade de processamento que executa um propósito específico, tais como uma recuperação, atualização, modificação ou remoção de um ou mais elementos de uma estrutura de armazenamento.

Nos trabalhos desenvolvidos por Boris Beizer [4] [5], uma transação foi utilizada para representar uma seqüência de operações que podem ser realizadas pelo sistema, pessoas ou dispositivos que estão fora do sistema. Por exemplo, a figura 17 ilustra a seqüência de tarefas que devem ser executadas para realizar a operação de saque.

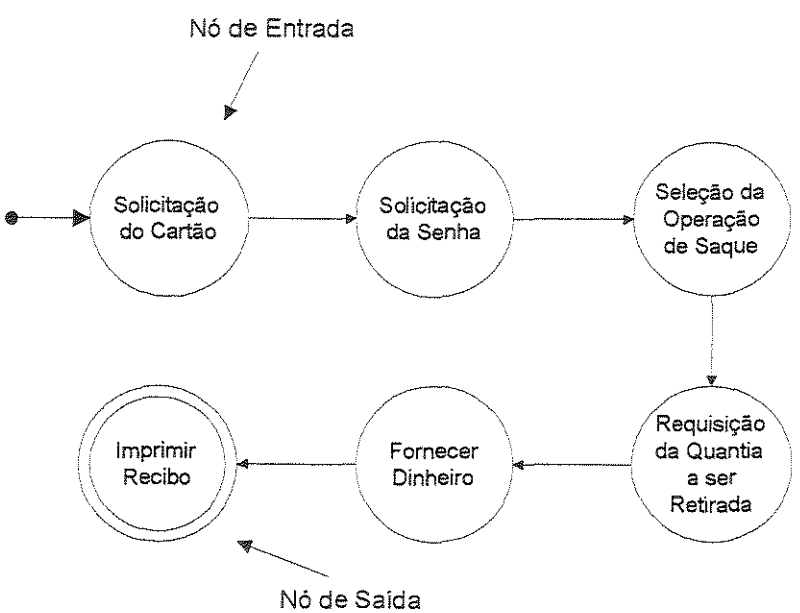

Figura 17 – Modelo de Fluxo de Transação

Graficamente o MFT é um grafo, onde os nós representam as etapas (ou tarefas) de processamento de uma transação e os arcos designam as seqüências entre os nós. Uma característica interessante do MFT é que ele fornece recursos que auxiliam na especificação de sistemas concorrentes. Por exemplo a figura 18 ilustra uma transação representando o processo de preparação de café (através do uso de uma cafeteira), que contém algumas tarefas que podem ser realizadas concorrentemente.

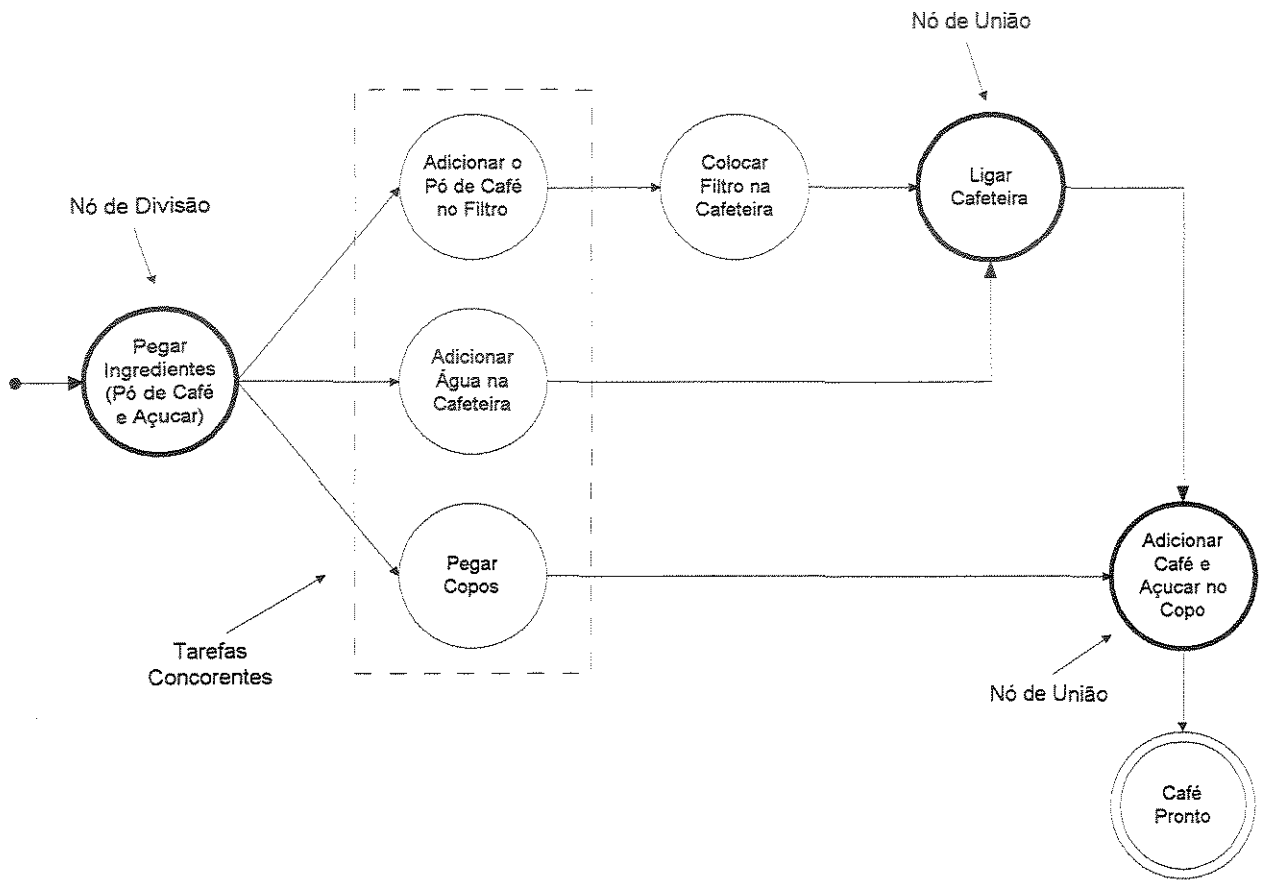

Figura 18 – Concorrência no Modelo de Fluxo de Transação

As tarefas que estão no interior do retângulo representam este caso, pois estas tarefas podem ser executadas simultaneamente. Para facilitar a modelagem deste tipo de situação, o MFT apresenta alguns tipos de nós especiais como os nós de divisão e união que estão em destaque na figura 18.

A presença destes tipos de nós obriga a existência de testes de sincronização para garantir a correção do sistema, por exemplo, no caso de um grafo apresentar nós de união este tipo de teste será necessário pois existem algumas situações que podem ocorrer durante a execução, como: as transações chegarem no mesmo instante ao nó ou uma alcançar o nó antes da outra.

Apesar do MFT apresentar algumas funcionalidades para a modelagem de sistemas concorrentes, este fator não foi tratado neste trabalho pois a ferramenta utilizada para a aplicação da estratégia (ConCAT, descrita no Capítulo 4) permite apenas a geração de casos de testes para sistemas que não apresentam concorrência.

Posteriormente Shel Siegel adaptou o MFT para o teste funcional de classes. Em [39], o modelo de fluxo de transação mostra as diferentes formas para a criação de um objeto, as diferentes tarefas ou processos que este pode realizar e as diferentes formas de destruí-lo.

As transações representam ciclos de vida de um objeto. Os nós representam os processos (tarefas) em si, ou seja, refletem os atributos e métodos públicos da classe, encapsulando todas as referências para outros objetos e os arcos representam as seqüências de execução válidas.

O grafo de fluxo que modela a transação inicia com a construção do objeto (possivelmente com múltiplos construtores). O fluxo descreve como os objetos respondem ao ambiente através da chamada de métodos. Cada nó no diagrama representa um ponto de controle na seqüência de comportamento, o modelo termina quando se destrói o objeto, chamando o destruidor para a classe.

Para utilizar o MFT como modelo de base para o teste de componentes foi estabelecida a seguinte definição para uma transação, baseada nas definições de [4] e [39]. Uma transação irá representar uma seqüência de execução de métodos, sendo considerados somente métodos pertencentes a interface do componente.

Isto foi realizado pois, como mencionado anteriormente, o objetivo dos testes neste trabalho é verificar se o componente realmente executa as funcionalidades oferecidas independente da forma como estas foram implementadas. Portanto o grafo representará as diferentes seqüências de tarefas ou funcionalidades que o componente pode realizar.

Os arcos irão representar as seqüências de execução e os nós os métodos ou conjunto de métodos que devem ser executados. Este grafo pode ser construído a partir do refinamento dos casos de uso do componente através dos seguintes passos:

1. Analisar a especificação e identificar todos os casos de uso que o componente pode processar. Para cada caso de uso identificado deve-se analisar quais são as tarefas que devem ser executadas para realizá-lo.

2. Realizar o mapeamento entre as tarefas identificadas no passo 1 para as operações da interface do componente, ou seja, para cada tarefa identificada verificar se existe uma operação da interface correspondente, sendo que mais do que uma tarefa pode ser mapeada para uma mesma operação. No caso de uma tarefa não ser mapeada para nenhuma operação, esta deve ser analisada novamente para verificar se é realmente necessária ou se o componente não a implementa. Também existe a possibilidade da especificação estar desatualizada, neste caso deve-se atualizá-la. Além disso, neste passo também é importante verificar se alguma tarefa não necessita de operações pertencentes a interface de outro componente (por exemplo, uma operação utiliza um objeto pertencente a outro componente como parâmetro de entrada), pois caso isto ocorra é necessário criar stubs que substituam o componente que possui esta interface para a realização dos testes.

3. Com base nos dados obtidos nos passos 1 (seqüências de execução) e 2 (métodos da interface do componente) pode-se construir o MFT, sendo que as seqüências de execução serão representadas pelos arcos e os métodos ou conjunto de métodos pelos nós do grafo de fluxo de transação.

A figura 19 apresenta um resumo do processo de construção do MFT a partir dos casos de uso de um componente. No Capítulo 6 são apresentados dois casos de estudo em que os passos na figura 19 foram aplicados para a construção do MFT.

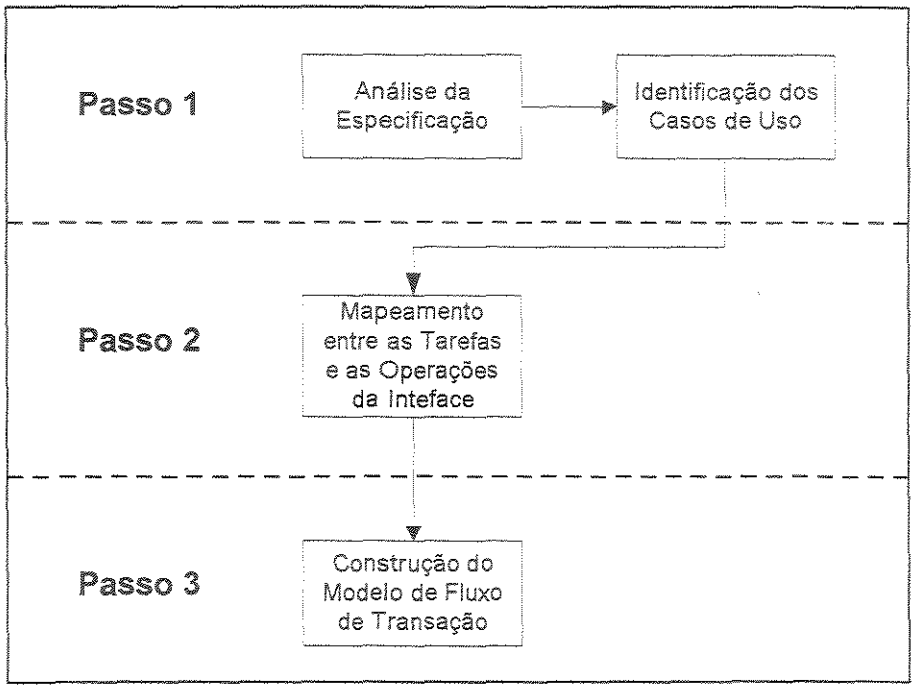

Figura 19 – Passos para a construção do MFT

A geração de casos de teste a partir do MFT é realizada através do percurso do grafo de fluxo de transação. Em [5] são propostos alguns métodos para a derivação de casos de testes a partir do MFT, como por exemplo:

- Cobertura dos Nós Entrada/Saída - Executar testes para assegurar que todas as entradas tenham sido exercitadas e que todas as saída tenham sido produzidas.

- Cobertura dos Nós - Este teste visa cobrir todos os nós, portanto tem como objetivo confirmar se todos os nós estão corretos.

- Cobertura dos Arcos - Através da cobertura dos arcos pode-se verificar não somente a correção do nós, mas também a interação entre os mesmos.

Segundo [4] as falhas que podem ser encontradas através do MFT são: falta de transações (funcionalidades), falta de arcos/nós e problemas com os tipos de nós (sincronização).

5.2.2 Assertivas

Uma assertiva é uma expressão booleana que define as condições necessárias para a execução correta de um sistema [10]. Segundo [2] as assertivas podem ser utilizadas para especificar os serviços fornecidos pelo componente e também as propriedades que este requer para que o componente forneça estes serviços corretamente, ou seja, como descrito no Capítulo 3, as assertivas podem ser utilizadas para expressar o contrato do componente. Assim, as assertivas podem ser utilizadas para:

- Especificar as condições que o componente requer do cliente para a utilização das operações de sua interface, ou seja, as condições que devem ser válidas para a execução de uma operação da interface do componente (pré-condição).

- Especificar as condições que devem ser válidas para a saída de uma operação da interface do componente (pós-condição). Uma pós-condição deve ser avaliada no momento em que a operação termina a execução e antes que a mensagem/valor resultante seja retornada para o cliente.

- Especificar condições que devem ser válidas para todas as operações de uma interface, ou seja, condições que se aplicam a todas as instâncias do componente que implementa esta interface (invariante).

Geralmente uma assertiva é composta de três partes [10]: um predicado, uma ação e um mecanismo para habilitar/desabilitar as assertivas durante a execução do componente. Os predicados descrevem as cláusulas ou condições que devem ser avaliadas. Uma ação pode ser escrita como uma simples mensagem ou como um mecanismo de recuperação. Em geral, existe um mecanismo de configuração que é utilizado para habilitar ou desabilitar as assertivas. Este recurso é útil para que um componente sob teste possa ser configurado em modo teste (assertivas habilitadas) ou em modo normal (assertivas desabilitadas), para que nenhum código "extra" seja utilizado durante a execução normal do sistema.

Quando alcança-se uma assertiva habilitada, verifica-se o valor do seu predicado (verdadeiro ou falso). Se a assertiva é verdadeira, a execução continua normalmente. Caso contrário diz-se que ocorreu uma violação da assertiva. Esta situação é algumas vezes chamada de falha de assertiva. Uma violação de uma assertiva resulta em uma transferência de controle para a ação da assertiva. Esta ação, geralmente, irá gerar uma mensagem de

diagnóstico e terminará a execução. A ação tem duas responsabilidades: notificação e continuação. As tabelas abaixo sugerem algumas opções [10].

	Modo de Continuação				
Opções de Continuação	0	1	2	3	4
Terminar	X				
Continuar Execução		X			
Sugerir Continuar ou Terminar			X		
Ativar *Debugger*				X	X
Tentar recuperar					X

Tabela 3 – Opções e modos de continuação

Ações avançadas podem incluir um relatório de falhas automático, na qual pode ser gravado um registro em um arquivo para ser usado na depuração do sistema.

	Modo de Notificação				
Opções de Notificação	0	1	2	3	4
Escrever no Console – *Default*	X	X	X	X	X
Escrever no *Log*		X	X	X	X
Escrever no Relatório de Falhas			X	X	X
Gerar um *trace*				X	X

Tabela 4 – Opções e modos de notificação

Na construção das assertivas deve-se analisar todos os dados que a operação tem disponível e utiliza, tais como variáveis, parâmetros e atributos. Em muitos casos, as assertivas são implementadas como exceções ou comandos como a macro *assert* na linguagem C++ que aborta o programa se a condição testada é falsa.

Para facilitar a inserção das assertivas existem algumas ferramentas, como por exemplo, a JML (do inglês, *Java Modeling Language*) [27], que auxilia a especificação de assertivas em linguagens de programação.

As assertivas também podem ser derivadas da OCL (do inglês, *Object Constraint Language*) [20]. A OCL é uma linguagem definida para expressar condições e outras expressões podendo ser utilizada juntamente com os diagramas da UML. Dessa forma, as assertivas usadas nos testes deveriam ser especificadas no modelo de análise.

Neste trabalho as assertivas (pré e pós condições) foram definidas através de algumas macros propostas em [43], sendo que, no caso de uma violação de uma assertiva a execução da seqüência de teste não é interrompida e uma mensagem de falha de assertiva é registra, ou seja, as opções de Continuação e Notificação (tabelas 3 e 4, Seção 5.2.2) adotadas foram a continuação da execução e o registro de uma mensagem de falha.

5.3 Especificação de Teste

Para que os testes possam ser gerados a partir do modelo de teste escolhido é necessária a construção de uma especificação de teste que descreva esse modelo. Assim, seu formato e conteúdo podem variar conforme o modelo que ela representa, pois ela deve conter as informações necessárias para gerar casos de teste que cubram tal modelo.

No caso do MFT é importante que a especificação de teste descreva todas as seqüências de tarefas válidas e também os parâmetros necessários para cada operação e os seus domínios, pois dependendo dos valores fornecidos para um determinado método este pode assumir caminhos distintos durante a sua execução. Desta forma, estas informações são muito importantes para que os testes possam ser gerados e controlados.

Em relação as assertivas, apesar de estarem definidas na Seção 5.2.2 como sendo parte da especificação do componente, elas não serão representadas na especificação de teste pois elas serão inseridas diretamente no código fonte do componente (mecanismos BIT). Portanto, a especificação de teste conterá informações relacionadas ao MFT e também sobre as classes, métodos, atributos e parâmetros pertencentes ao componente.

Um possível formato de uma especificação de testes para o modelo de fluxo de transação está ilustrado na figura 20 e foi definida em [43]. Nesta especificação são descritas informações sobre o grafo de fluxo de transação para que se possa derivar mensagens aceitas pelo componente. Desta forma, as informações relacionadas a cada nó auxiliarão no percurso do grafo para a derivação das transações. Além disso, informações sobre os métodos, atributos e parâmetros também são fornecidos para auxiliarem na geração de casos de teste.

```
Classe (Nome
        Abstrata/concreta          // Indica se a classe é abstrata ou não
        Nome Super Classe
        Lista_arq)                 // Lista de arquivos a serem incluidos na compilação

Atributo ( Nome,
          Tipo                     // Pode ser intervalo, string, conjunto, objeto ou ponteiro.
          Dado1                    // Para o tipo intervalo indica o valor mínimo,
                                   // String ou conjunto indica uma lista de valores permitidos
                                   // Objeto ou ponteiro indica a classe envolvida
          Dado2)                   //Para o tipo intervalo indica o valor máximo,
                                   // para os outros tipos não é utilizado

Método (Número_Identificador,      // Identificador único para o método
        Nome,
        Tipo_Retorno,              // Similar ao tipo atributo
        Classificação,             //Indica se o método é novo, recursivo ou redefinido.
                                   // Maiores detalhes em [43].
        Número_Parâmetro)          // Número de parâmetros existentes na assinatura do método

Parâmetro (Nome,
           Método Pertencente,     // Identificador do método
           Tipo,                   // Similar ao tipo atributo
           Dado1,
           Dado2)

Nó (Número_Identificador,          //Identificador único para o nó
    Nó_Inicial,                    //Indica se o nó é inicial ou não
    Arcos_Saída,                   //Indica o número de arcos de saída
    Lista_Métodos)                 //Lista das  assinaturas dos métodos que compõem o nó

Arco (Nó_Origem,                   //Identificador do nó de origem
      Nó_Destino)                  //Identificador do nó de destino
```

Figura 20 – Especificação de Teste

A especificação de teste da figura 20 foi construída para representar componentes de software constituídos de uma única classe, portanto, a especificação de teste aplicada neste trabalho foi construída com algumas modificações. A principal mudança foi a representação de métodos pertencentes à diferentes classes. Isto foi necessário para que se pudesse utilizar a ferramenta ConCAT para a geração dos casos de teste. A ConCAT faz a

leitura da especificação de teste e gera alguns arquivos que contêm dados de teste (casos de teste e seqüência de teste – driver específico).

Para isso, durante a geração dos casos de teste, considerou-se que todos os métodos públicos pertencessem a uma única classe. Dessa forma, foi possível gerar as seqüências de casos de teste automaticamente, utilizando a estratégia implementada na ferramenta ConCAT.

5.4 Mecanismos de Teste Embutido

O componente deve ser acrescido de mecanismos de teste embutido: assertivas (Seção 5.2.2) e métodos relatores. Estes mecanismos devem ter visibilidade sobre os membros privados do componente afim de facilitar o teste e a análise dos resultados dos casos de teste aplicados. Além disso, de acordo com [7] estes mecanismos devem ter uma interface padrão para facilitar a geração da seqüência de teste. A seguir temos uma descrição destes elementos:

- Assertivas - Como descrito nas seções anteriores, as assertivas serão utilizadas para expressar parte da especificação do componente, mais especificamente o contrato do componente, sendo inseridas diretamente no código fonte do componente como descrito na seção 5.2.2.

- Métodos relatores - Servem para armazenar os valores dos atributos dos objetos que compõem o componente durante a execução dos testes. Portanto os métodos relatores irão mostrar o estado do componente durante a execução dos casos de teste. Para isso é necessário que cada classe pertencente ao componente implemente um método relator, pois cada classe possui seus próprios atributos, os quais devem ser armazenados em um arquivo para análise posterior. Dessa forma, é importante que durante os testes os métodos relatores sejam executados antes e depois da execução de cada método da interface do componente afim de avaliar-se o estado do componente antes e após de cada método.

- Interface de Teste - Foi sugerido que um componente autotestável deverá possuir uma interface padrão para os mecanismos de teste embutido. O conceito de interface de teste foi proposto através de uma extensão do conceito de interface do

componente, em que a interface resume as funcionalidades do componente e a interface de teste representa as facilidades que o componente apresenta para a realização dos testes. A interface de teste (figura 21) do componente será composta de dois métodos BIT: um relator (para observar o estado do componente) e outro para o teste das invariantes do componente.

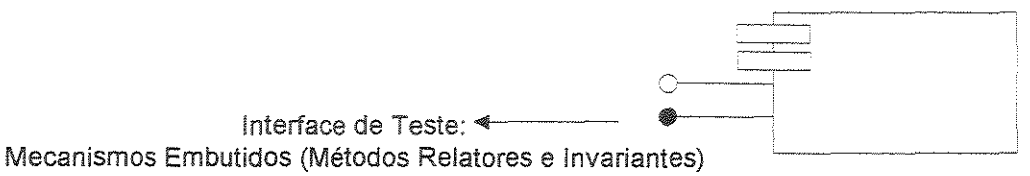

Figura 21 - Interface de Teste

A solução utilizada para adicionar os métodos relatores e os testes de invariante no componente foi a mesma aplicada em [43], onde utilizou-se uma classe abstrata denominada Autoteste, que define uma interface padrão para os mecanismos BIT. Assim as classes pertencentes ao componente devem tornar-se classes derivadas de Autoteste. Os métodos herdados da classe Autoteste devem ser redefinidos pelo usuário. Estes métodos possuem um formato pré-definido e podem ser estendidos para incluir outros dados que o usuário desejar.

5.5 Geração dos Casos de Testes

Uma vez que o componente possua uma especificação de teste e seja instrumentado com os mecanismos de teste embutido, este ainda necessita de um gerador de driver que deve ser capaz de ler a especificação de teste do componente e gerar um driver específico que contém uma seqüência de casos de teste. Além disso, este driver é responsável por ativar e controlar o componente durante a execução dos casos de teste.

O gerador de driver utilizado neste trabalho foi a ferramenta ConCAT, descrita no Capítulo 4. A ferramenta ConCAT faz a leitura da especificação de teste e gera alguns arquivos que contêm dados de teste. Esta ferramenta utiliza além da estratégia baseada no modelo de fluxo de transação, uma estratégia de teste de dados para a geração dos casos de teste.

A estratégia de teste de dados é necessária para selecionar os valores dos parâmetros dos métodos. Na implementação atual, a estratégia consiste em gerar os dados

aleatoriamente de acordo com a descrição dos valores possíveis como consta na especificação de teste. Isto pode ocasionar alguns problemas, por exemplo ao testar a operação de saque (figura 17) é necessário que a senha informada seja válida para o cartão utilizado. Desta forma, para aumentar a eficiência dos casos de teste, em alguns casos necessário alterar os dados gerados pela ConCAT para a realização dos testes.

5.6 Considerações Finais

Neste capítulo foi apresentada uma proposta para extensão do conceito de autoteste utilizado no desenvolvimento de classes autotestáveis em [43] para a construção de componentes de software autotestáveis. A principal alteração foi a adaptação do modelo de fluxo de transação para representar o comportamento do componente. Desta forma, o MFT representará as diferentes seqüências de tarefas ou funcionalidades que o componente pode realizar. Em relação ao métodos de teste embutido, as assertivas foram utilizadas para representar o contrato do componente e o os métodos relatores o estado do componente. Além disso, foi proposto que um componente autotestável possua uma interface de teste que deverá conter operações específicas para a realização de testes, mais especificamente métodos relatores e invariantes. No próximo Capítulo são apresentados exemplos de utilização da estratégia proposta e também uma análise das dificuldades e vantagens de sua utilização.

Capítulo 6

Estudos de Caso

Neste capítulo são apresentados dois estudos de casos para exemplificar a estratégia proposta para a construção de componentes de software autotestáveis. Os estudos de caso escolhidos foram um ATM (*Automatted Teller Machine*) e um controle de acervo para uma vídeo locadora. Este capítulo também apresenta uma avaliação das dificuldades e vantagens da utilização da estratégia proposta e uma análise da cobertura de código obtida através da execução dos casos de teste gerados a partir do MFT.

Para auxiliar na realização dos testes foram utilizadas duas ferramentas: a ConCAT (descrita no Capítulo 4) para a geração dos casos de teste e a Panorama (descrita na Seção 6.3.2) para realizar a análise da cobertura de código obtida pelos testes.

O capítulo está organizado da seguinte forma: na seção 6.1 é apresentado o estudo de caso do ATM incluindo a descrição do componente ATM e também todas as etapas para a construção de componentes autotestáveis descritas no Capítulo 5. A seção 6.2 apresenta o estudo de caso do Acervo contendo todos os passos da seção anterior. A seção 6.3 descreve a avaliação realizada para a análise da estratégia proposta e por fim a seção 6.4 apresenta um sumário do capítulo.

6.1 Estudo de Caso 1 – ATM

6.1.1 Descrição do Componente

Este estudo de caso foi baseado no sistema de Controle de ATM apresentado em [46] que simula um ATM conectado a um sistema bancário. Esta conexão é simulada através da leitura de um arquivo (arquivo de conexão) que contém dados sobre os clientes do banco. O

sistema de Controle de ATM deve ainda realizar a autenticação do usuário através da verificação dos dados do cartão e senha do usuário.

Uma vez realizada a autenticação do usuário, um dos seguintes tipos de transações bancárias pode se executado: saque, transferência ou verificação de saldo. Outras características do funcionamento deste sistema são:

- Os tipos de conta que o banco apresenta são conta corrente e poupança.

- Uma transação é inválida se o usuário tentar transferir/retirar uma quantia maior do que R$200,00 ou tentar transferir/retirar uma quantia que ultrapasse o saldo atual.

- Se as informações do cartão utilizadas forem inválidas (cartão danificado), o sistema deverá reter o cartão indicando que o mesmo está com problemas e que o cliente deverá procurar uma agência.

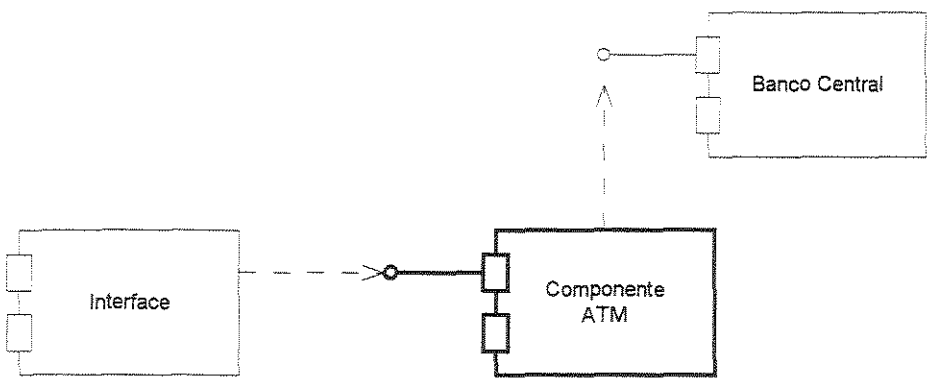

Figura 22 – Estrutura do Sistema de Controle de um ATM

A figura 22 apresenta parte da estrutura desse sistema, sendo que estes componentes são responsáveis por:

- Componente Interface: Responsável pela interação com os usuários externos, obtenção e passagem dos dados para o sistema computacional.

- Componente ATM: Responsável por estabelecer a conexão entre o ATM e o banco central. Além disso, realiza a autenticação do usuário (verificação dos dados do

cartão e senha) e as transações de saque, transferência, verificação de saldo. Este componente também é responsável por gerar um recibo sobre a transação executada.

- Componente Banco Central: Responsável por disponibilizar os dados/contas dos clientes (base de dados).

O componente selecionado para exemplificar a estratégia proposta para construção de componentes de software autotestáveis foi o Componente ATM. Este componente foi escolhido para tornar-se autotestável pois ele deve prover várias operações, sendo composto por um conjunto de classes como pode se visto posteriormente na próxima Seção.

As próximas seções descrevem os passos executados para tornar este componente autotestável.

6.1.2 Construção do Modelo de Fluxo de Transação

Como descrito no capítulo 5, a estratégia proposta engloba um conjunto de passos para a introdução do autoteste em um componente. A primeira etapa consiste na construção do modelo de fluxo de transação. Desta forma, nesta seção serão apresentados os passos para a sua construção.

Passo 1: Identificação dos casos de uso e levantamento das tarefas

Nesta etapa devem ser identificados os possíveis casos de uso que o componente pode realizar e as tarefas necessárias para realizá-los. Isto foi realizado através da análise da especificação considerando-se os casos de funcionamento normal e também os de exceção.

Um caso de uso descreve uma interação entre um ator e o sistema, sendo que este pode ser descrito por uma seqüência de tarefas (atividades) chamada de cenário. As tarefas de um cenário podem ser bem ou mal sucecidades. Um caso de uso de funcionamento normal é aquele livre de erros, isto é, todas as tarefas do cenário foram executadas com sucesso. Já os casos de uso de exceção são representados por desvios (situações excepcionais) ocorridas nas tarefas do cenário normal. A figura 23 apresenta os casos de uso identificados para o componente ATM.

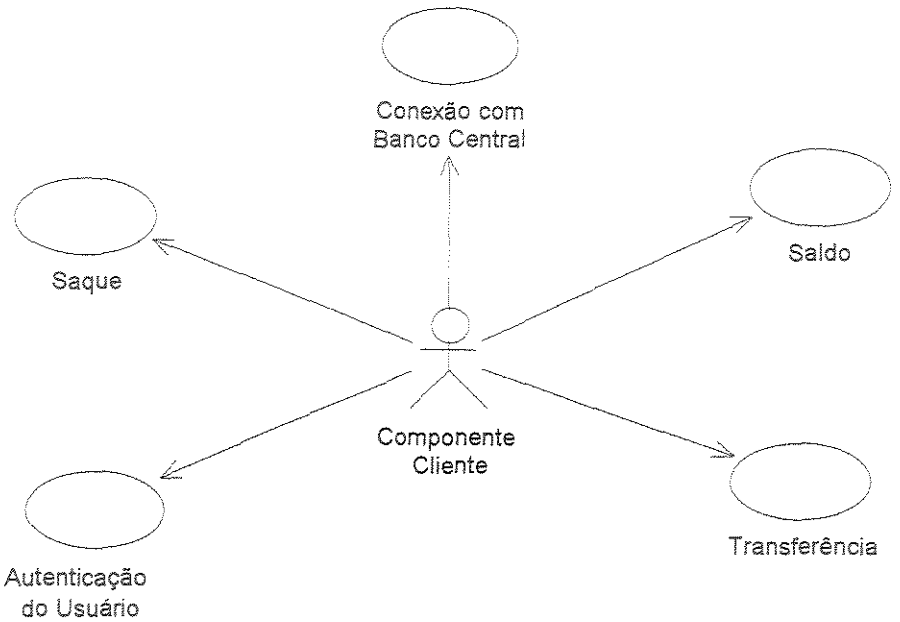

Figura 23 – Casos de Uso – Componente ATM

A descrição de um caso de uso normal pode incluir, além do seu fluxo de execução, suas pré-condições, invariantes e pós-condições [38].

Situações excepcionais são decorrentes de violação das pré-condições, invariantes ou pós-condições do comportamento normal do caso de uso. Violação de pós-condição ou invariantes representam que o serviço não pode ser realizado devido a falhas internas ao caso de uso como problemas na execução de qualquer de suas atividades. Violação de pré-condições representam falhas no cliente ao solicitar a realização do serviço fornecido pelo caso de uso.

A descrição textual comportamento dos caso de uso "Conexão com Banco Central", "Autenticação do Usuário" e "Saque" estão apresentadas nas Figuras 24, 25 e 26 respectivamente. As descrições dos demais casos de uso estão no Apêndice A.

Caso de Uso Conexão com Banco Central	
Descrição:	O Componente Interface solicita ao Componente ATM para que a conexão com o Banco Central seja estabelecida. Para simular esta conexão o Componente ATM realiza a leitura de um arquivo que deverá conter os dados dos clientes do banco. Este arquivo deve ser fornecido pelo Componente Banco Central.
Pré-Condição:	O arquivo de conexão deve ser válido/fornecido.
Invariante:	-
Cenário:	1. O Componente Interface solicita o estabelecimento da conexão ao Componente ATM 2. O Componente ATM solicita o arquivo que contém os dados dos clientes (arquivo de conexão) ao Componente Banco Central 3. O Componente ATM verifica se o arquivo de conexão é válido (arquivo válido) e realiza a leitura do arquivo.
Pós-Condição:	O arquivo de conexão deve ser lido.

Figura 24 – Caso de Uso Conexão com Banco Central

Caso de Uso Autenticação do Usuário	
Descrição:	O componente Interface solicita ao Componente ATM a verificação dos dados do cliente (cartão e senha). O componente ATM deverá verificar se a conexão foi estabelecida e se os dados fornecidos estão de acordo com os dados existentes no arquivo fornecido pelo Banco Central durante o estabelecimento da conexão.
Pré-Condição:	Os dados do cartão devem ser válidos; A senha deve ser válida
Invariante:	A conexão deve estar estabelecida
Cenário:	1. O Componente Interface solicita a verificação dos dados do cartão do cliente ao Componente ATM 2. O Componente ATM verifica se a conexão foi estabelecida (conexão estabelecida). 3. O Componente ATM verifica se os dados do cartão são válidos (dados válidos) e retorna uma mensagem informando que os dados do cliente são válidos. 4. O Componente Interface solicita a verificação da senha do cliente ao Componente ATM 5. O Componente ATM verifica que a senha é válida (senha válida) e retorna uma mensagem informando que a senha do cliente é válida.
Pós-Condição:	O cliente poderá executar um dos tipos de transação: saque, depósito e transferência.

Figura 25 – Caso de Uso Autenticação do Usuário

Caso de Uso Saque	
Descrição:	O componente Interface solicita ao Componente ATM a operação de saque fornecendo o tipo de conta e a quantia a ser retirada. O componente ATM verifica se a conexão está estabelecida e se a autenticação dos dados do cliente foi realizada. Em seguida verifica o tipo de conta e se o valor requisitado está dentro das políticas pré-definidas de saque. Se estas verificações estiverem corretas o componente ATM atualiza os dados da conta e envia uma mensagem para o Componente Interface para que este forneça o dinheiro ao cliente.
Pré-Condição:	A autenticação do usuário já deve ter sido realizada; O cliente deve possuir o tipo de conta informado; O valor requisitado deve estar dentro das políticas de saque.
Invariante:	A conexão deve estar estabelecida
Cenário:	1. O Componente Interface solicita a operação de saque ao Componente ATM 2. O Componente ATM verifica se a conexão foi estabelecida. 3. O Componente ATM verifica se a autenticação do usuário foi realizada. 4. O Componente ATM verifica se o tipo de conta é válido. 5. O Componente ATM verifica se a quantia está dentro das políticas pré-definidas de saque 6. O Componente ATM atualiza os dados da conta e envia uma mensagem para o Componente Interface para que este forneça o dinheiro ao cliente.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 26 – Caso de Uso Saque

Passo 2: Mapeamento das tarefas

Neste passo deve-se realizar o "mapeamento" entre as tarefas identificadas nos casos de uso para as operações da interface do componente, ou seja, para cada tarefa identificada verificar se existe uma operação da interface correspondente, sendo que mais do que uma tarefa pode ser mapeada para uma mesma operação.

Para isto deve-se analisar quais são as operações que pertencem a interface do componente. A figura 27 apresenta as classes e todos os métodos públicos que compõem o componente ATM. A implementação desse componente foi realizada na linguagem C++ e considerou-se que a interface do componente fosse composta de alguns métodos públicos pertencentes às classes do componente.

Isto foi realizado pois alguns métodos públicos são utilizados somente internamente no componente, ou seja, não são invocados diretamente por outros componentes e sim utilizados pelos métodos públicos que são diretamente invocados por outros componentes (métodos da interface do componente).

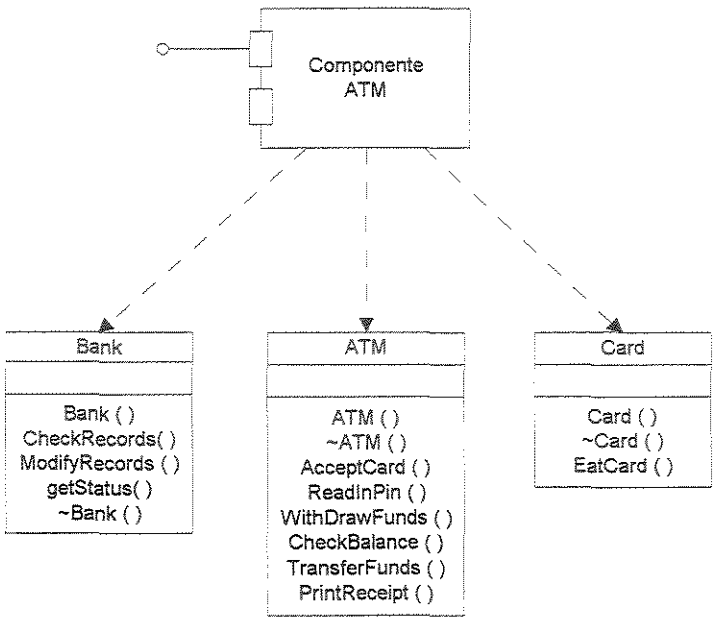

Figura 27 – Classes do Componente ATM

Para realizar este mapeamento deve-se analisar o relacionamento entre as tarefas identificadas no casos de uso (Apêndice A) e os métodos públicos do componente. A tabela 5 apresenta este relacionamento.

Caso de Uso	Tarefa	Classe: Método
Conexão com Banco Central	Solicitação do arquivo de conexão ao Componente Banco Central	Bank : Bank ( )
	Verificação e leitura do arquivo de conexão	
Autenticação do Usuário	Verificação da conexão	Bank: getStatus ( )
	Verificação dos dados do cartão	ATM : AcceptCard ( ) e Bank : CheckRecords ( )
	Verificação da senha	ATM : ReadInPin( )
	Retenção Cartão	Card : EatCard ( )
Saque	Verificação da conexão	Bank: getStatus ( )
	Verificação da autenticação do usuário	ATM : WithdrawFunds ( )
	Verificação do tipo da conta	
	Verificação do valor a ser retirado	
	Atualização dos dados da conta	Bank : ModifyRecords ( )
Saldo	Verificação da conexão	Bank: getStatus ( )
	Verificação da autenticação do usuário	ATM : CheckBalance ( )
	Verificação do tipo de conta	
	Gerar saldo da conta.	
Transferência	Verificação da conexão	Bank: getStatus ( )
	Verificação da autenticação do usuário	ATM : TransferFunds ( )
	Verificação do tipo da conta	
	Verificação do valor a ser retirado	
	Atualizar os dados da conta	Bank : ModifyRecords ( )
Gerar Recibo	Verificação da conexão	Bank: getStatus ( )
	Verificação da execução de alguma transação	ATM : PrintReceipts ( )
	Geração do recibo	

Tabela 5 – Relacionamento entre Tarefas e Métodos (Componente ATM)

Em relação à interface do componente, considerou-se que os métodos *checkRecords, modifyRecords, eatCard e getStatus* não fazem parte da interface pois eles são invocados por outros métodos já apresentados na tabela 4.

Desta forma, estes métodos não serão representados diretamente no MFT como pode ser visto na próxima Seção.

Passo 3: Construção do MFT

Com base nos dados do passo anterior e nos casos de uso do componente ATM, foi construído o MFT ilustrado na figura 28, sendo que os métodos contidos em cada nó estão descritos na tabela 6.

Em relação à simbologia utilizada para os nós N1, N4, N5, N6, N7 e N8, estes são diferenciados (representados por losangos), pois são classificados como nós de ramificação. Em um nó de ramificação uma transação de entrada apresenta várias alternativas de saída.

Tarefa	Classe : Método	Número da Tarefa (Nó)
Construtor da Classe Bank e também responsável pela leitura do arquivo que contém os dados dos clientes do banco (conexão com o banco central)	Bank : Bank ( )	N1
Destrutor da Classe Bank	Bank : ~Bank ( )	N2
Construtor da Classe ATM	ATM : ATM ( )	N3
Verificação do Cartão	ATM : AcceptCard ( )	N4
Verificação da Senha	ATM : ReadInPin ( )	N5
Saque	ATM : WithDrawFunds ( )	N6
Saldo	ATM : CheckBalance ( )	N7
Transferência	ATM : TransferFunds ( )	N8
Imprimir Recibo	ATM : PrintReceipts ( )	N9
Destrutor da Classe ATM	ATM : ~ATM ( )	N10

Tabela 6 – Relação entre as Tarefas, Métodos e Nós no MFT (Componente ATM)

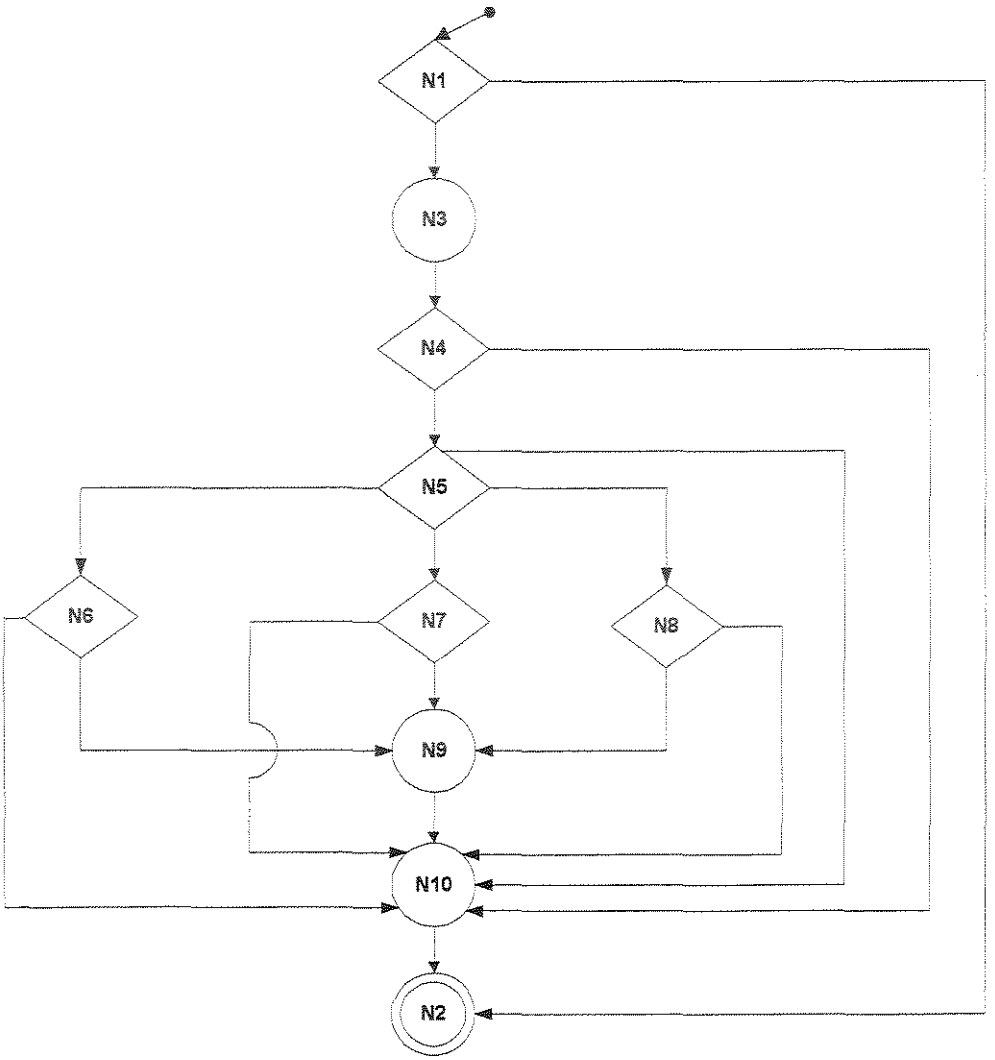

Figura 28 – MFT Componente ATM

6.1.3 Construção da Especificação de Teste

Como descrito no Capítulo 5, para construir uma especificação de teste que representasse o MFT de um componente composto por um conjunto de classes, foi utilizada a notação desenvolvida em [43] com algumas modificações.

Assim, considerou-se que todos os métodos públicos pertencessem a uma única classe. Por exemplo, considerou-se que todos os métodos públicos do componente pertencessem a classe "Bank". Dessa forma, foi possível gerar as seqüências de casos de teste automaticamente, utilizando a estratégia implementada na ferramenta ConCAT. A especificação de teste gerada para o componente ATM está ilustrada no Apêndice B.

Uma conseqüência dessa adaptação foi a necessidade de alterar os casos de teste gerados, pois, para que esses casos pudessem ser executados, a assinatura dos métodos deveria estar correta.

Para minimizar este problema, foi utilizada uma notação para distinguir os métodos pertencentes a cada classe. Assim os métodos que realmente não pertencessem à classe foram nomeados utilizando a seguinte regra: nome_classe+nome_método. Portanto, o método AcceptCard da classe ATM seria representado por "ATMAcceptCard".

O Apêndice C apresenta o driver específico gerado e também um caso de teste antes e depois da alteração da assinatura dos métodos.

6.1.4 Inserção dos Mecanismos de Teste Embutido

Nesse passo, o usuário deve adicionar ao componente os mecanismos de teste embutido que são as assertivas e o método relator. Analisando a especificação do componente (condições/cláusulas de uso - contratos), detectou-se os seguintes casos para a utilização de assertivas:

1. Pré-condição no método *Bank* para verificar se o arquivo dos dados dos clientes foi fornecido corretamente.

2. Invariante na classe ATM para verificar se a conexão foi estabelecida (verificar se o arquivo de conexão foi lido corretamente).

3. Pré-condição para os métodos *WithDrawFunds* e *TransferFunds* da classe ATM para verificar se a quantia requisitada não é superior a R$200,00.

4. Pré-condição para os métodos *WithDrawFunds*, *CheckBalance* e *TransferFunds* para verificar se o tipo de conta informado é válido e para verificar se a autenticação do usuário já foi realizada.

5. Pré-condição para o método *PrintReceipts* para verificar se alguma transação (saque, saldo e transferência) foi executada.

Neste trabalho, como descrito no Capítulo 5, as assertivas (pré e pós condições) foram definidas através de algumas macros propostas em [43]. A figura 29 ilustra a assertiva do método *WithDrawFunds* relativa ao limite de saque (R$200,00).

```
void ATM :: WithdrawFunds(int ret_val, int valor) {
        Pre_condição (valor < 200);
        ....
}
```

Figura 29 – Exemplo de utilização de Assertiva

Em relação a interface de teste, citada na Seção 5.4, a solução utilizada para adicionar os métodos relatores e os testes de invariante no componente foi a utilização de uma classe abstrata denominada Autoteste, que define uma interface padrão para os mecanismos BIT. Assim as classes pertencentes ao componente devem tornar-se classes derivadas de Autoteste.

Os métodos herdados da classe Autoteste devem ser redefinidos pelo usuário. Estes métodos possuem um formato pré-definido e podem ser estendidos para incluir outros dados que o usuário desejar. O Apêndice D apresenta o componente ATM após a inserção dos mecanismos de teste embutido.

6.2 Estudo de Caso 2 – Acervo

6.2.1 Descrição do Componente

Este componente é responsável pelo gerenciamento do acervo de uma vídeo locadora. Suas principais funcionalidade são a inserção, remoção e consulta dos dados de fitas, filmes e categorias. Outras características do seu funcionamento são:

- Uma categoria deve ser única no acervo, sendo que uma categoria pode possuir N filmes associados.

- Um filme deve ser único no acervo, sendo que um filme pode possuir N fitas associadas

- Para que um filme seja cadastrado é necessário que a categoria associada já esteja cadastrada.

- Para que uma fita seja cadastrada é necessário que o filme associado já esteja cadastrado.

- Para remover uma categoria é necessário que não exista nenhum filme associado ao mesmo.

- Para remover um filme é necessário que não exista nenhuma fita associada ao mesmo.

A implementação foi realizada na linguagem C++ e assim como na Seção 6.1.2, considerou-se que a interface fosse composta de alguns métodos públicos pertencentes as classes do componente. As próximas seções descrevem os passos executados para tornar este componente autotestável.

6.2.2 Construção do Modelo de Teste – Modelo de Fluxo de Transação

Assim como no exemplo do ATM, primeiramente será apresentada a construção do modelo de fluxo de transação para o componente Acervo. As próximas seções apresentam os passos realização para a sua construção.

Passo 1: Identificação dos casos de uso

Nesta etapa devem ser identificados os possíveis casos de uso que o componente pode realizar e as tarefas necessárias para realizá-los. Isto foi realizado através da analise da especificação considerando-se os casos de funcionamento normal e também os casos de exceção. A figura 30 apresenta os casos de uso identificados.

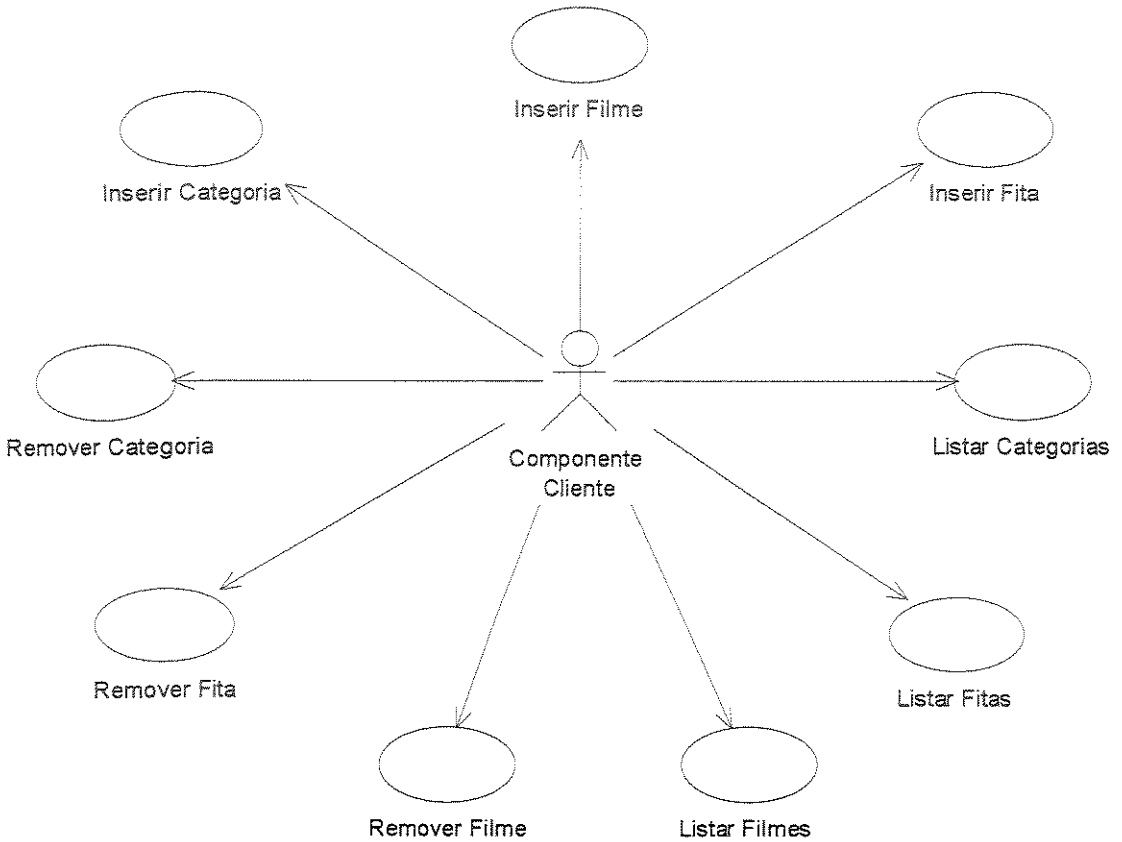

Figura 30 – Casos de Uso – Componente Acervo

Neste exemplo pode-se notar que não existe nenhuma restrição em relação ao número de inserções e remoções que podem ocorrer, sendo que isto pode ocasionar a existência de ciclos.

Segundo [4], caso os ciclos não possam ser evitados pode-se aplicar testes de ciclos (do inglês, *loop testing*) juntamente com o MFT. Todavia, isto não foi realizado neste trabalho pois o número de testes poderia ser muito extenso para ser realizado manualmente, além disso, a ferramenta utilizada na geração dos casos de teste (ConCAT) não oferece suporte para este tipo de teste.

Desta forma, os casos de uso foram elaborados de forma a não formarem ciclos. A descrição textual comportamento dos caso de uso "Inserir Filme" está descrita na Figura 31. As descrições dos demais casos de uso está no Apêndice E.

Caso de Uso Inserir Filme	
Descrição:	Inserção de um novo filme
Pré-Condição:	Categoria existente; Filme inexistente
Invariante:	-
Cenário:	1. O Componente Cliente solicita a inserção de um novo filme 2. O Componente Acervo verifica se: ▪ A categoria existe (categoria existente); ▪ O filme existe (filme inexistente) e realiza o cadastramento do filme.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 31 – Caso de Uso Inserir Filme

Passo 2: Mapeamento das tarefas

Neste passo deve-se realizar o "mapeamento" entre as tarefas identificadas nos casos de uso para os métodos da interface do componente. A figura 32 apresenta as classes e os métodos que compõe o componente Acervo.

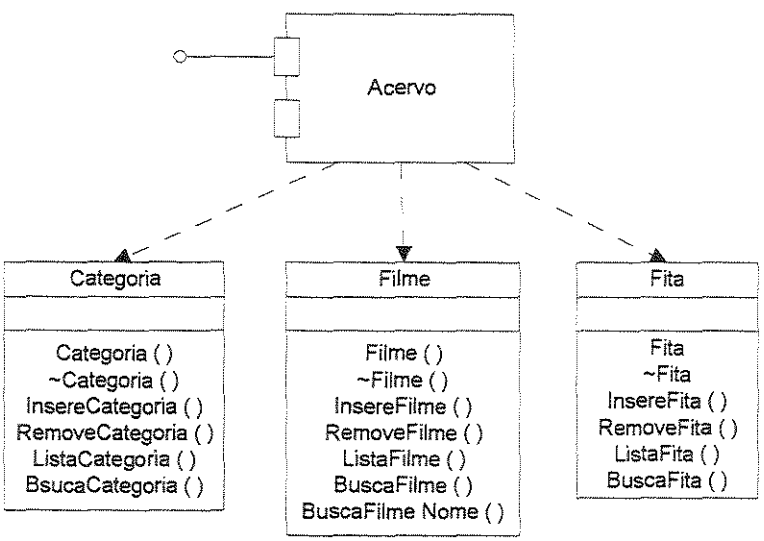

Figura 32 – Classes do Componente Acervo

Para realizar este mapeamento deve-se analisar o relacionamento entre as tarefas identificadas no casos de uso (Apêndice E) e os métodos públicos do componente. A tabela 7 apresenta este relacionamento.

Caso de Uso	Tarefa	Classe: Método
Inserir Categoria	Verificar se a categoria existe	Categoria : buscaCategoria ( )
	Cadastrar Categoria	Categoria : insereCategoria ( )
Inserir Filme	Verificar se a categoria existe	Categoria : buscaCategoria ( )
	Verificar se o filme existe	Filme: buscaFilmeNome ( )
	Cadastrar Filme	Filme: insereFilme ( )
Inserir Fita	Verificar se o filme existe	Filme: buscaFilmeNome ( )
	Cadastrar Fita	Fita : insereFita ( )
Remover Categoria	Verificar se a categoria existe	Categoria : buscaCategoria ( )
	Verificar se existem filmes associados	Filme: buscaFilme ( )
	Remover Categoria	Categoria : removeCategoria ( )
Remover Filme	Verificar se o filme existe	Filme: buscaFilmeNome ( )
	Verificar se existem fitas associadas	Fita : buscaFita ( )
	Remover Filme	Filme : removeFilme ( )
Remover Fita	Verificar se a fita existe	Fita : buscaFita ( )
	Remover Fita	Fita : removeFita ( )
Listar Categoria	Listar Categoria	Categoria : listaCategoria ( )
Listar Filme	Listar Filme	Filme : listaFilme ( )
Listar Fita	Listar Fita	Fita : listaFita ( )

Tabela 7 – Relacionamento entre Tarefas e Métodos (Componente Acervo)

Em relação a interface do componente considerou-se que somente os métodos *buscaCategoria*, *buscaFilme*, *buscaFilmeNome* e *buscaFita* não fazem parte da interface pois estes métodos são invocados por outros métodos apresentados na tabela 7.

Passo 3: Construção do MFT

Com base nos dados da tabela 7 e também nos casos de uso, foi construído o MFT ilustrado na figura 33, sendo que os métodos contidos em cada nó estão descritos na tabela 8.

Tarefa	Método:Classe Pertencente	Número da Tarefa (Nó)
Construtor da Classe Categoria	Categoria ( ) : Categoria	N1
Destrutor da Classe Categoria	~ Categoria ( ) : Categoria	N2
Inserir uma Categoria	InsereCategoria ( ) : categoria	N3
Remover uma Categoria	RemoveCategoria ( ) : categoria	N4
Listar Categorias	ListaCategoria ( ) : categoria	N5
Construtor da Classe Filme	Filme ( ) : filme	N6
Destrutor da Classe Filme	~ Filme ( ) : filme	N7
Inserir um Filme	InsereFilme ( ) : filme	N8
Remover um filme	RemoveFilme ( ) : filme	N9
Listar Filmes	ListaFilme ( ) : filme	N10
Construtor da Classe Fita	Fita ( ) : fita	N11
Destrutor da Classe Fita	~ Fita ( ) : fita	N12
Inserir uma Fita	InsereFita ( ) : fita	N13
Remover uma Fita	RemoveFita ( ) : fita	N14
Listar Fitas	ListaFita ( ) : fita	N15

Tabela 8 – Relação entre as Tarefas, Métodos e Nós no MFT (Componente Acervo)

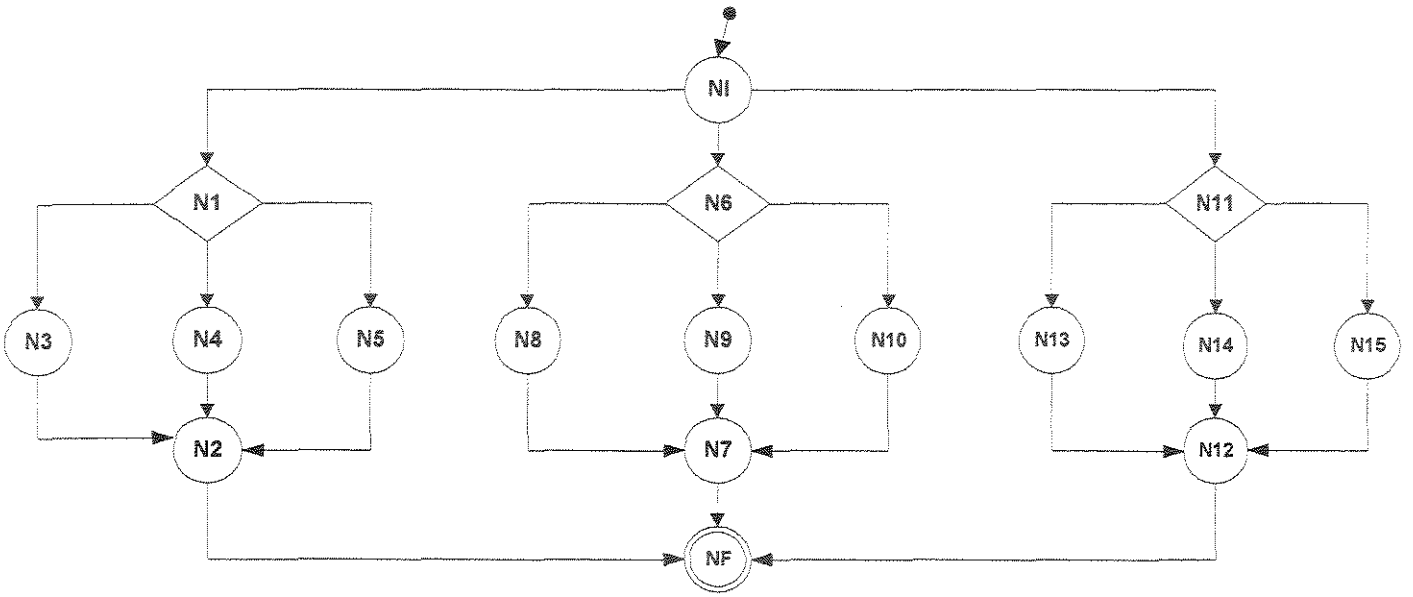

Figura 33 – MFT Acervo

Um detalhe nesta figura é a existência dos nós NI e NF. Estes nós podem ser utilizados quando existam vários pontos de entrada e saída no modelo, pois:

- No caso de vários pontos de entrada, a existência de um nó inicial NI associado aos diferentes pontos de entrada facilita os testes pois pode-se indicar ao driver específico que nesse ponto devem ser criados os objetos necessários para a execução dos testes.

- No caso de vários pontos de saída, inserir um nó final NF associado aos demais pontos de saída facilita os testes pois nesse ponto pode-se indicar ao driver específico que os objetos criados podem ser destruídos.

Desta forma garante-se que o grafo tenha somente um nó inicial e um nó final, o que facilitará a geração dos casos de teste. Portanto foram acrescentados os nós NI e NF indicando os nós inicial e final respectivamente.

6.2.3 Construção da Especificação de Teste

Assim como no exemplo anterior, a ferramenta ConCAT foi utilizada para a geração dos casos de teste. Dessa forma, os passos descritos na seção 6.1.3 para a construção da especificação de teste também foram aplicados para o componente acervo. A especificação de teste gerada para este componente está ilustrada no Apêndice F. O Apêndice G apresenta o driver específico gerado e também um caso de teste antes e depois da alteração da assinatura dos métodos.

6.2.4 Inserção dos Mecanismos de Teste Embutido

Analisando a especificação do componente (condições/cláusulas de uso), detectou-se os seguintes casos para a utilização de assertivas (contrato do componente):

- Pré-condição no método RemoveCategoria para verificar se categoria a ser removida não possui filmes associados.

- Pré-condição no método RemoveFilme para verificar se filme a ser removido não possui fitas associadas.

- Pré-condição no método InsereFilme para verificar se a categoria indicada é válida, ou seja, existente.

- Pré-condição no método InsereFita para verificar se o filme indicado é válido, ou seja, existente.

Assim como no exemplo anterior, as assertivas (pré e pós condições) foram definidas através das macros propostas em [43], o método relator e as invariantes foram inseridas através da utilização da classe Autoteste. O Apêndice H apresenta o componente Acervo após a inserção dos mecanismos de teste embutido.

6.3 Avaliação da Estratégia

6.3.1 Considerações

Após a criação da especificação de teste e da inserção dos mecanismos de teste embutido, a ferramenta ConCAT foi utilizada para a geração da sequência de teste com base na especificação de teste desenvolvida. Para que os casos de teste pudessem ser executados, foi necessário modificá-los devido à:

- Estratégia de teste de dados, pois como descrito no Capítulo 5, na implementação atual da ConCAT a estratégia de dados consiste na geração de dados aleatoriamente (cada dado é gerado independentemente do valor anterior). Consequentemente, o teste em algumas situações se tornaria difícil caso os dados não estejam corretos. Por exemplo, no caso de uso "Autenticação do usuário" é necessário que se informe a senha correta para o cartão utilizado. Assim, em alguns casos, optou-se por alterar os dados gerados pela ConCAT para melhorar os casos de teste aplicados.

- Atualização da assinatura dos métodos (problema descrito na Seção 6.1.3 – Construção da Especificação de Teste)

Além disso, para realizar o teste de algumas situações, por exemplo "Inserir Categoria (Categoria Existente)", foi necessário criar casos de teste devido problema da ocorrência de ciclos no MFT descrito na Seção 6.2.2.

6.3.2 Descrição do Experimento

Após a geração e execução dos casos de teste, realizou-se uma análise da cobertura de código obtida através dos testes gerados a partir do MFT. Isto foi realizado pois esta análise auxilia a detectar problemas como:

- A implementação de requisitos não documentados, isto é, não foram contemplados pelo modelo de teste.

- A existência de partes do código que não são executáveis.

- Avaliação da necessidade de se gerar novos casos de teste pois os testes com base nos requisitos podem não ser suficientes, uma vez que o modelo pode representá-los apenas parcialmente.

Para realizar esta medição foi utilizada a ferramenta Panorama [24]. A Panorama é uma ferramenta desenvolvida para auxiliar nos testes, avaliação e manutenção de sistemas codificados nas linguagens C e C++. Esta ferramenta realiza análise estática, obtendo métricas como complexidade ciclomática, e análise dinâmica, fornecendo a cobertura de código para os testes realizados.

O critério utilizado para a medir a quantidade de código executada foi o de cobertura de ramos (do inglês, *Branch Test Coverage*). Um ramo é definido como um trecho no programa onde existem duas ou mais alternativas de processamento. Desta forma, a cobertura é definida pela razão entre o número de alternativas (ramos) existentes e o número de ramos executados. A figura 34 apresenta um resumo das etapas realizadas neste experimento.

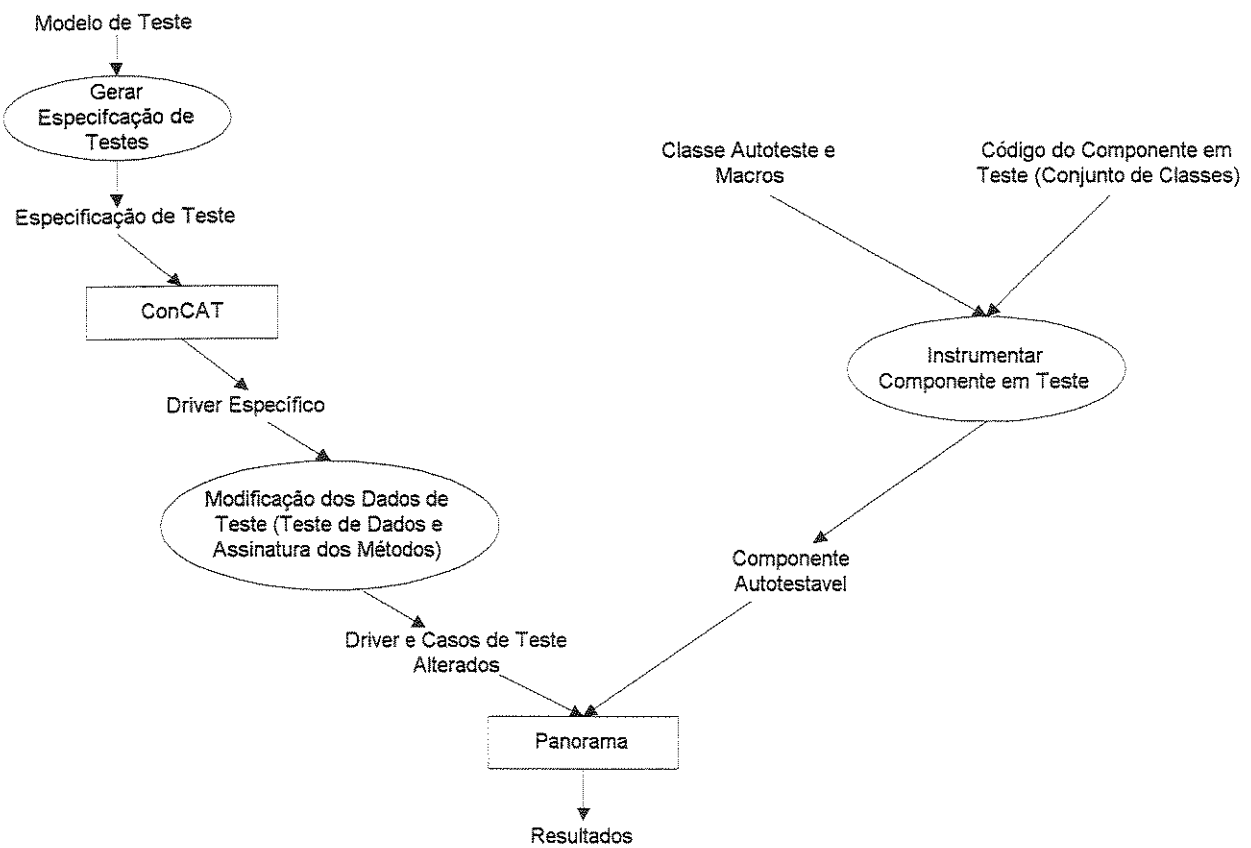

Figura 34 – Etapas do Experimento

6.3.3 Resultados Coletados e Análise dos Resultados

Esta seção apresenta os resultados coletados após a execução dos testes e também uma análise da estratégia utilizada. Os dados obtidos através da Panorama estão apresentados da seguinte forma:

- Número de ramos existentes em cada método (Interface do componente) e o percentual executado nos testes.

- Número de ramos existentes no componente (conjunto de classes, incluindo métodos públicos e privados) e o percentual executado nos testes.

As tabelas 9 e 10 apresentam o resultados dos testes para os componentes Acervo e ATM, respectivamente.

Componente Acervo:

Método	Número de Ramos	Número de Ramos Executados	Porcentagem Executada	Número de Casos de Teste
Categoria : categoria ( )	3	3	100 %	18
Filme : filme ( )	3	3	100 %	
Fita : fita ( )	3	3	100 %	
InsereCategoria ( ) : categoria	11	9	81.81 %	
RemoveCategoria ( ) : categoria	13	7	53,8 %	
ListarCategoria ( ) : categoria	11	9	81.81 %	
InsereFilme ( ) : filme	13	11	84.6 %	
RemoveFilme ( ) : filme	11	9	81.8 %	
ListarFilme ( ) : filme	9	7	77.7 %	
InsereFita ( ) : fita	11	9	81.81 %	
RemoveFita ( ) : fita	13	7	53.8 %	
ListarFita ( ) : fita	9	7	73.7 %	
Componente	154	124	80.5 %	

Tabela 9 - Resumo dos Testes – Componente Acervo

Componente ATM:

Método	Número de Ramos	Número de Ramos Executados	Porcentagem Executada	Número de Casos de Teste
Bank : Bank ( )	3	3	100 %	20
ATM : ATM ( )	1	1	100 %	
ATM : AcceptCard ( )	7	6	85.7 %	
ATM : ReadInPin ( )	3	3	100 %	
ATM : WithDrawFunds ( )	11	9	81.81 %	
ATM : CheckBalance ( )	5	4	80 %	
ATM :TransferFunds ( )	11	9	81.81 %	
ATM : PrintReceipt ( )	5	4	80% %	
Componente	92	69	75 %	

Tabela 10 – Resumo dos Testes – Componente ATM

Em relação ao número de falhas detectadas, no componente ATM ocorreu a violação das assertivas definidas para a pré-condição dos métodos WithDrawFunds e TransferFunds da classe ATM, utilizados para verificar se a quantia requisitada não é superior a R$200,00, sendo que o componente não apresentava tratamento para esta situação. Nenhuma outra falha foi detectada nos testes, isto talvez porque os componentes já tenham sido previamente testados.

6.3.4 Dificuldades e Vantagens

Nesta seção serão apresentados as vantagens da aplicação da estratégia proposta e também as dificuldades encontradas para a sua aplicação.

Dificuldades Encontradas:

- O modelo de fluxo de transação (versão descrita em [4]) se mostrou "semi-formal", pois verificou-se que a sua semântica está mal definida. Isto foi relatado pelo próprio autor do modelo através de informações trocadas em [6].

- Para facilitar a construção e uso de componentes autotestáveis é importante um suporte automatizado à estratégia proposta. Como pode ser visto durante o trabalho, foi necessário realizar intervenções manuais para que a estratégia proposta fosse utilizada, sendo elas:

- Criação da especificação de teste
- Modificação dos casos de teste:
  - Teste de dados (geração de parâmetros)
  - Assinatura dos métodos
- Inserção de assertivas e métodos relatores

Vantagens:

Com esta estratégia é possível melhorar a testabilidade de um componente da seguinte forma:

- Aumento da observabilidade e controlabilidade através do uso de mecanismos como assertivas e métodos relatores.
- Melhoria da documentação do componente, pois com a especificação de testes fazendo parte do componente tem-se uma descrição legível do que o componente faz embutida no mesmo.
- Facilidade maior para geração de testes, pois a especificação de testes presente no componente vai permitir gerar testes de forma automática.
- Facilidade na manutenção do componente, pois com a especificação presente é possível alterá-la juntamente com o código do componente.
- A inserção de assertivas no sistema podem servir de oráculo parcial nos testes.

6.3.5 Comparação com os Trabalhos Relacionados

Esta Seção apresenta uma comparação dos trabalhos descritos Seção 4.5, incluindo o trabalho apresentado nesta dissertação. A tabela 11 apresenta as propriedades que afetam a testabilidade (capacidade teste embutido, geração de teste, reutilização dos casos de teste e ferramentas de teste) descritas na Seção 4.2 e também nos principais objetivos de cada trabalho.

Trabalhos	Capacidade de teste embutido	Geração de testes	Reutilização dos casos de teste	Ferramentas de teste	Contribuição para questões de teste
Jeffrey Voas et al	Assertivas	---	---	*PiSCES* e Assert ++	Indicam que o uso de assertivas é útil para melhorar a testabilidade e fornecem ferramentas que auxiliam na inserção de assertivas
Yingxy Wang et al	Casos de teste embutidos diretamente no código fonte	---	Os casos de teste podem ser herdados e reutilizados da mesma forma que o código fonte	---	Mecanismos de teste são embutidos no componente, facilitando o reuso dos mesmos
Yves Le Traon et al	Rotinas de teste no código fonte	Propõe alguns critérios para teste de integração e regressão mas os testes não são gerados automaticamente	Os casos de teste podem ser herdados e reutilizados da mesma forma que o código fonte	---	Apresenta uma abordagem para o uso do autoteste (casos de teste embutidos no componente) e também na análise de mutantes como métrica de qualidade
Este Trabalho	Modelo de comportamento (MFT) + Assertivas (Contrato) + Métodos Relatores	Utiliza critérios baseados no Modelo de Fluxo de Transação e teste de dados	Reutiliza os casos de teste com base na ConCAT. Os caminhos no MFT que contenham somente métodos não modificados não são gerados outra vez (reutiliza casos de teste)	ConCAT	Estratégia para a construção de componentes de software autotestáveis na qual a especificação de teste, assertivas e métodos relatores são embutidos no mesmo, ao invés dos casos de teste

Tabela 11– Comparação dos Trabalhos

6.4 Considerações Finais

Este capítulo apresentou estudos de caso realizados para ilustrar a estratégia proposta e avaliar o quanto esta pode auxiliar na realização de testes de boa qualidade. Além disso também foi apresentada uma análise das dificuldades e vantagens da utilização da estratégia. O próximo Capítulo descreve alguns trabalhos correlatos que abordam problemas relacionados com a melhoria da testabilidade.

Capítulo 7

Conclusões

Neste trabalho foi apresentado o uso do conceito de autoteste, aplicado inicialmente em circuitos integrados, para a melhoria da testabilidade de componentes de software compostos por um conjunto de classes. Este trabalho é uma evolução da pesquisa realizada por [43], em que o conceito de autoteste foi utilizado para a construção componentes autotestáveis constituídos de uma única classe.

Para isso foi proposta uma estratégia que mostra os passos que o produtor de um componente deve realizar para construir componentes que possuam mecanismos de teste embutidos que o consumidor (usuário do componente) possa utilizar para determinar se o componente provê as funcionalidades desejadas.

Desta forma, foi proposto que um componente autotestável deverá possuir uma interface de teste. Esta interface representa as facilidades que o componente apresenta para a realização dos testes, sendo composta pelos mecanismos de teste embutidos (assertivas e métodos relatores). Além disso, propõe-se que uma especificação do componente (modelo de teste e especificação de teste) seja criada e fornecida juntamente com o componente, afim de facilitar a geração dos casos de teste e descrição de suas funcionalidades. No capítulo 6 foram apresentados dois estudos de caso ilustrando a aplicação da estratégia proposta para a melhoria da testabilidade de componentes de software.

Durante o desenvolvimento deste trabalho, observou-se as seguintes contribuições:

- Um estudo sobre o modelo de fluxo de transação e de como utilizá-lo na integração de vários objetos.

- Um estudo sobre o desenvolvimento baseado em componentes e sua relação com a atividade de testes.

- Um estudo sobre testabilidade de software englobando aspectos como as características que afetam a testabilidade de um componente de software e trabalhos relacionados com a melhoria da testabilidade de software.

- Proposta de uma estratégia para a construção de componentes de software autotestáveis através da extensão do conceito de autoteste aplicado a classes.

- Realização de estudos de caso para ilustrar a estratégia proposta e avaliar as dificuldades e vantagens de sua utilização.

Pode-se destacar algumas linhas de pesquisa a partir do trabalho apresentado nessa dissertação:

- Umas dessas linhas envolve a continuação dos experimentos para a avaliação da estratégia proposta, principalmente utilizando componentes compostos por um número maior de classes.

- Construção de uma infra-estrutura que ofereça suporte automatizado à estratégia proposta, sendo composta, ao menos de:

    1. Uma ferramenta para auxiliar a criação da especificação de teste.

    2. Um gerador de casos de teste que seja capaz ler a especificação de teste e gerar casos de teste incluindo uma estratégia de teste de dados.

    3. Uma ferramenta para auxiliar a inserção dos mecanismos de teste embutido.

    4. Um oráculo para que os testes possam ser avaliados.

    5. Um histórico de testes para armazenar os casos de teste, para que alguns destes possam ser reutilizados quando o componente for adaptado.

- Utilização de outros métodos ou ferramentas para a inserção de assertivas, como por exemplo JML, pois como mencionado anteriormente as assertivas podem servir como oráculo durante os testes.

Bibliografia

[1] American National Standards Institute. *Dictionary for Information Systems*, 1991.

[2] Bachman, F. & et al. *Technical Concepts of Component Based Software Engineering, CMU/SEI-2000-TR-008.* Obtido na Internet URL: www.sei.cmu.edu/ publications/documents/00.reports/ 00tr008.html, 2000.

[3] Barn B. & Brown A.W., *Improving Software Reuse Through Component-Based Development Tools*, Obtido na Internet URL: www.jorvik.com/alan/icsr5/index.html, 1997.

[4] Beizer, B. *Software Testing Techniques*. International Thomson Computer Press 1990.

[5] Beizer, B. *Black-Box Testing*. John Wiley & Sons 1995.

[6] Beizer, B. Informações obtidas através de troca de mensagens, 2000.

[7] Binder, R. V. *Design for Testability in Object-Oriented Systems. Communications of the ACM*, Setembro/Vol. 37, N°9, 1994.

[8] Binder, R.V. *Testing Object-Oriented Software: A Status Report.* Obtido na Internet URL: www.rbsc.com, 1999.

[9] Binder, R.V. *The FREE Approach to Testing Object-Oriented Software: An Overview*. Obtido na Internet URL: www.rbsc.com,1999.

[10] Binder, R.V. *Testing Object-Oriented Systems. Models, Patterns and Tools* - Addison-Wesley 2000.

[11] Booch G., Rumbaugh J. & Jacobson I.. *The Unified Modeling Language User Guide*. Addsion-Wesley, 1999.

[12] Bosch, J. & Szyperski, C. & Weck, W. *Summary of the Second International Workshop on Component-Oriented Programming (WCOP'97)*. Jyväskylä, Finland, Junho 1997.

[13] Brown A.W., *The Current State of CBSE. IEEE Software*, pages 37-46, September/Outubro 1998.

[14] Chessman J. & Daniels J. *Uml Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, Outubro 2000.

[15] Councill, W.T. *Third-Party Testing and the Quality of Software Components. IEEE Software*, July, Vol. 16, N° 4, 1999.

[16] DeMillo R. A. & Lipton R. J. & Sayward F. G. *Hints on test data selection: help for the practicing programmer*, IEEE Computer, Vol. 11 Number 4 Pages 34-41 April, 1978.

[17] Deveaux D. & Jézequel J. M. *Des classes Auto-testables*, Obtido na Internet URL: www.univ-ubs.fr/valoria/aglae/publis/pdf/agl1999-1.pdf, 2000

[18] D'Souza, D. & Wills, A. C. *Components and Frameworks with UML – The Catalysis Approach*. Addison-Wesley Publishing Company,1997.

[19] Freedman, R. S. *Testability of Software Components*. IEEE Trans. On Software Engineering, June, Vol. 17, N° 6, 1991.

[20] Fowler, M. & Scott K., *UML Distilled*, Addison-Wesley,1997.

[21] Gao, J. Testing *Component Testability and Component Testing Challenges* 2000. Obtido na Internet URL: *www.sei.cmu.edu/cbs/cbse2000/papers/18/18.html*, 2000.

[22] Harrold, M.J. & McGregor J. D. *Incremental Testing of Object-Oriented Class Structures*. Clemson Universtiy, 1992.

[23] Hoffman, D.M. *Hardware Testing and Software ICS*, Proceedings of Pacific NW Software Quality Conference, Portland, Oregon, EUA, Setembro 1989.

[24] Informações Obtidas na Internet URL: www.softwareautomation.com, 2000.

[25] JavaSoft SUN Micorsystems. *The JavaBeans Component Architecture*. Obtido na Internet URL: http://www.java.sun.com/beans, 2000.

[26] Krajnc M. *What is Component-Oriented Programming*. Oberon Microsystems, Janeiro, 1997.

[27] Leavens, G. T. & Baker A. L. & Ruby C. *JML: Java Modeling Langague*. Obtido na Internet URL:www.cs.iastate.edu/~leavens/JML.html, 2001.

[28] Leite, J.C.B & Loques, O. L. *Introdução à Tolerância a Falhas* – Capítulo 4. II Simpósio de Computação Tolerante a Falhas, Campinas, 1987.

[29] Martins, E. *Injeção de falhas por software na validação da segurança no funcionamento*. I Simpósio Regional de Tolerância a Falhas, Porto Alegre, 1996.

[30] Martins, E. *Verificação e Validação de Software*. Instituto de Computação – Unicamp. Abril 1999.

[31] Microsoft Corporation. *DCOM – The Distributed Component Object Model*. Obtido na Internet URL: www.microsoft.com/dcom, 2000.

[32] Myers, G. J. *The Art of Software Testing*. John-Wiley & Sons, 1979.

[33] OMG Object Management Group. *CORBA – The Common Object Request Broker Architecture*. Obtida na Internet URL: www.corba.org, 2000

[34] Pigorski, T. M. *Pratical Software Maintenance, Chapter 16*. John Wiley & Sons, 1997.

[35] Pressman, R.S. *Software Engineering – A practitioner's approach. McGraw-Hill, 4ª edition*, 1997.

[36] Reliable Software Technologies Corporation (RSTC), *Testability of Object-Oriented Systems Technical Report*, Dezembro, 1994.

[37] Rubira, C & Buzato, L. *Construção de Sistemas Orientados a Objetos Confiáveis*. XI Escola de Computação, Rio de Janeiro, Brasil, Julho 1998.

[38] Schneider, G. & Winters J. P. *Applying Use Cases: A Practical Guide*, Addison-Wesley 2000.

[39] Siegel, S. *Object Oriented Software Testing: a Hierarquical Approach*, John Wiley & Sons, 1ª edição, New York, 1996.

[40] Sterling Software, *Component Based Development and Object Modeling - Categorizing&Deploying Components*, Obtido na Internet URL: www.cool.sterling.com/cbd/whitepaper/5.html, 1997

[41] Szypersky, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Publishing Company, ACM Press, New York 1998.

[42] Traon L. Y. & Deveaux, D. & Jezequel J., *Self-testable components: from pragmatic tests to design-for-testability methodology*, Obtido na Internet URL: www.iro.montreal.cal~devezux/ publis/PS/Tools-99.ps.g, 2000.

[43] Toyota, C. M. & Martins, E. *Construção de Classes Autotestáveis*. VIII Simpósio de Computação Tolerante a Falhas, Campinas 1999.

[44] Voas, J. *PIE: A Dinamic Failure-Based Technique*, IEEE Trans. On Software Engineering, Agosto, 1992.

[45] Voas, J.& Miller K. *Software Testability: The New Verification*, IEEE Software, Vol. 12, Maio, 1995

[46] Voas, J. & Schmid, M. & Schatz, M. *A Testability-Based Assertion Placement Tool for Object-Oriented Software*. Department of Commerce – USA (NIST GCR 98-735), Janeiro 1998.

[47] Wang, Y. & King, G. & Court, I. & Ross, M. & Staples G. *On Built-in Tests in Object-Oriented Reegineering*. Obtido na Internet URL: www. iamwww.unibe.ch/~famoos/ESEC97/submissions/wang1.pdf, 1999.

[48] Wang Y. & King, G. & Wickburg H., *A Method for Built-in Tests in Component-based Software Maintenance*, IEEE International Conference on Software Maintenance and Reengineering, 1999

[49] Weyuker, E. J. *Testing Component-Based Software: A Cautionary Tale. Communications of the ACM*, September/Vol. 37, N°9, 1998.

[50] Weiss, G. M., *Adaptação de Componentes de Software para o Desenvolvimento de Sistemas Confiáveis*. Dissertação de Mestrado – Instituto de Computação - Unicamp, 2001.

Apêndice A

Casos de Uso – Componente ATM

Neste Apêndice são apresentados os casos de uso identificados para o Componente ATM, considerando-se os casos de funcionamento normal e também os de exceção. A seguir temos a descrição de todos os casos de uso identificados.

Caso de Uso Conexão com Banco Central	
Descrição:	O Componente Interface solicita ao Componente ATM para que a conexão com o Banco Central seja estabelecida. Para simular esta conexão o Componente ATM realiza a leitura de um arquivo que deverá conter os dados dos clientes do banco. Este arquivo deve ser fornecido pelo Componente Banco Central.
Pré-Condição:	O arquivo de conexão deve ser válido/fornecido.
Invariante:	-
Cenário:	1. O Componente Interface solicita o estabelecimento da conexão ao Componente ATM 2. O Componente ATM solicita o arquivo que contém os dados dos clientes (arquivo de conexão) ao Componente Banco Central 3. O Componente ATM verifica se o arquivo de conexão é válido (arquivo válido) e realiza a leitura do arquivo.
Pós-Condição:	O arquivo de conexão deve ser lido.

Figura 35 - Caso de Uso Conexão com Banco Central – Cenário Normal

Caso de Uso Conexão com Banco Central	
Cenário Excepcional: Violação de Pré-Condição	Durante o processo de conexão com o Banco Central verifica-se que o arquivo que contém os dados dos clientes do banco fornecido pelo Componente Banco Central é inválido ou não foi fornecido.
Pós-Condição:	O arquivo de conexão não deve ser lido.

Figura 36 - Caso de Uso Conexão com Banco Central – Cenário Excepcional

Caso de Uso Autenticação do Usuário	
Descrição:	O componente Interface solicita ao Componente ATM a verificação dos dados do cliente (cartão e senha). O componente ATM deverá verificar se a conexão foi estabelecida e se os dados fornecidos estão de acordo com os dados existentes no arquivo fornecido pelo Banco Central durante o estabelecimento da conexão.
Pré-Condição:	Os dados do cartão devem ser válidos; A senha deve ser válida
Invariante:	A conexão deve estar estabelecida
Cenário:	1. O Componente Interface solicita a verificação dos dados do cartão do cliente ao Componente ATM 2. O Componente ATM verifica se a conexão foi estabelecida (conexão estabelecida). 3. O Componente ATM verifica se os dados do cartão são válidos (dados válidos) e retorna uma mensagem informando que os dados do cliente são válidos. 4. O Componente Interface solicita a verificação da senha do cliente ao Componente ATM 5. O Componente ATM verifica que a senha é válida (senha válida) e retorna uma mensagem informando que a senha do cliente é válida.
Pós-Condição:	O cliente poderá executar um dos tipos de transação: saque, depósito e transferência.

Figura 37- Caso de Uso Autenticação do Usuário – Cenário Normal

Caso de Uso Autenticação do Usuário	
Cenário Excepcional 1: Violação de Pré-Condição	Durante o processo de autenticação do usuário o componente ATM verifica que os dados do cartão são inválidas. Neste caso o cartão é retido e uma mensagem de erro é gerada.
Pós-Condição:	O cartão deve ser retido.
Cenário Excepcional 2: Violação de Pré-Condição	Durante o processo de autenticação do usuário o componente ATM verifica que a senha é inválida. . Neste caso uma mensagem de erro é gerada.
Pós-Condição:	Geração de uma mensagem de erro indicando que a senha é inválida
Cenário Excepcional 3: Violação de Invariante	Durante o processo de autenticação do usuário o componente ATM verifica que os dados do cartão são inválidas. Neste caso o cartão é retido e uma mensagem de erro é gerada.
Pós-Condição:	Geração de uma mensagem de erro indicando que a conexão não foi estabelecida.

Figura 38 - Caso de Uso Autenticação do Usuário – Cenários Excepcionais

Caso de Uso Saque	
Descrição:	O componente Interface solicita ao Componente ATM a operação de saque fornecendo o tipo de conta e a quantia a ser retirada. O componente ATM verifica se a conexão está estabelecida e se a autenticação dos dados do cliente foi realizada. Em seguida verifica o tipo de conta e se o valor requisitado está dentro das políticas pré-definidas de saque. Se estas verificações estiverem corretas o componente ATM atualiza os dados da conta e envia uma mensagem para o Componente Interface para que este forneça o dinheiro ao cliente.
Pré-Condição:	A autenticação do usuário já deve ter sido realizada; O cliente deve possuir o tipo de conta informado; O valor requisitado deve estar dentro das políticas de saque.
Invariante:	A conexão deve estar estabelecida
Cenário:	1. O Componente Interface solicita a operação de saque ao Componente ATM 2. O Componente ATM verifica se a conexão foi estabelecida. 3. O Componente ATM verifica se a autenticação do usuário foi realizada. 4. O Componente ATM verifica se o tipo de conta é válido. 5. O Componente ATM verifica se a quantia está dentro das políticas pré-definidas de saque 6. O Componente ATM atualiza os dados da conta e envia uma mensagem para o Componente Interface para que este forneça o dinheiro ao cliente.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 39 - Caso de Uso Saque – Cenário Normal

Caso de Uso Saque	
Cenário Excepcional 1: Violação de Invariante	Durante o processo de saque o componente ATM constata que a conexão não está estabelecida.
Pós-Condição:	Geração de uma mensagem de erro.
Cenário Excepcional 2: Violação de Pré-Condição	Durante o processo de saque o componente ATM constata que a autenticação do usuário não foi realizada
Pós-Condição:	Geração de uma mensagem de erro
Cenário Excepcional 3: Violação de Pré-Condição	Durante o processo de saque o componente ATM constata que o tipo de conta informado é inválido.
Pós-Condição:	Geração de uma mensagem de erro.
Cenário Excepcional 4: Violação de Pré-Condição	Durante o processo de saque o componente ATM verifica que o valor requisitado é maior o que o saldo da conta ou superior a R$200.00
Pós-Condição:	Geração de uma mensagem de erro.

Figura 40 - Caso de Uso Saque – Cenários Excepcionais

Caso de Uso Saldo	
Descrição:	O componente Interface solicita ao Componente ATM a operação de saldo fornecendo o tipo de conta. O componente ATM verifica se a conexão está estabelecida e se a autenticação dos dados do cliente foi realizada. Em seguida verifica se o cliente possui o tipo de conta informado. Se estas informações estiverem corretas o componente ATM envia uma mensagem para o Componente Interface informando o saldo da conta.
Pré-Condição:	A autenticação do usuário já deve ter sido realizada; O cliente deve possuir o tipo de conta informado
Invariante:	A conexão deve estar estabelecida
Cenário:	1. O Componente Interface solicita a operação de saldo ao Componente ATM 2. O Componente ATM verifica se a conexão foi estabelecida. 3. O Componente ATM verifica se a autenticação do usuário foi realizada. 4. O Componente ATM verifica se o tipo de conta é válido. 5. O Componente ATM envia uma mensagem para o Componente Interface informando o saldo da conta.
Pós-Condição:	Geração de uma mensagem indicando o saldo da conta

Figura 41- Caso de Uso Saldo – Cenário Normal

Caso de Uso Saldo	
Cenário Excepcional 1: Violação de Invariante	Durante o processo de saque o componente ATM constata que a conexão não está estabelecida.
Pós-Condição:	Geração de uma mensagem de erro.
Cenário Excepcional 2: Violação de Pré-Condição	Durante o processo de saque o componente ATM constata que a autenticação do usuário não foi realizada
Pós-Condição:	Geração de uma mensagem de erro
Cenário Excepcional 3: Violação de Pré-Condição	Durante o processo de verificação do saldo o componente ATM constata que o tipo de conta informado é inválido.
Pós-Condição:	Geração de uma mensagem de erro.

Figura 42 - - Caso de Uso Saldo – Cenários Excepcionais

Caso de Uso Transferência	
Descrição:	O componente Interface solicita ao Componente ATM a operação de transferência fornecendo o tipo de conta e a quantia a ser transferida. O componente ATM verifica se a conexão está estabelecida e se a autenticação dos dados do cliente foi realizada. Em seguida verifica o tipo de conta e se o valor requisitado está dentro das políticas pré-definidas de saque. Se estas verificações estiverem corretas o componente ATM atualiza os dados das contas e envia uma mensagem para o Componente Interface indicando que a operação foi realizada.
Pré-Condição:	A autenticação do usuário já deve ter sido realizada; O cliente deve possuir o tipo de conta informado; O valor requisitado deve estar dentro das políticas de saque.
Invariante:	A conexão deve estar estabelecida
Cenário:	1. O Componente Interface solicita a operação de transferência ao Componente ATM 2. O Componente ATM verifica se a conexão foi estabelecida. 3. O Componente ATM verifica se a autenticação do usuário foi realizada. 4. O Componente ATM verifica o tipo de conta informado. 5. O Componente ATM verifica se a quantia está dentro das políticas pré-definidas de transferência 6. O Componente ATM atualiza os dados das contas e envia uma mensagem para o Componente Interface indicando que a operação foi realizada.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 43 - Caso de Uso Transferência – Cenário Normal

Caso de Uso Transferência	
Cenário Excepcional 1: Violação de Invariante	Durante o processo de saque o componente ATM constata que a conexão não está estabelecida.
Pós-Condição:	Geração de uma mensagem de erro.
Cenário Excepcional 2: Violação de Pré-Condição	Durante o processo de saque o componente ATM constata que a autenticação do usuário não foi realizada
Pós-Condição:	Geração de uma mensagem de erro
Cenário Excepcional 3: Violação de Pré-Condição	Durante o processo de transferência o componente ATM constata que o cliente não possui o tipo de conta informado.
Pós-Condição:	Geração de uma mensagem de erro.
Cenário Excepcional 4: Violação de Pré-Condição	Durante o processo de transferência o componente ATM verifica que o valor indicado é maior o que o saldo da conta ou superior a R$200,00
Pós-Condição:	Geração de uma mensagem de erro.

Figura 44 - Caso de Uso Transferência - Cenários Excepcionais

Apêndice B

Especificação de Teste – Componente ATM

A figura a seguir a apresenta a especificação de teste gerada para o componente ATM.

```
classe('Bank',n,_,[]).

atributo ('fp', ponteiro, 'File',_).
atributo ('RecordNumber', intervalo, 1,1000).
atributo ('Database', ponteiro, 'char',_).
atributo ('Status', intervalo, 0,1).
atributo ('_stub', objeto, 'stub',_).

metodo(m1,'Bank',_,novo,0).
metodo(m2,'~Bank',_,novo,0).
metodo(m3,'ATMATM',_,novo,0).
metodo(m4,'ATMAcceptCard','void',novo,1).
metodo(m5,'ATMReadInPin','void',novo,1).
metodo(m6,'ATMWithdrawFunds','void',novo,2).
metodo(m7,'ATMCheckBalance','void',novo,1).
metodo(m8,'ATMTransferFunds','void',novo,2).
metodo(m9,'ATMPrintReceipt','void',novo,0).
metodo(m10,'ATM~ATM',_,novo,0).

parametro('Records','m4',string,['ukuma.txt','weiss.txt','braga.txt','uber.txt','saito.txt','fontanini.txt'],_).
parametro('pin','m5',intervalo,1,1000).
parametro('ret_val','m6',intervalo,1,10).
parametro('valor','m6',intervalo,1,200).
parametro('ret_val','m7',intervalo,1,10).
parametro('ret_val','m8',intervalo,1,10).
parametro('valor','m8',intervalo,1,200).

no(n1,sim,2,[m1]).
no(n2,nao,0,[m2]).
no(n3,nao,1,[m3]).
no(n4,nao,2,[m4]).
no(n5,nao,4,[m5]).
no(n6,nao,2,[m6]).
no(n7,nao,2,[m7]).
no(n8,nao,2,[m8]).
no(n9,nao,1,[m9]).
no(n10,nao,1,[m10]).
```

```
arco(n1,n3).
arco(n3,n4).
arco(n1,n2).
arco(n4,n5).
arco(n4,n10).
arco(n5,n6).
arco(n5,n7).
arco(n5,n8).
arco(n5,n10).
arco(n6,n9).
arco(n6,n10).
arco(n7,n9).
arco(n7,n10).
arco(n8,n9).
arco(n8,n10).
arco(n9,n10)
arco (n10,n2).
```

Figura 45 – Especificação de Teste – Componente ATM

Apêndice C

Driver Específico e Caso de Teste – Componente ATM

Neste Apêndice são apresentados alguns dados gerados pela ConCAT após a leitura da especificação de teste. A figura 34 apresenta o driver específico gerado.

```
#include <fstream.h>
#include <iostream.h>
#include "bank.cpp"

#include "Bank_ct0.cc"

#define TESTE

void main() {
  Bank* obj1 = new Bank;
  Caso_teste1_0(obj1);
  Bank* obj2 = new Bank;
  Caso_teste2_0(obj2);
  Bank* obj3 = new Bank;
  Caso_teste3_0(obj3);
  Bank* obj4 = new Bank;
  Caso_teste4_0(obj4);
  Bank* obj5 = new Bank;
  Caso_teste5_0(obj5);
  Bank* obj6 = new Bank;
  Caso_teste6_0(obj6);
  Bank* obj7 = new Bank;
  Caso_teste7_0(obj7);
  Bank* obj8 = new Bank;
  Caso_teste8_0(obj8);
  Bank* obj9 = new Bank;
  Caso_teste9_0(obj9);
}
```

Figura 46 – Driver Gerado para o Componente ATM

As figuras 35 e 36 apresentam um caso de teste gerado antes e após a modificação da assinatura dos métodos, respectivamente.

```
template <class Tipo>
void Caso_teste1_0(Tipo* cst) {
  char* met= new char[30];
  ofstream arq("Result.txt",ios::app);
  if (! arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Arquivo criado! \n";
  try {
    cst->Testa_Invariante();
    met = "ATMATM()";
    cst->ATMATM();
    cst->Testa_Invariante();
    met = "ATMAcceptCard(\"braga.txt\")";
    cst->ATMAcceptCard("braga.txt");
    cst->Testa_Invariante();
    met = "ATMReadInPin(\"30\")";
    cst->ATMReadInPin(30);
    cst->Testa_Invariante();
    met = "ATMWithdrawFunds(1,10)";
    cst->ATMWithdrawFunds(1,10);
    cst->Testa_Invariante();
    met = "ATMPrintReceipts()";
    cst->ATMPrintReceipts();
    cst->Testa_Invariante();
    met = "ATM~ATM()";
    cst->ATM~ATM();
    cst->Testa_Invariante();
    met = "~Bank()";
    cst->~Bank();
    cst->Testa_Invariante();
    arq << "Caso_teste1_0 OK!\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
    delete cst;
  }

  catch(char* c) {
    arq << "Caso_teste1_0 \n";
    arq.flush();
    cout << "...";
    arq << c << "\n";
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
  }
```

```
  catch(...) {
    arq.close();
    }
}
```

Figura 47 – Caso de Teste gerado

```
template <class Tipo>
void Caso_teste1_0(Tipo* cst) {
  char* met= new char[30];
  ofstream arq("Result.txt",ios::app);
  if (! Arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Arquivo criado! \n";
  try {
    cst->Testa_Invariante();
    met = "ATMATM()";
    ATM* atm = new ATM (cst);
    atm->Testa_Invariante();
    met = "ATMAcceptCard(\"braga.txt\")";
    atm->AcceptCard("braga.txt");
    atm->Testa_Invariante();
    met = "ATMReadInPin(\"30\")";
    atm->ReadInPin(30);
    atm->Testa_Invariante();
    met = "ATMWithdrawFunds(1,10)";
    atm->WithdrawFunds(1,10);
    atm->Testa_Invariante();
    met = "ATMPrintReceipts()";
    atm->PrintReceipts();
    atm->Testa_Invariante();
    met = "ATM~ATM()";
    atm->~ATM();
    cst->Testa_Invariante();
    met = "~Bank()";
    cst->~Bank();
    cst->Testa_Invariante();
    arq << "Caso_teste1_0 OK!\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
    delete cst;
  }

  catch(char* c) {
    arq << "Caso_teste1_0 \n";
    arq.flush();
    cout << "...";
    arq << c << "\n";
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator("Result.txt");
```

```
    arq << "\n";
    arq.close();
  }

  catch(...) {
    arq.close();
    }
}
```

Figura 48 – Caso de Teste Após a Modificação da Assinatura dos Métodos

Apêndice D

Código - Componente ATM

Neste Apêndice é apresentado o código fonte do Componente ATM após a inserção dos métodos de teste embutido.

```
#include <stdio.h>
#include <string.h>
#include <fstream.h>
#include <stdlib.h>
#include <process.h>
#include "Autoteste.h"
#include "stub.h"

#define CHECKING  1
#define SAVINGS   2
#define WITHDRAW  3
#define TRANSFER  4
#define BALANCE   5
#define  ERROR    -1


struct Account {
  int AccountNo;
  int Balance;
};

struct CustomerRecord {
  char lastname[32];
  char firstname[32];
  char middle[32];
  int pin;
  unsigned int Accounts;
  struct Account CustomerAccounts[2];
};

class Bank:Autoteste{
  private:
    FILE *fp;
    int RecordNumber;
    struct CustomerRecord BankRecords[30];
    char *Database;
```

```
 int Status; //variavel que controla o estado da leitura do arquivo de conexao:
             //Status=0 -> Falha na leitura; Status=1 -> Leitura realizada com sucesso
 stub _stub; //Stub gerado para simular o Componente Banco Central
 public:
  Bank();
  ~Bank();
  void Testa_Invariante ();
  void Relator (char* arquivo);
  int CheckRecords (struct CustomerRecord &);
  int ModifyRecords (struct CustomerRecord &, int ret_val, int valor, int transaction);
  int getStatus ();
};

class Card {
 private:
  char _Card[32];
 public:
  Card () {}
  ~Card () {}
  void EatCard();
};

class ATM : Autoteste {
 private:
  struct CustomerRecord CurrentRecord;
  Bank *bank_link;
  FILE *fp;
  Card card;
  Int transactionType;  //indica o tipo de operação:saque/saldo/transferencia que foi executado
  Int transactionStatus;//indica o estado da transação, podendo assumir os seguintes valores:
                        // 0 -> Nenhuma verifica dos dados do cliente (cartão e senha) foram realizados
                        // 1 -> Dados do cartão corretos;
                        // 2 -> Senha correta;

 public:
  ATM (Bank *);
  ~ATM();
  int AcceptCard (char[]);
  int PrintReceipts ();
  int ReadInPin(int pin);
  int WithdrawFunds (int ret_val, int valor);
  int CheckBalance (int ret_val);
  int TransferFunds (int ret_val, int valor);
  void Testa_Invariante ();
  void Relator (char* arquivo);
};

void Card::EatCard() {
}

ATM::ATM(Bank *bank) {
        bank_link = bank;
        transactionType = 0;
        transactionStatus=0;
}
```

```
ATM::~ATM() {
}

int ATM::AcceptCard(char Records[]) {
   char *CurrentCard;
   CurrentCard=Records;
   if (bank_link->getStatus()!=1) //Verifica se a leitura arquivo de conexão já foi realizada
       return(0);
   fp = fopen (CurrentCard, "r");
   if (fp==NULL){
          card.EatCard();
          return (0);
   }
   fscanf(fp, "%s %s %s", CurrentRecord.lastname, CurrentRecord.firstname, CurrentRecord.middle);
   if (!bank_link->CheckRecords(CurrentRecord)) {
          return (0);
   }
   transactionStatus=1;
   return (1);
}

int ATM::ReadInPin (int pin) {
    if ((bank_link->getStatus()!=1) || (transactionStatus!=1) ||(pin!=CurrentRecord.pin))
          return(0);
    transactionStatus=2;
    return (1);
}

int ATM :: WithdrawFunds(int ret_val, int valor) {
   int ret = 0;
   Pre_condicao ((valor <= 200) || (transactionStatus == 2));
   Pre_condicao ((ret_val >=1) && (ret_val<=2));
   if ((bank_link->getStatus()!=1) || (transactionStatus!=2))
          return(0);
   if ((ret_val==CHECKING) || (ret_val==SAVINGS)){
          if (ret_val == CHECKING) {
               if ( valor > CurrentRecord.CustomerAccounts[0].Balance)
                     return (0);
               else
                  CurrentRecord.CustomerAccounts[0].Balance=CurrentRecord.CustomerAccounts[0].Balance-
valor;
          }
          else {
               if (valor > CurrentRecord.CustomerAccounts[1].Balance)
                    return (0);
               else
                  CurrentRecord.CustomerAccounts[1].Balance=CurrentRecord.CustomerAccounts[0].Balance-
valor;
          }
          transactionType = 3;
          transactionStatus=0;
          bank_link->ModifyRecords(CurrentRecord, ret_val, valor, transactionType);
          return (1);
          }
   else
      return (0);
```

```
}
int ATM::CheckBalance(int ret_val) {
   Pre_condicao (transactionStatus==2);
   Pre_condicao ((ret_val >=1) && (ret_val<=2));
   if ((bank_link->getStatus()!=1) || (transactionStatus!=2))
         return(0);
   if ((ret_val==CHECKING) || (ret_val==SAVINGS)) {
         transactionType = 5;
         transactionStatus=0;
         return CurrentRecord.CustomerAccounts[ret_val-1].Balance;
          }
   else
         return -1;
}

int ATM::TransferFunds(int ret_val, int valor) {
      int ret = 0;
      Pre_condicao ((valor <= 200) || (transactionStatus == 2));
      Pre_condicao ((ret_val >=1) && (ret_val<=2));
      if ((bank_link->getStatus()!=1) || (transactionStatus!=2))
            return(0);
      if ((ret_val==CHECKING) || (ret_val==SAVINGS)){
            if (ret_val == CHECKING) {
                if ( valor > CurrentRecord.CustomerAccounts[0].Balance)
                        return (0);
                else{
                        CurrentRecord.CustomerAccounts[CHECKING-1].Balance -= valor;
                        CurrentRecord.CustomerAccounts[SAVINGS-1].Balance += valor;
                }
            }
            else {
                 if (valor > CurrentRecord.CustomerAccounts[1].Balance)
                        return (0);
                 else{
                        CurrentRecord.CustomerAccounts[SAVINGS-1].Balance -= valor;
                        CurrentRecord.CustomerAccounts[CHECKING-1].Balance += valor;
                 }
            }
            transactionType = 4;
            transactionStatus=0;
            bank_link->ModifyRecords(CurrentRecord, ret_val, valor, transactionType);
            return (1);
            }
      else
            return (0);
}

int ATM::PrintReceipts() {
      char   c = '\n';
      Pre_condicao (transactionType!=0);
      if (transactionType==0)
            return ERROR;
      fp=fopen ("Receipt.txt", "w");
      if (fp ==NULL) {
            return ERROR;;
       }
```

```
    fprintf(fp,"%s%c", CurrentRecord.lastname, c);
    fprintf(fp,"%s%c", CurrentRecord.firstname, c);
    fprintf(fp,"%s%c", CurrentRecord.middle, c);
    fprintf(fp,"%d%c", transactionType, c);
    fclose (fp);
    return (1);
}

void ATM::Testa_Invariante() {
    Invariante (bank_link->getStatus()==1);
}

void ATM::Relator(char* arquivo) {
   ofstream resultados(arquivo,ios::app);
   if (!resultados) {
       cout <<"Erro abriando arquivo!" << endl;
       exit (1);
   }
   else {
       resultados<<"Tipo da Transação" << transactionType << "\n";
       resultados<<"Status" << transactionStatus << "\n";
   }
   resultados.flush();
}

Bank::Bank () {
    FILE *fp;
    Database=_stub.getConnection();
    fp=fopen (Database, "a");
    Pre_condicao (fp!=NULL);
    if (fp ==NULL) {
          Status=0;
          exit (-1);
    }
    fclose (fp);
    Status=1;
 }


Bank::~Bank() {
   Status = 0;
}

int Bank::CheckRecords(struct CustomerRecord &record) {
    fp=fopen (Database, "r");
    while(fscanf(fp, "%s %s %s %d %d", BankRecords[1].lastname, BankRecords[1].firstname,
BankRecords[1].middle, &BankRecords[1].pin,&BankRecords[1].Accounts)!=EOF) {
    if(BankRecords[1].Accounts & CHECKING) {
          fscanf(fp, "%d %i",&BankRecords[1].CustomerAccounts[0].AccountNo,
&BankRecords[1].CustomerAccounts[0].Balance);
     }

    if(BankRecords[1].Accounts & SAVINGS) {
          fscanf(fp, "%d %i",&BankRecords[1].CustomerAccounts[1].AccountNo,
&BankRecords[1].CustomerAccounts[1].Balance);
     }
```

```
    if (!strcmp(BankRecords[1].lastname, record.lastname) && !strcmp(BankRecords[1].firstname,
record.firstname) && !strcmp(BankRecords[1].middle, record.middle)) {
            record.pin = BankRecords[1].pin;
            record.Accounts=BankRecords[1].Accounts;
            for (int y = 0; y < 2; y++) {

            record.CustomerAccounts[y].AccountNo=BankRecords[1].CustomerAccounts[y].AccountNo;
            record.CustomerAccounts[y].Balance=BankRecords[1].CustomerAccounts[y].Balance;
    }
            return (1);
            fclose (fp);
            break;
    }
}

return (0);
}

int Bank::ModifyRecords (struct CustomerRecord &record, int ret_val, int valor, int transaction) {
        RecordNumber = 0;
        int x=0;
       char   c = '\n';
        fp=fopen (Database, "r");
       if (fp ==NULL) {
           exit (-1);
        }
      while(fscanf(fp, "%s %s %s %d %d",
BankRecords[RecordNumber].lastname,BankRecords[RecordNumber]. firstname,
BankRecords[RecordNumber].middle,
&BankRecords[RecordNumber].pin,&BankRecords[RecordNumber].Accounts)!=EOF) {
            fscanf(fp, "%d %i",&BankRecords[RecordNumber].CustomerAccounts[0].AccountNo,
&BankRecords[RecordNumber].CustomerAccounts[0].Balance);
            fscanf(fp, "%d %i",&BankRecords[RecordNumber].CustomerAccounts[1].AccountNo,
&BankRecords[RecordNumber].CustomerAccounts[1].Balance);
            if (!strcmp(BankRecords[RecordNumber].lastname, record.lastname) &&
!strcmp(BankRecords[RecordNumber].firstname, record.firstname) &&
!strcmp(BankRecords[RecordNumber].middle, record.middle)) {
                    if (transaction == WITHDRAW) {
                            if (ret_val == CHECKING)

BankRecords[RecordNumber].CustomerAccounts[0].Balance=record.CustomerAccounts[0].Balance;
                            else

BankRecords[RecordNumber].CustomerAccounts[1].Balance=record.CustomerAccounts[1].Balance;
                            }
                    if (transaction == TRANSFER) {

BankRecords[RecordNumber].CustomerAccounts[0].Balance=record.CustomerAccounts[0].Balance;

BankRecords[RecordNumber].CustomerAccounts[1].Balance=record.CustomerAccounts[1].Balance;
                    }

            }
            RecordNumber++;
            }
            fclose (fp);
```

```
        fp=fopen (Database, "w");
        if (fp ==NULL) {
        exit (-1);
        }
        x = RecordNumber;
        RecordNumber=0;
        while (RecordNumber<x) {
                fprintf(fp,"%s%c", BankRecords[RecordNumber].lastname, c);
                fprintf(fp,"%s%c", BankRecords[RecordNumber].firstname, c);
                fprintf(fp,"%s%c", BankRecords[RecordNumber].middle, c);
                fprintf(fp,"%d\n", BankRecords[RecordNumber].pin);
                fprintf(fp,"%d\n", BankRecords[RecordNumber].Accounts);
                fprintf(fp,"%d\n", BankRecords[RecordNumber].CustomerAccounts[0].AccountNo);
                fprintf(fp,"%i\n", BankRecords[RecordNumber].CustomerAccounts[0].Balance);
                fprintf(fp,"%d\n", BankRecords[RecordNumber].CustomerAccounts[1].AccountNo);
                fprintf(fp,"%i\n", BankRecords[RecordNumber].CustomerAccounts[1].Balance);
                RecordNumber++;
        }
        fclose (fp);
        return (1);
}

int Bank::getStatus () {
        return Status;
}

void Bank::Testa_Invariante() {
}

void Bank::Relator(char* arquivo) {
      ofstream resultados(arquivo,ios::app);
      if (!resultados) {
      cout <<"Erro abriando arquivo!" << endl;
      exit (1);
   }
   else {
        resultados<<"Database" << Database << "\n";
        resultados<<"Status" << Status << "\n";
   }
   resultados.flush();
}
```

Apêndice E

Casos de Uso – Componente Acervo

Neste Apêndice são apresentados os casos de uso identificados para o Componente Acervo e também as tarefas (atividades) que devem ser executadas para realizá-lo, considerando-se os casos de funcionamento normal e também os de exceção. A seguir temos a descrição de todos os casos de uso identificados.

Caso de Uso Inserir Categoria	
Descrição:	Inserção de uma nova categoria
Pré-Condição:	Categoria inexistente
Invariante:	-
Cenário:	1. O Componente Cliente solicita a inserção de uma nova categoria 2. O Componente Acervo verifica se a categoria existe (categoria inexistente) e realiza o cadastramento da categoria
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 49 - Caso de Uso Inserir Categoria - Cenário Normal

Caso de Uso Inserir Categoria	
Cenário Excepcional: Violação de Pré-Condição	Inserção de uma categoria existente
Pós-Condição:	Geração de uma mensagem de erro

Figura 50 - Caso de Uso Inserir Categoria - Cenário Excepcional

Caso de Uso Inserir Filme	
Descrição:	Inserção de um novo filme
Pré-Condição:	Categoria existente; Filme inexistente
Invariante:	-
Cenário:	3. O Componente Cliente solicita a inserção de um novo filme 4. O Componente Acervo verifica se: ▪ A categoria existe (categoria existente); ▪ O filme existe (filme inexistente) e realiza o cadastramento do filme.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 51 - Caso de Uso Inserir Filme - Cenário Normal

Caso de Uso Inserir Filme	
Cenário Excepcional 1: Violação de Pré-Condição	Inserção de um filme existente
Pós-Condição:	Geração de uma mensagem de erro
Cenário Excepcional 2: Violação de Pré-Condição	Inserção de um filme (Categoria Inexistente)
Pós-Condição:	Geração de uma mensagem de erro

Figura 52 - Caso de Uso Inserir Filme - Cenários Excepcionais

Caso de Uso Inserir Fita	
Descrição:	Inserção de uma fita
Pré-Condição:	Filme existente
Invariante:	-
Cenário:	1. O Componente Cliente solicita a inserção de uma nova fita 2. O Componente Acervo verifica se o filme existe (filme existente) e realiza o cadastramento da fita.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 53 - Caso de Uso Inserir Fita - Cenário Normal

Caso de Uso Inserir Fita	
Cenário Excepcional: Violação de Pré-Condição	Inserção de uma fita (Filme Inexistente)
Pós-Condição:	Geração de uma mensagem de erro

Figura 54 - Caso de Uso Inserir Fita - Cenário Excepcional

Caso de Uso Remover Categoria	
Descrição:	Remover categoria
Pré-Condição:	Categoria existente; Categoria sem filmes associados
Invariante:	-
Cenário:	1. O Componente Cliente solicita a remoção de uma categoria 2. O Componente Acervo verifica se: ▪ A categoria existe (categoria existente) ▪ A categoria possui filmes associados (não possui) e realiza a retirada da categoria
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 55 - Caso de Uso Remover Categoria - Cenário Normal

Caso de Uso Remover Categoria	
Cenário Excepcional 1: Violação de Pré-Condição	Remover categoria inexistente
Pós-Condição:	Geração de uma mensagem de erro
Cenário Excepcional 2: Violação de Pré-Condição	Remover categoria (Filmes associados)
Pós-Condição:	Geração de uma mensagem de erro

Figura 56 - Caso de Uso Remover Categoria - Cenários Excepcionais

Caso de Uso Remover Filme	
Descrição:	Remover filme
Pré-Condição:	Filme existente; Filme sem fitas associadas
Invariante:	-
Cenário:	1. O Componente Cliente solicita a remoção de um filme 2. O Componente Acervo verifica se: ▪ O filme existe (categoria existente) ▪ O filme possui fitas associadas (não possui) e realiza a retirada do filme
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 57 - Caso de Uso Remover Filme - Cenário Normal

Caso de Uso Remover Filme	
Cenário Excepcional 1: Violação de Pré-Condição	Remover filme inexistente
Pós-Condição:	Geração de uma mensagem de erro
Cenário Excepcional 2: Violação de Pré-Condição	Remover Filme (Fitas associadas)
Pós-Condição:	Geração de uma mensagem de erro

Figura 58 - Caso de Uso Remover Filme - Cenários Excepcionais

Caso de Uso Remover Fita	
Descrição:	Remover fita
Pré-Condição:	Filme existente
Invariante:	-
Cenário:	1. O Componente Cliente solicita a remoção de uma fita 2. O Componente Acervo verifica se a fita existe (fita existente) e realiza a retirada da fita.
Pós-Condição:	Geração de uma mensagem indicando sucesso na operação.

Figura 59 - Caso de Uso Remover Fita - Cenário Normal

Caso de Uso Remover Fita	
Cenário Excepcional: **Violação de Pré-Condição**	Remover fita inexistente
Pós-Condição:	Geração de uma mensagem de erro

Figura 60 - Caso de Uso Remover Fita - Cenário Excepcional

Caso de Uso Listar Categoria	
Descrição:	Listar categoria
Pré-Condição:	-
Invariante:	-
Cenário:	1. O Componente Cliente solicita a listagem das categorias 2. O Componente Acervo gera uma listagem das categorias
Pós-Condição:	Geração da listagem de categorias.

Figura 61 - Caso de Uso Listar Categoria - Cenário Normal

Caso de Uso Listar Filme	
Descrição:	Listar categoria
Pré-Condição:	-
Invariante:	-
Cenário:	1. O Componente Cliente solicita a listagem dos filmes 2. O Componente Acervo gera uma listagem dos filmes
Pós-Condição:	Geração da listagem de filmes.

Figura 62 - Caso de Uso Listar Filme - Cenário Normal

Caso de Uso Listar Fita	
Descrição:	Listar categoria
Pré-Condição:	
Invariante:	
Cenário:	1. O Componente Cliente solicita a listagem das fitas 2. O Componente Acervo gera uma listagem das fitas
Pós-Condição:	Geração da listagem de fitas.

Figura 63 - Caso de Uso Listar Fita - Cenário Normal

Apêndice F

Especificação de Teste – Componente Acervo

A figura a seguir a apresenta a especificação de teste gerada para o componente Acervo.

```
classe('Categoria',n,_,[]).

atributo ('fp', ponteiro, 'File',_).

metodo(m1,'categoria',_,novo,0).
metodo(m2,'insereCategoria','void',novo,1).
metodo(m3,'removeCategoria','void',novo,1).
metodo(m4,'listaCategoria','void',novo,0).
metodo(m5,'FilmeinsereFilme','void',novo,3).
metodo(m6,'FilmeremoveFilme','void',novo,1).
metodo(m7,'FilmelistaFilme','void',novo,0).
metodo(m8,'FitainsereFita','void',novo,1).
metodo(m9,'FitaremoveFita','void',novo,1).
metodo(m10,'FitalistaFita','void',novo,0).

parametro('nomeCategoria','m2',string,['acao','romance','suspense'],_).
parametro('idCategoria','m3',intervalo,1,100).
parametro('nomeFilme','m5',string,['Seven', 'Titanic', 'Coracao Valente'],_).
parametro('anoProducao','m5',intervalo,1970,2002).
parametro('idCategoria','m5',intervalo,1,100).
parametro('idFilme','m6',intervalo,1,100).
parametro('nomeFilme','m8',string,['Seven', 'Titanic', 'Coracao Valente'],_).
parametro('idFita','m9',intervalo,1,100).

no(n1,sim,9,[m1]).
no(n2,nao,0,[m2]).
no(n3,nao,0,[m3]).
no(n4,nao,0,[m4]).
no(n5,nao,0,[m5]).
no(n6,nao,0,[m6]).
no(n7,nao,0,[m7]).
no(n8,nao,0,[m8]).
no(n9,nao,0,[m9]).
no(n10,nao,0,[m10]).

arco(n1,n2).
arco(n1,n3).
```

```
arco(n1,n4).
arco(n1,n5).
arco(n1,n6).
arco(n1,n7).
arco(n1,n8).
arco(n1,n9).
arco(n1,n10).
```

Figura 64 – Especificação de Teste – Componente Acervo

Apêndice G

Driver Específico e Caso de Teste – Componente Acervo

Neste Apêndice são apresentados alguns dados gerados pela ConCAT após a leitura da especificação de teste. A figura 38 apresenta o driver específico gerado.

```
#include <fstream.h>
#include <iostream.h>
#include "categoria.cpp"
#include "categoria_ct0.cc"

#define TESTE

void main() {
  categoria* obj1 = new categoria;
  Caso_teste1_0(obj1);
  categoria* obj2 = new categoria;
  Caso_teste2_0(obj2);
  categoria* obj3 = new categoria;
  Caso_teste3_0(obj3);
  categoria* obj4 = new categoria;
  Caso_teste1_1(obj4);
  categoria* obj5 = new categoria;
  Caso_teste2_1(obj5);
  categoria* obj6 = new categoria;
  Caso_teste3_1(obj6);
  categoria* obj7 = new categoria;
  Caso_teste7_1(obj7);
  categoria* obj8 = new categoria;
  Caso_teste8_1(obj8);
  Categoria* obj9 = new categoria;
  Caso_teste9_1(obj9);
}
```

Figura 65 – Driver Gerado para o Componente Acervo

Neste estudo de caso também foi necessário alterar o driver gerado pois no MFT construído para este componente (Figura 29 – Capítulo 6) existiam mais do que um nó

inicial contendo métodos construtores de classes diferentes, sendo que a ferramenta ConCAT gera o driver baseado na existência de métodos construtores de apenas uma classe. Desta forma foi necessário modificar o driver gerado para que os teste pudessem ser realizados.

```
#include <fstream.h>
#include <iostream.h>
#include "acervo.h"
#include "casos.h"

#define TESTE

void main() {
  Categoria* obj1 = new Categoria;
  Caso_teste1_0(obj1);
  Categoria* obj2 = new Categoria;
  Caso_teste2_0(obj2);
  Categoria* obj3 = new Categoria;
  Caso_teste3_0(obj3);
  Filme* obj4 = new Filme;
  Caso_teste4_0(obj4);
  Filme* obj5 = new Filme;
  Caso_teste5_0(obj5);
  Filme* obj6 = new Filme;
  Caso_teste6_0(obj6);
  Fita* obj7 = new Fita;
  Caso_teste7_0(obj7);
  Fita* obj8 = new Fita;
  Caso_teste8_0(obj8);
  Fita* obj9 = new Fita;
  Caso_teste9_0(obj9);
}
```

Figura 66 – Driver Modificado – Componente Acervo

Em relação as modificações realizados nos casos de teste, as figuras 40 e 41 apresentam um caso de teste antes e após a modificação da assinatura dos métodos.

```
template <class Tipo>
void Caso_teste4_0(Tipo* cst) {
  char* met= new char[30];
  ofstream arq("Result.txt",ios::app);
  if (! arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Arquivo criado! \n";
  try {
    cst->Testa_Invariante();
    met = "FilmeinsereFilme(Seven, 1970,1)";
    cst-> FilmeinsereFilme(Seven, 1970,1);
```

```
    cst->Testa_Invariante();
    arq << "Caso_teste4_0 OK!\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
    delete cst;
  }

  catch(char* c) {
    arq << "Caso_teste4_0 \n";
    arq.flush();
    cout << "...";
    arq << c << "\n";
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
  }

  catch(...) {
    arq.close();
    }
}
```

Figura 67 - Caso de Teste gerado – Componente Acervo

```
template <class Tipo>
void Caso_teste4_0(Tipo* cst) {
  char* met= new char[30];
  ofstream arq("Result.txt",ios::app);
  if (! arq)
    cout << "Erro abrindo arquivo! \n";
  else
    cout << "Arquivo criado! \n";
  try {
    cst->Testa_Invariante();
    met = "FilmeinsereFilme(Seven, 1970,1)";
    cst-> insereFilme(Seven, 1970,1);
    cst->Testa_Invariante();
    arq << "Caso_teste4_0 OK!\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
    delete cst;
  }

  catch(char* c) {
    arq << "Caso_teste4_0 \n";
    arq.flush();
    cout << "...";
    arq << c << "\n";
```

```
    arq << "Metodo chamado: " << met << "\n";
    arq.flush();
    cst->Relator("Result.txt");
    arq << "\n";
    arq.close();
  }

  catch(...) {
    arq.close();
    }
}
```

Figura 68 - Caso de Teste Modificado – Componente Acervo

Apêndice H

Código - Componente Acervo

Neste Apêndice é apresentado o código fonte do Componente Acervo após a inserção dos métodos de teste embutido.

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <fstream.h>
#include "autoteste.h"
#define TESTE

struct noCategoria {
            char nomeCategoria[30];
    int idCategoria;
};

struct noFilme {
    char nomeFilme[30];
    int idFilme;
    int anoProducao;
    int idCategoria;
};

struct noFita {
            int idFita;
    int idFilme;
};

class categoria: Autoteste{
    public:
     categoria();
     int insereCategoria (char nomeCategoria[30]);
     int removeCategoria (int idCategoria);
     void listaCategoria ();
     void Testa_Invariante ();
     void Relator (char* arquivo);
     int buscaCategoria (int idCategoria);
    private:
     FILE *fp;
     int buscaCategoriaNome (char nomeCategoria[30]);
```

```
    int getTamanhoLista ();
    int qtaCategoria;
    noCategoria dadosCategoria[30];
};

class filme: Autoteste {
   public:
    filme ();
    int insereFilme (char nomeFilme[30], int anoProducao, int _idCategoria);
    int removeFilme (int idFilme);
    void listaFilme ();
    int buscaFilme (int idCategory);
    int buscaFilmeNome (char nomeFilme[30]);
    void Testa_Invariante ();
    void Relator (char* arquivo);
   private:
    FILE *fp;
    int buscaIdFilme (int idFilme);
    int getTamanhoLista ();
   int qtaFilme;
   noFilme dadosFilme[30];
};

class fita: Autoteste {
   public:
    fita();
    int insereFita (char nomeFilme[30]);
    int removeFita (int idFita);
    void listaFita ();
    int buscaFita (int idFilme);
    void Testa_Invariante ();
    void Relator (char* arquivo);
   private:
    FILE *fp;
    int qtaFita;
    noFita dadosFita[30];
};

/*-----------------------------METODOS-----------------------------------*/
categoria::categoria(){
        char  c = '\n';
        fp=fopen ("categoria.txt", "r");
        if (fp==NULL) {
                fp=fopen ("categoria.txt","w");
                qtaCategoria = 0;
                fprintf(fp,"%i%c", qtaCategoria, c);
        }
        else
                fscanf(fp, "%i",&qtaCategoria);
        fclose (fp);
}
/*-----------------------------------------------------------------------*/
int categoria::insereCategoria(char _nomeCategoria[30]) {
        char  c = '\n';
        int cont,aux=0;
  if (qtaCategoria==0) {
```

```
                fp=fopen ("categoria.txt","w");
                qtaCategoria++;
                fprintf(fp,"%i%c", qtaCategoria, c);
                fprintf(fp,"%i%c", qtaCategoria, c);
                fprintf(fp,"%s%c", _nomeCategoria, c);
                fclose (fp);
                return (1);
        }
        else {
                if (buscaCategoriaNome(_nomeCategoria)!=1) {
                        fp=fopen ("categoria.txt", "r+");
                        fscanf(fp, "%i",&qtaCategoria);
                        while(fscanf(fp, "%i %s",
&dadosCategoria[aux].idCategoria,dadosCategoria[aux].nomeCategoria)!=EOF) {
                                aux++;
                        }
                        qtaCategoria=aux+1;
                        fclose(fp);

                        fp=fopen ("categoria.txt","w");
                        fprintf(fp,"%i%c", qtaCategoria, c);
                        cont = aux;
                        aux =0;
                        while(aux<cont){
                                fprintf(fp,"%i%c", dadosCategoria[aux].idCategoria, c);
                                fprintf(fp,"%s%c", dadosCategoria[aux].nomeCategoria, c);
                                aux++;
                        }
                        fprintf(fp,"%i%c", dadosCategoria[aux-1].idCategoria+1, c);
                        fprintf(fp,"%s%c", _nomeCategoria, c);
                        fclose (fp);
                        return (1);

                }
                else
                        return (0);
        }
}
/*-----------------------------------------------------------------*/
int categoria::removeCategoria(int _idCategoria){
        int cont;
        filme *_filme = new filme;
        Pre_condicao (_filme->buscaFilme(_idCategoria)!=0);
        if ((_filme->buscaFilme(_idCategoria)!=0) || (buscaCategoria(_idCategoria)==0))
                        return (0);
        else {
                char c='\n';
                int aux=0;
                fp=fopen ("categoria.txt","r");
                fscanf(fp, "%i",&qtaCategoria);
                while(fscanf(fp, "%i %s",
&dadosCategoria[aux].idCategoria,dadosCategoria[aux].nomeCategoria)!=EOF){
                        if (dadosCategoria[aux].idCategoria==_idCategoria) {
                                dadosCategoria[aux].idCategoria=-1;

                                }
```

```
                        aux++;
                }
                fclose (fp);
                aux= qtaCategoria-1;
                fp=fopen ("categoria.txt","w");
                rewind(fp);
                fprintf(fp,"%i%c", aux, c);
                cont = aux;
                aux =0;
                while(aux<cont){
                        if (dadosCategoria[aux].idCategoria!=-1) {
                                fprintf(fp,"%i%c", dadosCategoria[aux].idCategoria, c);
                                fprintf(fp,"%s%c", dadosCategoria[aux].nomeCategoria, c);
                        }
                        aux++;
                }

        fclose (fp);
        return (1);
        }
}
/*-----------------------------------------------------------------*/
void categoria::listaCategoria(){
        char c = '\n';
        int aux=0;
        int qta;
        int cont;
        char mensagem [30] = "Numero de Categorias: ";
        fp=fopen ("categoria.txt","r");
        fscanf(fp, "%i",&qta);
        if (qta==0) {
                fclose (fp);
                fp=fopen ("ListagemCategorias.txt","w");
                fprintf(fp,"%s", mensagem);
                fprintf(fp,"%i%c", qta, c);
                fclose (fp);
        }
        else {
                while(fscanf(fp, "%i %s",
&dadosCategoria[aux].idCategoria,dadosCategoria[aux].nomeCategoria)!=EOF){
                        aux++;
                }
                fclose (fp);

                fp=fopen ("ListagemCategorias.txt","w");
                fprintf(fp,"%s", mensagem);
                fprintf(fp,"%i%c", qta, c);
                cont = aux;
                aux =0;
                while(aux<cont){
                        if (dadosCategoria[aux].idCategoria!=-1) {
                                fprintf(fp,"%i%c", dadosCategoria[aux].idCategoria, c);
                                fprintf(fp,"%s%c", dadosCategoria[aux].nomeCategoria, c);
                        }
                        aux++;
                }
```

```
                    fclose (fp);
            }
}
/*-------------------------------------------------------------------*/
void categoria::Testa_Invariante() {

}
/*-------------------------------------------------------------------*/
void categoria::Relator(char* arquivo) {
    int qta=0;
    int aux =0;
    ofstream resultados(arquivo,ios::app);
    if (!resultados) {
      cout <<"Erro abriando arquivo!" << endl;
      exit (1);
    }
    else {
      fp=fopen ("categoria.txt","r");
      fscanf(fp, "%i",&qta);
      if (qta==0)
          resultados<<"Qta_Categoria " << qta << "\n";
     else {
          resultados<<"Qta_Categoria " << qta << "\n";
          while(fscanf(fp, "%i %s",
&dadosCategoria[aux].idCategoria,dadosCategoria[aux].nomeCategoria)!=EOF){
                resultados<<"IdCategoria " << dadosCategoria[aux].idCategoria << "\n";
                resultados<<"Nome " << dadosCategoria[aux].nomeCategoria << "\n";
                aux++;
              }
       fclose (fp);
       }
    }
    resultados.flush();
}

/*-------------------------------------------------------------------*/
int categoria::buscaCategoria (int _idCategoria){
          char   c = '\n';
          int aux=0;
          fp=fopen ("categoria.txt", "r");
          fscanf(fp, "%i",&qtaCategoria);
          while(fscanf(fp, "%i %s",
&dadosCategoria[aux].idCategoria,dadosCategoria[aux].nomeCategoria)!=EOF){
                    if (dadosCategoria[aux].idCategoria==_idCategoria) {
                              fclose (fp);
                              return (1);
                    }
          }
          fclose (fp);
          return (0);
}
/*-------------------------------------------------------------------*/
int categoria::buscaCategoriaNome (char _nomeCategoria [30]){
          char   c = '\n';
          int aux=0;
          fp=fopen ("categoria.txt", "r");
```

```
        fscanf(fp, "%i",&qtaCategoria);
        while(fscanf(fp, "%i %s",
&dadosCategoria[aux].idCategoria,dadosCategoria[aux].nomeCategoria)!=EOF){
                if (!strcmp(dadosCategoria[aux].nomeCategoria,_nomeCategoria)) {
                        fclose (fp);
                        return (1);
                }
        }
        fclose (fp);
        return (0);
}

/*-------------------------------------------------------------------*/
int categoria::getTamanhoLista () {
        return qtaCategoria;
}

/*-------------------------------------------------------------------*/
filme::filme(){
        char   c = '\n';
        FILE *fp;
        fp=fopen ("filme.txt", "r");
        if (fp==NULL) {
                fp=fopen ("filme.txt","w");
                qtaFilme = 0;
                fprintf(fp,"%i%c", qtaFilme, c);
        }
        else
                fscanf(fp, "%i",&qtaFilme);
        fclose (fp);
}

/*-------------------------------------------------------------------*/
int filme::insereFilme(char _nomeFilme[30], int _anoProducao, int _idCategoria){
   categoria *_categoria = new categoria;
        Pre_condicao (_categoria->buscaCategoria(_idCategoria)!=0);
        if (_categoria->buscaCategoria(_idCategoria)!=1)
                        return (0);
        else{
                char   c = '\n';
                int cont=0;
                int aux=0;
                if (qtaFilme==0) {
                        fp=fopen ("filme.txt","w");
                        qtaFilme++;
                        fprintf(fp,"%i%c", qtaFilme, c);
                        fprintf(fp,"%i%c", qtaFilme, c);
                        fprintf(fp,"%s%c", _nomeFilme, c);
                        fprintf(fp,"%i%c", _anoProducao, c);
                        fprintf(fp,"%i%c", _idCategoria, c);
                        fclose (fp);
                        return (1);
                }
                else {
                        if (buscaFilmeNome (_nomeFilme)!=1) {
                                fp=fopen ("filme.txt", "r");
```

```
                                        fscanf(fp, "%i",&qtaFilme);
                                        while(fscanf(fp, "%i %s %i %i",
&dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme, &dadosFilme[aux].anoProducao,
&dadosFilme[aux].idCategoria)!=EOF){
                                                aux++;
                                        }
                                        qtaFilme= aux+1;
                                    fclose(fp);
                                        fp=fopen ("filme.txt","w");
                                        fprintf(fp,"%i%c", qtaFilme, c);
                                        cont = aux;
                                        aux =0;
                                        while(aux<cont){
                                                fprintf(fp,"%i%c", dadosFilme[aux].idFilme, c);
                                                fprintf(fp,"%s%c", dadosFilme[aux].nomeFilme, c);
                                                fprintf(fp,"%i%c", dadosFilme[aux].anoProducao, c);
                                                fprintf(fp,"%i%c", dadosFilme[aux].idCategoria, c);
                                                aux++;
                                        }
                                        fprintf(fp,"%i%c", dadosFilme[aux-1].idFilme+1, c);
                                        fprintf(fp,"%s%c", _nomeFilme, c);
                                        fprintf(fp,"%i%c", _anoProducao, c);
                                        fprintf(fp,"%i%c", _idCategoria, c);
                                        fclose (fp);
                                        return (1);
                                }
                                else {
                                        fclose (fp);
                                        return (0);
                                }
                        }
                }

}

/*-------------------------------------------------------------------------*/
int filme::removeFilme(int _idFilme){
        int cont;
        fita *_fita = new fita;
        Pre_condicao (_fita->buscaFita(_idFilme)!=0);
        if ((_fita->buscaFita(_idFilme)!=0) || (buscaIdFilme(_idFilme)==0))
                        return (0);
        else {
                char  c = '\n';
                int aux=0;
                fp=fopen ("filme.txt","r");
                fscanf(fp, "%i",&qtaFilme);
                while(fscanf(fp, "%i %s %i %i", &dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme,
&dadosFilme[aux].anoProducao, &dadosFilme[aux].idCategoria)!=EOF){
                        if (dadosFilme[aux].idFilme==_idFilme) {
                                dadosFilme[aux].idFilme=-1;


                                }
                        aux++;
                }
```

```
                fclose (fp);
                aux= qtaFilme-1;
                fp=fopen ("filme.txt","w");
                fprintf(fp,"%i%c", aux, c);
                cont = aux;
                aux =0;
                while(aux<cont){
                        fprintf(fp,"%i%c", dadosFilme[aux].idFilme, c);
                        fprintf(fp,"%s%c", dadosFilme[aux].nomeFilme, c);
                        fprintf(fp,"%i%c", dadosFilme[aux].anoProducao, c);
                        fprintf(fp,"%i%c", dadosFilme[aux].idCategoria, c);
                        aux++;
                }
        fclose (fp);
        return (1);
        }

}


/*-------------------------------------------------------------------*/
void filme::listaFilme(){

        char c = '\n';
        int aux=0;
        int qta;
        int cont;
        char mensagem [30] = "Numero de Filmes: ";
        fp=fopen ("filme.txt","r");
        fscanf(fp, "%i",&qta);
        if (qta==0) {
                fclose (fp);
                fp=fopen ("ListagemFilmes.txt","w");
                fprintf(fp,"%s", mensagem);
                fprintf(fp,"%i%c", qta, c);
                fclose (fp);
        }
        else {
                while(fscanf(fp, "%i %s %i %i", &dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme,
&dadosFilme[aux].anoProducao, &dadosFilme[aux].idCategoria)!=EOF){
                        aux++;
                }

                fclose (fp);

                fp=fopen ("ListagemFilmes.txt","w");
                fprintf(fp,"%s", mensagem);
                fprintf(fp,"%i%c", qta, c);
                cont = aux;
                aux =0;
                while(aux<cont){
                        fprintf(fp,"%i%c", dadosFilme[aux].idFilme, c);
                        fprintf(fp,"%s%c", dadosFilme[aux].nomeFilme, c);
                        fprintf(fp,"%i%c", dadosFilme[aux].anoProducao, c);
                        fprintf(fp,"%i%c", dadosFilme[aux].idCategoria, c);
                        aux++;
```

```
                }
        }
}

/*-------------------------------------------------------------------------*/
void filme::Testa_Invariante() {
}
/*-------------------------------------------------------------------------*/
void filme::Relator(char* arquivo) {
    int qta=0;
    int aux =0;
    ofstream resultados(arquivo,ios::app);
    if (!resultados) {
      cout <<"Erro abriando arquivo!" << endl;
      exit (1);
    }
    else {
       fp=fopen ("filme.txt","r");
       fscanf(fp, "%i",&qta);
       if (qta==0)
           resultados<<"Qta_Filme " << qta << "\n";
       else {
           resultados<<"Qta_Filme " << qta << "\n";
           while(fscanf(fp, "%i %s %i %i", &dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme,
&dadosFilme[aux].anoProducao, &dadosFilme[aux].idCategoria)!=EOF){
                   resultados<<"IdFilme " << dadosFilme[aux].idFilme << "\n";
                   resultados<<"Nome " <<dadosFilme[aux].nomeFilme << "\n";
                   resultados<<"Ano_Producao " << dadosFilme[aux].anoProducao << "\n";
                   resultados<<"IdCategoria " << dadosFilme[aux].idCategoria << "\n";
                  aux++;
           }
          fclose (fp);
       }
    }
   resultados.flush();
}
/*-------------------------------------------------------------------------*/
int filme::getTamanhoLista () {
        return qtaFilme;
}

/*-------------------------------------------------------------------------*/
int filme::buscaFilmeNome (char nomeFilme[30]){
        char  c = '\n';
        int aux=0;
        fp=fopen ("filme.txt", "r");
        fscanf(fp, "%i",&qtaFilme);
        while(fscanf(fp, "%i %s %i %i", &dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme,
&dadosFilme[aux].anoProducao, &dadosFilme[aux].idCategoria)!=EOF){
                if (!strcmp(dadosFilme[aux].nomeFilme,nomeFilme)) {
                        fclose (fp);
                        return dadosFilme[aux].idFilme;
                }
        }
        fclose (fp);
        return (0);
```

```
}

/*-------------------------------------------------------------------*/
int filme::buscaFilme (int idCategoria){
        char   c = '\n';
        int aux=0;
        fp=fopen ("filme.txt", "r");
        fscanf(fp, "%i",&qtaFilme);
        while(fscanf(fp, "%i %s %i %i", &dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme,
&dadosFilme[aux].anoProducao, &dadosFilme[aux].idCategoria)!=EOF){
                if (dadosFilme[aux].idCategoria==idCategoria) {
                        fclose (fp);
                        return (1);
                }
        }
        fclose (fp);
        return (0);

}

/*-------------------------------------------------------------------*/
int filme::buscaIdFilme (int idFilme){
        char   c = '\n';
        int aux=0;
        fp=fopen ("filme.txt", "r");
        fscanf(fp, "%i",&qtaFilme);
        while(fscanf(fp, "%i %s %i %i", &dadosFilme[aux].idFilme,dadosFilme[aux].nomeFilme,
&dadosFilme[aux].anoProducao, &dadosFilme[aux].idCategoria)!=EOF){
                if (dadosFilme[aux].idFilme==idFilme) {
                        fclose (fp);
                        return (1);
                }
        }
        fclose (fp);
        return (0);

}
/*-------------------------------------------------------------------*/
fita::fita () {
        char   c = '\n';
        FILE *fp;
        fp=fopen ("fita.txt", "r");
        if (fp==NULL) {
                fp=fopen ("fita.txt","w");
                qtaFita = 0;
                fprintf(fp,"%i%c", qtaFita, c);
        }
        else
                fscanf(fp, "%i",&qtaFita);
        fclose (fp);
}
/*-------------------------------------------------------------------*/
int fita::insereFita(char nomeFilme[30]){
        filme *_filme = new filme;
        Pre_condicao (_filme->buscaFilmeNome(nomeFilme)!=0);
        if (_filme->buscaFilmeNome(nomeFilme)==0)
```

```
                return (0);
        else {
                char   c = '\n';
                int aux=0;
          if (qtaFita==0) {
                fp=fopen ("fita.txt","w");
                qtaFita++;
                fprintf(fp,"%i%c", qtaFita, c);
                fprintf(fp,"%i%c", qtaFita, c);
                fprintf(fp,"%i%c",_filme->buscaFilmeNome(nomeFilme), c);
                fclose (fp);
                return (1);
                }
                else {
                fp=fopen ("fita.txt", "a");
                while(fscanf(fp, "%i %i ", &dadosFita[1].idFita,dadosFita[1].idFilme)!=EOF) {
                        aux++;
                }
                qtaFita=aux+1;
                rewind(fp);
                fprintf(fp,"%i%c", qtaFita, c);
              while(fscanf(fp, "%i %i ", &dadosFita[1].idFita,dadosFita[1].idFilme)!=EOF);
                fprintf(fp,"%i%c", dadosFita[1].idFita+1, c);
                fprintf(fp,"%i%c",_filme->buscaFilmeNome(nomeFilme), c);
                fclose (fp);
                return (1);
                }
        }
}
/*-------------------------------------------------------------------*/
int fita::removeFita(int _idFita){
        char   c = '\n';
        int aux=0;
        int cont;
        fp=fopen ("fita.txt","r");
        fscanf(fp, "%i",&qtaFita);
        if ((qtaFita==0) || (buscaFita(_idFita)==0)){
                return (0);
        }
        else {
                while(fscanf(fp, "%i %i", &dadosFita[aux].idFita,&dadosFita[aux].idFilme)!=EOF){
                        if (dadosFita[aux].idFita==_idFita) {
                                dadosFita[aux].idFita=-1;
                        }
                        aux++;
                }
                fclose (fp);
                aux= qtaFita-1;
                fp=fopen ("fita.txt","w");
                fprintf(fp,"%i%c", aux, c);
                cont = aux;
                aux =0;
                while(aux<cont){
                        if (dadosFita[aux].idFita!=-1) {
                                fprintf(fp,"%i%c", dadosFita[aux].idFita, c);
                                fprintf(fp,"%i%c", dadosFita[aux].idFita, c);
```

```
                        }
                        aux++;
                }
                fclose (fp);
                return (1);
        }
}
/*-------------------------------------------------------------------*/
void fita::listaFita(){
        char c = '\n';
        int aux=0;
        int qta;
        int cont;
        char mensagem [30] = "Numero de Fitas: ";
        fp=fopen ("fita.txt","r");
        fscanf(fp, "%i",&qta);
        if (qta==0) {
                fclose (fp);
                fp=fopen ("ListagemFitas.txt","w");
                fprintf(fp,"%s", mensagem);
                fprintf(fp,"%i%c", qta, c);
                fclose (fp);
        }
        else {
                while(fscanf(fp, "%i %i", &dadosFita[aux].idFita,&dadosFita[aux].idFilme)!=EOF){
                        aux++;
                }

                fclose (fp);

                fp=fopen ("ListagemFitas.txt","w");
                fprintf(fp,"%s", mensagem);
                fprintf(fp,"%i%c", qta, c);
                cont = aux;
                aux =0;
                while(aux<cont){
                        fprintf(fp,"%i%c", dadosFita[aux].idFita, c);
                        fprintf(fp,"%i%c", dadosFita[aux].idFita, c);
                        aux++;
                }

        }
}

/*-------------------------------------------------------------------*/
void fita::Testa_Invariante() {
}
/*-------------------------------------------------------------------*/
void fita::Relator(char* arquivo) {
    int qta=0;
    int aux =0;
    ofstream resultados(arquivo,ios::app);
    if (!resultados) {
        cout <<"Erro abriando arquivo!" << endl;
        exit (1);
    }
```

```
    else {
        fp=fopen ("fita.txt","r");
        fscanf(fp, "%i",&qta);
        if (qta==0)
        resultados<<"Qta_Fita " << qta << "\n";
        else {
            resultados<<"Qta_Fita " << qta << "\n";
            while(fscanf(fp, "%i %i", &dadosFita[aux].idFita,&dadosFita[aux].idFilme)!=EOF){
                    resultados<<"IdFita " << dadosFita[aux].idFita << "\n";
                    resultados<<"IdFilme " << dadosFita[aux].idFilme << "\n";
                    aux+
+;
            }
            fclose (fp);
        }
     }
   resultados.flush();
}

/*-------------------------------------------------------------------*/
int fita::buscaFita (int idFilme){
        char   c = '\n';
        int aux=0;
        fp=fopen ("fita.txt", "r");
        fscanf(fp, "%i",&qtaFita);
        while(fscanf(fp, "%i %i", &dadosFita[aux].idFita,&dadosFita[aux].idFilme)!=EOF){
                if (dadosFita[aux].idFilme==idFilme) {
                        fclose (fp);
                        return (1);
                }
        }
        fclose (fp);
        return (0);
}
```