Melhoria da Testabilidade de Classes Usando o Conceito de Autoteste

Cristina Maria Toyota

Dissertação de Mestrado

Instituto de Computação

Universidade Estadual de Campinas

Melhoria da Testabilidade de Classes Usando o Conceito de Autoteste

Cristina Maria Toyota

Junho de 2000

Banca Examinadora:

- Profa. Dra. Eliane Martins
  IC - UNICAMP (Orientadora)
- Profa. Dra. Ana Maria de Alencar Price
  Instituto de Informática - UFRGS
- Profa. Dra. Cecília M. F. Rubira
  IC - UNICAMP
- Prof. Dr. Luiz Eduardo Buzato (Suplente)
  IC - UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Toyota, Cristina Maria

T668m Melhoria da testabilidade de classes usando o conceito de autoteste / Cristina Maria Toyota -- Campinas, [S.P. :s.n.], 2000.

Orientador : Eliane Martins

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

1. Testes. 2. Engenharia de Software. 3. Prolog (Linguagem de programação de computador). 4. Software - Testes. I. Martins, Eliane. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Melhoria da Testabilidade de Classes Usando o Conceito de Autoteste

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Cristina Maria Toyota e aprovada pela Banca Examinadora.

Campinas, 17 de julho de 2000.

Eliane Martins
(Orientadora)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 17 de julho de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Ana Maria de Alencar Price
UFRGS

Profa. Dra. Cecília Mary Fischer Rubira
IC-UNICAMP

Profa. Dra. Eliane Martins
IC-UNICAMP

Aos que passam pela nossa vida

Cada um que passa em nossa vida
passa sozinho...

Porque cada pessoa é única para nós,
e nenhuma substitui a outra...

Cada um que passa em nossa vida
passa sozinho,
mas não vai só...

Cada um que passa em nossa vida
leva um pouco de nós mesmos
e nos deixa um pouco de si mesmo...

Há os que levam muito,
mas não há os que não levam nada...

Há os que deixam muito,
mas não há os que não deixam nada...

Esta é a mais bela realidade da vida,
a prova tremenda de que cada um é importante
e que ninguém se aproxima do outro por acaso...

Saint-Exuperry

Agradecimentos

Primeiramente agradeço a Deus por estar comigo sempre, iluminando meu caminho e me incentivando através de meus amigos e parentes.

Agradeço a meus pais e irmãos pelo seu amor e grande incentivo para o término deste trabalho.

À minha orientadora, Profa. Dra. Eliane Martins, pela amizade, carinho, e principalmente pelo apoio e incentivo à conclusão deste trabalho. Obrigada por ter acreditado em mim Eliane!

Meu agradecimento para os amigos que conheci durante o mestrado, pessoas maravilhosas que tornaram este período muito importante na minha vida. Em especial para Alessandra e Edi, Cris Campos, Freud, Jane, Glaucia, Ghandi e Norma, Gisele, Luciano, Marília, Mário e Laura, Marcelo Marcos, Roseli, Vera, André, Delano, Adriana, Pascual, Guilhermes, Vaguinho, Valter, Alessandro, Ghandi de Maringá.

Não poderia deixar de agradecer também aos meu amigos da ex-república dos "BBB": Márcio, Rogério e Ralph. Obrigada por sua amizade, vocês moram no meu coração!

Finalmente, as Marquinhas (minhas irmãzinhas de coração): Amandita, Karina, Paty e Selma. Meninas, como agradecer tanta carinho, amizade e paciência durante todo este tempo? Difícil... Mas vou deixar registrado aqui meu muito obrigada por estarem ao meu lado sempre, nos bons e maus momentos!

Ao CNPq pelo apoio financeiro durante a realização do trabalho.

Resumo

O esforço e custo adicionais exigidos pelo processo de teste de um software ou componente depende em geral de quão fácil é testá-lo. No caso de componentes reutilizáveis essa habilidade se torna mais importante ainda, pois eles devem ser testados durante seu desenvolvimento, toda vez que forem reutilizados em um novo contexto, e a cada vez que sofram alguma alteração.

Uma proposta para construir classes com maior testabilidade foi feita fazendo-se uma analogia entre circuitos integrados e classes. Foi proposta a construção de classes autotestáveis e tal como em circuitos integrados, estas classes teriam mecanismos embutidos específicos para teste.

Outro fator importante para testabilidade do software é a possibilidade de reutilização de testes. Por esta razão foi empregada nesse estudo a técnica incremental hierárquica, que leva em conta a hierarquia de herança permitindo que, em alguns casos, seja possível reutilizar casos de testes gerados para a classe base nos testes de suas classes derivadas.

Este trabalho apresenta a metodologia definida para construir uma classe autotestável, bem como o protótipo construído para automatizar parte desta metodologia, mostrando assim a sua viabilidade.

Abstract

In general the additional effort and cost required by the testing process depends on the testability of the software or component being tested. Testability is more important for reusable components because they need testing during their development and after each change (either in the class or in the environment).

An approach comparing classes and integrated circuits of hardware had been proposed for constructing classes with higher testability. The resulting class is called built-in self-testing class, because it contains built-in testing mechanisms.

Class testability will be increased if reuse could be extended to the test cases too. The incremental hierarchical technique considers the inheritance relationship to test classes by reusing test cases of a parent class when testing a subclass.

This work presentes: a methodology to construct built-in self-testing classes and a prototyping tool, ConCAT, that automates part of this methodology, by showing its feasibility.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A orientação a objetos é uma abordagem para o desenvolvimento de software que vem sendo cada vez mais utilizada pois o conceito de objetos permite a criação de sistemas mais flexíveis e modulares, ou seja, mais fáceis de manter. A utilização deste paradigma favorece a reutilização de componentes de software (classes ou grupo de classes relacionadas) que é muito importante para aumentar a produtividade da equipe [Sie96]. A reutilização implica que o componente reutilizado seja confiável. Uma das formas de se garantir a confiabilidade, além de outras qualidades do componente, é através de testes.

Testes constituem uma atividade de validação que consiste em exercitar o software, fornecendo-lhe entradas, com o objetivo de achar falhas. Como em geral não é possível testar exaustivamente um programa, isto é, fornecer a este todas as entradas possíveis, deve-se escolher um conjunto finito de entradas que tenham boa capacidade de achar falhas. Essa escolha é baseada em critérios, os quais podem estar relacionados a um modelo do programa (tem-se então os testes estruturais ou caixa-branca), a um modelo das funcionalidades do sistema (tem-se então os testes funcionais ou caixa-preta), ou ainda a um modelo das falhas que podem ser cometidas pelos desenvolvedores (tem-se então os testes baseados em falhas). Os critérios determinam um conjunto de elementos do modelo que serão exercitados durante os testes.

O processo de teste envolve a geração, execução e avaliação dos testes [Pos96]. Em geral, isso demanda um esforço e custo adicionais ao desenvolvimento e manutenção do software ou componente, dependendo de quão testável é o componente, isto é, quão fácil é testá-lo.

Componentes reutilizáveis devem ser testados durante seu desenvolvimento, toda a vez que forem utilizados em um novo contexto, e a cada vez que sofram alguma alteração [PK90]. Dessa forma, é essencial que tais componentes sejam fáceis de testar para que os custos com a realização dos testes seja reduzida.

O teste de software orientado a objetos pode, algumas vezes, se beneficiar das características deste paradigma. Por exemplo, no teste de uma classe derivada pode ser considerado se a classe base já foi testada para reduzir o esforço, e assim, o custo do processo de teste. Contudo, algumas características do paradigma orientado a objetos

afetam a testabilidade dos sistemas, tais como herança, polimorfismo, ligação dinâmica, encapsulamento e ocultação da informação.

O polimorfismo com ligação dinâmica aumenta o número de possíveis caminhos de execução, tornando difícil a identificação e o teste de todos os caminhos. O encapsulamento reduz a testabilidade por reduzir a controlabilidade e observabilidade do estado interno da classe necessárias para os testes. A herança, como um dos principais mecanismos para a reutilização, faz com que a quantidade de testes a serem realizados aumente, além de levantar questões sobre o teste de classes derivadas.

Assim, embora o teste de software orientado a objetos possa utilizar técnicas de teste desenvolvidas para software procedimental, as diferenças entre estes paradigmas são importantes o suficiente para levar ao desenvolvimento de metodologias de teste específicas para o orientação a objetos [BS94].

1.1 Motivação

Fazendo-se uma analogia entre circuitos integrados e classes [Hof89], surgiu a proposta de se aplicar às classes, as técnicas de projeto para testabilidade (DFT, do inglês *Design for Testability*) e mais o conceito de autoteste, usados no projeto de circuitos VLSI. Tem-se com isso a proposta para a criação de classes autotestáveis [Bin94a]. Da mesma forma que no hardware, em que circuitos extras são embutidos em um componente voltados unicamente para funções de teste, mecanismos similares também devem ser acrescentados a uma classe. Diferentemente do hardware, esses mecanismos não precisam permanecer durante a fase operacional do sistema, sua introdução ou não podendo ser controlada através de diretivas de compilação.

No entanto, enquanto no hardware os testes gerados podem ser reaplicados a cada reuso do componente, no software isso nem sempre é possível pois, na maioria das vezes, os componentes sofrem alguma alteração quando são reutilizados. Para permitir que o reuso possa ser aplicável também aos testes, Harrold e McGregor propuseram a técnica *incremental hierárquica* [HM92], que leva em conta a hierarquia de herança, permitindo que, em alguns casos, seja possível reutilizar casos de testes gerados para a classe nos testes de suas classes derivadas.

1.2 Objetivos

O objetivo principal deste trabalho é verificar a viabilidade de se construir e utilizar classes autotestáveis, definindo todos os pontos em aberto nesta idéia, como o modelo de teste, a especificação de teste e os mecanismos de teste embutidos. Contudo, avaliar a eficiência da utilização de classes autotestáveis em relação a outras técnicas de teste de classes está fora do escopo deste trabalho.

Outro objetivo é combinar o conceito de autoteste com a técnica *incremental hierárquica*, melhorando a testabilidade de componentes reutilizáveis e permitindo a reutilização de casos de teste, sempre que possível.

Para mostrar a viabilidade de nossa proposta foi desenvolvida uma metodologia para construção e utilização de classes autotestáveis. Um protótipo de uma ferramenta que implementa parte desta metodologia também foi construído.

1.3 Terminologia

As definições dos termos **falha** (*fault*), **erro** (*error*) e **defeito** (*failure*) são utilizados na literatura de diversas formas [Mar98]. Neste trabalho, as definições estão de acordo com a terminologia introduzida em [LL87]. Uma falha é a causa direta de um erro. Um erro é a parte do estado do sistema que pode levar a um defeito. Por exemplo, se em uma instrução do software o sinal "<" foi trocado por ">", isto é uma falha. A execução desta instrução causa um erro e, se este erro leva o software a produzir um resultado incorreto, então um defeito é apresentado.

1.4 Organização da Dissertação

O capítulo 2 apresenta os conceitos básicos de teste de software. Ele apresenta os objetivos dos testes, as fases de teste, os componentes necessários para a sua realização e o processo envolvido nesta atividade. Também são apresentados a classificação dos testes segundo os critérios de seleção em: testes baseados em especificação, testes baseados em implementação e testes baseados em falhas. Em especial, é apresentada a técnica de teste fluxo de transação que é baseada em especificação e que será utilizada neste trabalho.

O capítulo 3 apresenta os conceitos de orientação a objetos e o impacto que as características das linguagens orientadas a objetos tem sobre a testabilidade de um sistema. A técnica incremental hierárquica é apresentada como uma das técnicas utilizadas no teste de classes derivadas. Algumas abordagens de teste de sistemas orientados a objetos e ferramentas dadas na literatura também são apresentadas. O capítulo apresenta ainda o conceito de testabilidade e os aspectos que influenciam a testabilidade de um sistema. São apresentadas a abordagem de Projeto visando Testabilidade (DFT, do inglês *Design for Testability*), sua aplicação em sistemas orientados a objetos e o conceito de classes autotestáveis.

A partir deste conceito foi elaborada a metodologia apresentada no capítulo 4 para construir e utilizar uma classe autotestável. É apresentada a especificação de teste definida para gerar testes utilizando a técnica de teste de fluxo de transação e a técnica incremental hierárquica visando a reutilização de testes. Também é apresentado como instrumentar a classe, acrescentando a ela os mecanismos de autoteste; e, como gerar e aplicar os casos de teste para a classe.

Parte desta metodologia foi automatizada através da ferramenta ConCAT apresentada no capítulo 5. Foi implementado um protótipo que faz a geração automática dos casos de teste e *driver* a partir da especificação de teste da classe. Neste capítulo são apresentados os passos para se utilizar ConCAT bem como seus componentes e características.

O capítulo 6 apresenta alguns trabalhos relacionados apresentados na literatura.

Finalmente, o capítulo 7 apresenta as conclusões deste trabalho, suas contribuições e sugestões de trabalhos futuros.

Capítulo 2

Testes de Software

Teste é uma das atividades de verificação e validação dinâmica de software que são utilizadas para garantir a qualidade do software produzido. Os testes são desenvolvidos em fases seguindo o desenvolvimento do software. Primeiro testa-se cada unidade individualmente, depois grupos de unidades, e então o sistema como um todo. Já o projeto dos testes é feito a partir de um modelo de base (código fonte, documento de projeto detalhado, especificação de requisitos ou o modelo de falhas cometidas durante o desenvolvimento do sistema), o qual classifica os testes em: baseados na implementação, na especificação ou em falhas. Um bom plano de teste deve envolver estes três tipos de testes pois cada um aborda diferentes tipos de falhas.

Em geral, o processo de teste demanda muito esforço e tempo, de modo que a testabilidade do sistema a ser testado se torna um fator de alta relevância para a redução do custo com testes. O conceito de testabilidade será apresentado no Capítulo 3.

A automação de todo ou parte deste processo é necessária devido ao grande número de casos de teste que devem ser gerados e aplicados ao software; muitas vezes estes testes devem ser repetidos até mais de uma vez e a realização manual se torna inviável.

Neste capítulo são apresentados conceitos de testes de sistemas procedimentais importantes para o desenvolvimento e compreensão do trabalho. Testes de sistemas orientados a objetos serão tratados no Capítulo 3. Na seção 2.1 são apresentados os objetivos dos testes. Na seção 2.2 são descritas as fases de teste para sistemas procedimentais e na seção 2.3, os conceitos de *driver* e *stub*. Na seção 2.4 é apresentada a classificação dos testes segundo os critérios de seleção e na seção 2.5, o modelo de teste de fluxo de transação, uma técnica de teste baseada na especificação. Finalmente, na seção 2.6 é apresentado o processo de teste e na seção 2.7, um sumário do capítulo.

2.1 Objetivos

Teste é uma das atividades para garantia da qualidade de software, seu principal objetivo é detectar falhas [Pre95] e não provar que o software está correto. Se na execução dos testes nenhuma falha foi descoberta, isso não quer dizer que o software não contenha falhas. Os testes também podem ser utilizados para obter medidas da qualidade de software. A realização dos testes é importante para complementar outras formas de verificação e validação, pois é a única que permite exercitar o comportamento operacional do programa [Mar98].

A atividade de teste de software combina uma estratégia de múltiplos passos com uma série de técnicas de projeto de casos de teste que auxiliam a garantir uma efetiva detecção de falhas [Pre95].

2.2 Fases de Teste

Testes são divididos em fases, sendo que os casos de teste para cada fase são preparados (planejados e especificados) em uma determinada fase do desenvolvimento de software.

Para sistemas tradicionais são consideradas as seguintes fases de teste:

- Testes de unidades
  Uma unidade, de acordo com o padrão IEEE 610-12-1990, é um componente de software que não pode ser dividido, por exemplo uma função ou subprograma. Testes de unidades visam mostrar que a unidade testada não satisfaz a funcionalidade especificada e/ou que o código implementado não está de acordo com o definido no projeto detalhado [Bei90].

- Testes de integração
  Apesar de serem testadas isoladamente, as unidades devem ser testadas novamente ao serem integradas para formar módulos ou subsistemas. Estes testes visam mostrar a existência de falhas nas interações entre as unidades quando estas são integradas [Bei90].

- Testes de validação
  Estes testes visam validar os requisitos do sistema levantados na análise de requisitos [Pre95].

- Testes de sistema
  Os testes de sistema têm como objetivo testar comportamentos que são expostos somente testando o sistema todo, incluindo hardware e software. Ele inclui teste de desempenho, segurança, recuperação, estresse e inicialização [Bei90].

Um tipo de teste muito utilizado nos testes de integração são os testes de regressão. Testes de regressão consistem na re-execução de casos de teste já aplicados quando ocorre alguma alteração no sistema [Mar98]. Eles auxiliam a verificar se o comportamento do sistema foi alterado somente conforme o exigido pelas alterações já realizadas.

Uma estratégia de teste define a ordem em que estas fases serão realizadas. A mais utilizada diz que os testes devem iniciar com os testes de unidades, prosseguindo pelos testes de integração das unidades já testadas isoladamente, os testes de validação com todos os componentes já integrados, e então os testes de sistema quando o sistema é integrado com hardware e ambiente onde ele será implantado.

2.3 *Drivers* e *Stubs*

Para os testes de unidades e de integração é necessário o desenvolvimento de componentes de software que auxiliem a realização dos testes pois uma unidade não é um programa individual que pode ser executada isoladamente [Pre95]. Estes componentes de software são *drivers* e *stubs*.

Drivers geralmente são um tipo de "programa principal" que executa os casos de teste sobre a unidade e retorna os resultados. Também é função do *driver* disponibilizar à unidade as estruturas de dados que ela utilize.

Stubs são componentes que substituem os módulos chamados pela unidade sob teste, logo devem ter a mesma interface que esses módulos. A implementação de um *stub*, contudo, pode variar de um processamento simples como a exibição de uma mensagem de rastreamento, a um processamento mais complexo como o mapeamento de um parâmetro de entrada a um valor de retorno associado [Pre95].

A construção de *drivers* e *stubs* representa um esforço e custo adicionais aos testes, de modo que algumas estratégias visam minimizar sua utilização, por exemplo testando primeiro as unidades que oferecem serviços para depois testar aquelas que utilizam tais serviços. Um outro modo de simplificar a fase de testes de unidades é projetando unidades coesas, pois estas geralmente não necessitam dos serviços oferecidos por outras unidades.

2.4 Classificação dos Testes Segundo os Critérios de Seleção

Uma das grandes dificuldades em testes é escolher um subconjunto de entradas com maior probabilidade de encontrar o maior número possível de falhas [Mye79, Mar98], dado que é impossível testar exaustivamente um programa, ou seja, aplicar todas as entradas possíveis para ele. Os critérios de seleção de testes tentam solucionar este problema estabelecendo condições que devem ser satisfeitas durante os testes. Um critério de seleção determina um conjunto finito de elementos do modelo que devem ser exercitados durante os testes [Mar98]. Este modelo, chamado de modelo de base, pode ser a especificação do sistema, o

código fonte ou um modelo das falhas cometidas durante o desenvolvimento do sistema. A partir destes modelos são abstraídos modelos de teste [Sie96], pois nem sempre se pode testar diretamente tais modelos de base. Com modelos de teste é mais fácil identificar as possíveis falhas e gerar casos de teste que as detectem.

De acordo com estes modelos podemos classificar os métodos de seleção de testes em:

- testes baseados na especificação
  Nestes testes o programa ou sistema é testado como uma caixa preta, sujeita a entradas e cujas saídas são verificadas contra o comportamento especificado. É considerado o ponto de vista do usuário pois as funcionalidades e características do programa são enfocadas ao invés de detalhes de implementação. O objetivo destes testes é determinar se todos os requisitos foram implementados. Assim, são utilizados como modelo de base para derivação dos casos de teste: a especificação de requisitos, especificação de projeto ou até uma descrição informal das funcionalidades do sistema.

- testes baseados na implementação
  O teste baseado em implementação ou de caixa branca observa detalhes de implementação, tais como: estilo de programação, métodos de controle, linguagem fonte, projeto da base de dados e detalhes de codificação. O modelo de base para derivação de casos de teste é o código fonte do sistema.

- testes baseados em falhas
  Testes baseados em falhas buscam detectar defeitos do programa (um valor incorreto produzido na execução do programa) quando não existe um modelo de falhas; ou analisar se um determinado conjunto de casos de teste pode detectar as falhas descritas no modelo de falhas do programa. Um exemplo desta última utilização são os testes de mutação [Mar98].

Não existem contradições entre a utilização destes tipos de testes pois eles têm como alvo diferentes tipos de falhas. A arte de testar, em parte, está em escolher entre esses testes.

Neste trabalho são abordados testes funcionais, em particular, a técnica chamada teste de fluxo de transação que será detalhada na próxima seção.

2.5 Teste de Fluxo de Transação

Esta é uma técnica de teste baseada na especificação definida por B. Beizer [Bei90, Bei95] para os testes de sistemas concorrentes e adaptada em [Sie96] para o teste funcional de classes.

Um modelo de fluxo de transação mostra o fluxo de trabalho para o sistema ou objeto. No contexto de uma classe, um fluxo de trabalho pode ser definido como a criação,

o processamento e a destruição do objeto. Assim, cada modelo de fluxo de transação consiste dos diferentes modos de criar um objeto, as diferentes tarefas ou processos que o objeto realiza durante seu ciclo de vida e os diferentes modos que este objeto pode ser destruído. No contexto de um sistema procedimental, um programa é visto como uma coleção de funções que interagem e um fluxo de trabalho é uma seqüência de uma ou mais funções executadas pelo sistema.

Um modelo de fluxo de transação é representado através de um grafo, de modo que uma transação é um caminho através deste grafo. Para um objeto, os arcos representam a seqüência de execução e cada nó representa a execução de um método. O nó inicial representa a construção de um objeto e o nó final, a sua destruição. Pode haver múltiplos construtores e/ou destrutores, podendo haver então mais de um nó inicial ou final.

O grafo de fluxo de transação é construído a partir dos casos de uso da classe ou dos requisitos do sistema. Passos para construção do grafo são dados em [Sie96] e [Bei95]. A principal dificuldade na construção do grafo para sistemas procedimentais é a identificação de todas as transações pois algumas funções estão implícitas dentro de módulos maiores [Bei95, How86]. Contudo, em sistemas orientados a objetos esta dificuldade é contornada por considerar cada método da classe como uma função.

Construído o grafo, os testes são derivados percorrendo o grafo. A existência de um oráculo, como em outros tipos de teste, é imprescindível. Oráculos são fontes externas de informação sobre as funções [How86]. Segundo Howden [How86], existem três tipos básicos de oráculo: oráculos de entrada/saída, oráculos de traço e oráculos de interface. Um oráculo de entrada/saída é um mecanismo que determina se uma saída obtida é correta ou não para uma dada entrada. Um oráculo de traço é capaz de determinar se uma seqüência de chamadas de funções é correta. Um oráculo de interface é usado para determinar se a transformação de um tipo de dado em outro é correta. Fazendo-se um paralelo com a análise funcional de [How86], para o teste de fluxo de transação pode ser utilizado um oráculo de traço.

2.6 Processo de Teste

O processo de teste inclui planejar, identificar, especificar, executar, analisar e finalizar os testes, como ilustrado pela figura 2.1 [Jac92].

O planejamento dos testes deve considerar os padrões que serão utilizados para os testes e os recursos requeridos para cada tipo de teste (de unidades, de integração, etc.). O plano não deveria controlar o teste em detalhes, mas servir como uma base para as atividades de teste.

Com a identificação do que deve ser testado pode-se estimar com maior precisão os recursos requeridos, o que auxilia na especificação e execução dos testes.

As especificações de teste oferecem instruções de execução detalhadas, de modo que pessoas não familiarizadas com a aplicação ou sistema possam executar os casos de teste. Uma especificação deve conter como o teste deve ser feito, em que ordem, os resultados

esperados, critérios para término dos testes, além das condições de teste, tais como: ambiente de execução, equipamentos e *drivers*.

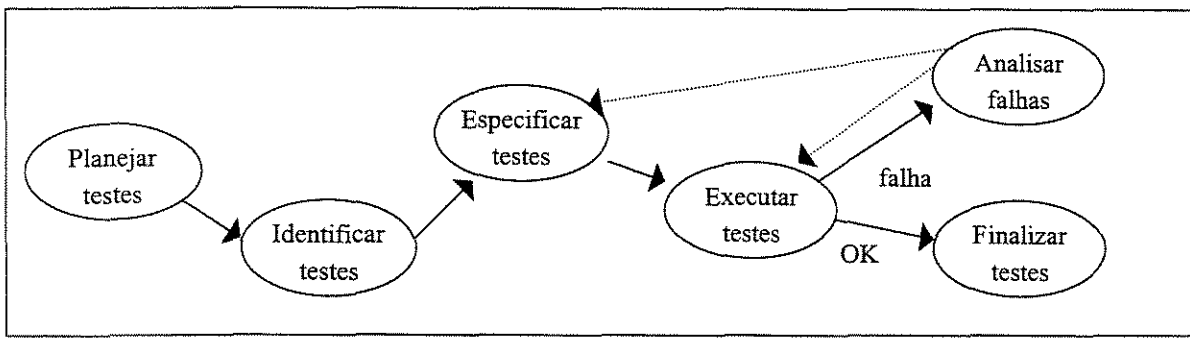


Figura 2.1: Processo de teste.

Na execução dos testes são utilizadas a especificação destes e gerados relatórios de teste. Durante a execução, se algum teste detecta uma falha a execução é interrompida e o resultado anotado, então o defeito é analisado e a falha corrigida se possível.

Se são detectados defeitos quando o teste é feito, então ele deve ser analisado e a falha que o causou identificada. Depois de corrigida a falha, o caso de teste deve ser aplicado novamente.

Ao término dos testes os equipamentos e *drivers* deveriam ser armazenados para uso futuro, assim como toda documentação preparada para os testes.

Embora programadores, testadores e gerentes saibam que o código deva ser projetado e testado, muitos não parecem estar cientes que os testes em si também devam ser projetados e testados. Se os testes não são formalmente especificados, fica mais difícil repetí-los. Desse modo, depois que uma falha é corrigida, não há como se certificar que o teste de regressão foi idêntico aos testes que descobriram essa falha [Bei90].

Uma pergunta que sempre surge quando se realiza a atividade de teste é como saber se foram realizados testes suficientes. Ainda não existe resposta definitiva para essa dúvida, somente algumas respostas pragmáticas e algumas primeiras tentativas empíricas [Pre95]. A compilação de métricas durante o teste do software e o uso dos modelos de confiabilidade existentes permitem desenvolver diretrizes significativas para responder a essa pergunta. Uma das métricas utilizada é a cobertura dos testes em relação a um critério de seleção, a qual é dada normalmente em forma de porcentagem. A cobertura pode ser calculada antes dos testes para obter o nível de cobertura desejado ou após os testes para verificar a cobertura do modelo obtida pelos testes executados.

2.7 Sumário

Neste capítulo foram apresentados alguns conceitos relacionados aos testes de sistemas procedimentais que são fundamentais para a compreensão do trabalho.

Na seção 2.1 foi apresentado que o objetivo principal dos testes é detectar falhas no software e não provar que ele está correto. Na seção 2.2 foi apresentado que em sistemas procedimentais os testes são gerados e aplicados em fases. Inicialmente são aplicados testes sobre cada unidade, depois testes de integração sobre as unidades já testadas, e testes de validação e de sistema sobre o sistema completo. Durante os testes de integração é freqüente o uso de testes de regressão que são a reaplicação de casos de testes após alguma modificação no programa. Na seção 2.3 foram abordados os conceitos de *driver* e *stub*, que são componentes necessários para a realização de testes de unidades e de integração.

Na seção 2.4 foi apresentado que os testes podem ser classificados segundo os critérios de seleção em: testes baseados na especificação, testes baseados na implementação e testes baseados em falhas. O teste de fluxo de transação, descrito na seção 2.5, é uma técnica de teste baseada na especificação e será utilizada neste trabalho.

Finalmente, na seção 2.6 foi descrito o processo de teste que vai desde o planejamento até a execução dos casos de teste e análise dos resultados.

Capítulo 3

Teste de Software Orientado a Objetos

A orientação a objetos permite a criação de programas modulares, de modo que cada módulo encapsula função e estado na forma de procedimentos e variáveis [LRP+95]. Este conceito de encapsulamento, mais os conceitos de polimorfismo, ligação dinâmica e herança podem definir o paradigma orientado a objetos. A utilização deste paradigma favorece principalmente a reutilização de componentes de software (classe ou grupos de classes relacionadas).

A utilização do paradigma orientado a objetos produz sistemas bem projetados, porém isso não assegura a produção de sistemas corretos [BS94]. Assim, é necessário 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 para tanto devem ser testados para garantir o nível de confiança desejado.

Em geral, testar um software demanda muito esforço e tempo, de modo que a testabilidade do sistema a ser testado se torna um fator de alta relevância para a redução do custo com testes. A crescente importância da testabilidade como fator da qualidade de um software [Pre95] se justifica porque quanto maior a testabilidade do sistema, menores serão as dificuldades para realização do processo de teste.

No teste de software orientado a objetos as dificuldades aumentam devido a características como herança, polimorfismo com ligação dinâmica, encapsulamento e ocultação da informação. Além disso, no caso de reutilização de classes a quantidade de testes aumenta, pois a classe deve ser testada novamente a cada vez que é reutilizada em um novo contexto e testes novos devem ser construídos e executados para testar características novas ou modificadas. A fim de reduzir a quantidade de testes necessária, principalmente devido aos testes de regressão, existem duas possibilidades [VM95]:

- selecionar testes com maior probabilidade de revelar falhas, ou
- projetar o software que tenha maior probabilidade de apresentar defeitos quando uma falha existir.

Neste trabalho foi considerada a segunda possibilidade e para isso foram utilizadas técnicas de Projeto para testabilidade (DFT, do inglês *Design for Testability*).

Este capítulo tem como objetivo apresentar as dificuldades que surgem no teste de sistemas orientados a objetos, os aspectos de um software que influenciam em sua testabilidade, algumas abordagens de DFT para sistemas orientados a objetos e o conceito de classe autotestável, assim como algumas abordagens e ferramentas para o teste de sistemas orientados a objetos. Ele está organizado do seguinte modo: na seção 3.1 são apresentados os principais conceitos de orientação a objetos; na seção 3.2 são apresentadas as principais dificuldades para se testar um software orientado a objetos; na seção 3.3, são apresentadas as abordagens para teste de sistemas orientados a objetos FREE [Bin94b], hierárquica [Sie96] e Ordem de Teste [KGH97]; na seção 3.4 são descritas algumas ferramentas de teste de sistemas orientados a objetos; na seção 3.5 são apresentadas propostas para a melhoria da testabilidade de sistemas orientados a objetos, e finalmente, na seção 3.6 é apresentado um sumário do capítulo.

3.1 Conceitos de Orientação a Objetos

Nesta seção são apresentados os conceitos básicos de orientação a objetos. As definições foram baseadas nas terminologias dadas por [RBP+94], [Sie96], [KGH+95] e [BR98].

- **Objeto**
  Objetos são blocos de construção básicos de um software orientado a objetos. Um objeto modela uma entidade ou coisa no domínio de aplicação. Uma das definições para objeto é como um conceito, uma abstração, algo com limites nítidos e significado em relação ao problema em causa [RBP+94]. Um objeto possui um conjunto de propriedades cujos valores definem seu estado e um conjunto de operações sobre suas propriedades que definem seu comportamento e interface pública. Entidades externas e outros objetos não podem acessar diretamente o estado interno de um objeto, somente através do envio de mensagens [BR98]. Cada objeto possui ainda uma identidade que pode ser usada para identificá-lo unicamente ou distinguí-lo de objetos similares [KGH+95].

- **Classe**
  Uma classe é uma abstração que define o tipo ou estrutura de um conjunto de objetos similares, ou seja, ela define as propriedades e o comportamento que compõem os objetos [KGH+95]. Um objeto é instância de apenas uma classe. As propriedades do objeto são chamadas atributos e a implementação de uma operação é chamada método. Em C++ atributos e métodos são chamados dados e funções membro, respectivamente.

- **Encapsulamento**
  O encapsulamento consiste na separação dos aspectos externos de um objeto, acessíveis

por outros objetos, dos detalhes de implementação [RBP+94]. Os dados estão inseridos no objeto e somente os métodos do objeto têm acesso a eles. Esta característica facilita o projeto de objetos fracamente acoplados e fortemente coesos [Sie96]. Desse modo, um objeto pode ter sua implementação alterada sem que isso afete as aplicações que o utilizam.

- **Ligação dinâmica**
  A ligação dinâmica implica na determinação em tempo de execução de qual operação será executada dependendo do nome da operação e do tipo do objeto.

- **Polimorfismo**
  Polimorfismo significa a habilidade de tomar mais de uma forma: um atributo pode ter mais de um conjunto de valores e uma operação pode ser implementada por mais de um método [KGH+95]. Assim, dois ou mais objetos podem realizar de formas diferentes a mesma operação.

  Dentre as formas de polimorfismo estão a reescrita e a sobrecarga. A reescrita permite a redefinição de uma operação em uma subclasse mantendo a mesma assinatura da operação [Sie96]. A sobrecarga permite que um nome de função seja utilizada mais de uma vez porém com parâmetros de tipos diferentes. O sistema em tempo de execução ligará corretamente a chamada do método ao objeto correspondente através da ligação dinâmica.

- **Herança**
  Herança é um tipo de relacionamento que define uma hierarquia entre classes, permitindo que elas possam compartilhar atributos e operações [RBP+94]. 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 acrescenta características próprias e exclusivas. Existem dois usos principais do mecanismo de herança [BR98]:

  - Herança de implementação: utilizada para implementar um tipo abstrato de dados similar a outros já existentes. Por exemplo, implementar uma classe pilha a partir de uma classe lista. Este uso da herança não visa garantir que a classe derivada tenha o mesmo comportamento da classe base. Isto pode levar a um comportamento incorreto da classe derivada pois ela pode herdar um comportamento não desejado da classe base.

  - Herança de tipos ou comportamento: utilizada para construir uma hierarquia de tipos, com subtipos (classes derivadas) e supertipos (classes base). Um subtipo deve fornecer pelo menos o comportamento de sua classe base. Por exemplo, criar uma classe gerente a partir de uma classe pessoa. Um gerente possui as mesmas características que uma pessoa e algumas propriedades ou comportamento adicionais. Este tipo de herança implementa relacionamentos de generalização/especialização entre classes.

O conceito de herança múltipla surge quando uma classe é derivada a partir de mais de uma classe base. Por exemplo, monitores de disciplinas possuem características das classes estudante e professor [KGH+95].

- **Classes abstratas**
  Uma classe abstrata é aquela que não possui instâncias diretas, ao contrário de uma classe concreta [RBP+94]. Classes abstratas organizam características comuns a diversas classes. Geralmente, são utilizadas para definir uma interface padrão para suas classes derivadas ou um protocolo para uma operação sem fornecer uma implementação para ela; esta operação é chamada de operação abstrata.

3.2 Dificuldades no Teste de Software Orientado a Objetos

A tecnologia orientada a objetos tem sido utilizada no processo de desenvolvimento de software para aprimorar a qualidade do produto sob determinadas restrições de custo e recursos humanos. Desse modo, muitos benefícios têm sido obtidos, tais como redução do tempo de desenvolvimento e maior reutilização. Alguns aspectos da tecnologia orientada a objetos podem auxiliar os testes, tal como permitir a redução do esforço necessário para testar classes derivadas depois que sua classe base foi testada. Mas, as poderosas características deste novo paradigma também introduzem algumas dificuldades. Segundo [KGH+95, SN94, BS94] estas dificuldades podem ser resumidas como:

- **Encapsulamento e ocultação da informação**
  Estas características fazem com que o único modo de se observar o estado de um objeto seja através de suas operações, o que reduz a capacidade de observação requerida durante os testes.

  Elas levam também ao chamado problema do entendimento. Uma funcionalidade pode ser implementada através da chamada de métodos de outros objetos, que por sua vez podem chamar outros, produzindo a chamada cadeia de invocação de métodos. Para se testar tal funcionalidade é necessário entender as seqüências de métodos e as funcionalidades das classes envolvidas antes de preparar os casos de teste, o que aumenta a complexidade dos testes.

- **Classes parametrizadas e abstratas**
  Classes desses tipos não podem ser instanciadas diretamente pois lhes faltam informações para isso, logo não é possível testá-las diretamente também. O conjunto de casos de testes mínimo desenvolvido deve poder ser utilizado com as diferentes construções para as informações que faltam nestas classes: parte da implementação, no caso de classes abstratas, ou componentes ainda não especificados, no caso de classes parametrizadas. Este problema ainda é um tópico de pesquisa.

- **Herança**
  O mecanismo de herança é o que mais contribui para o reuso. Como a classe derivada herda características e operações da(s) classe(s) base, parece natural que se a classe base foi devidamente testada então as propriedades herdadas não precisam ser testadas novamente, mas isso não é verdade. É difícil estabelecer um método para decidir quais propriedades herdadas precisam ser testadas novamente [BS94], pois isso depende do mecanismo de herança utilizado e, como dito na seção 3.1, este mecanismo pode seguir esquemas diferentes, como citados a seguir:

  - Herança estrita: as propriedades herdadas pela classe derivada não podem ser modificadas (reescritas), somente podem ser adicionadas novas propriedades. Porém, a adição de novas operações e características podem levar o objeto a um estado ainda não testado para as operações herdadas. Neste caso, elas precisam ser testadas.

  - Especialização: neste caso, as propriedades herdadas podem ser reescritas e novas propriedades podem ser adicionadas. Naturalmente, as operações reescritas devem ser re-testadas na classe derivada, assim como as operações que as invocam em suas implementações, pois estas operações tiveram seus comportamentos alterados.

  - Herança de implementação: neste esquema, a classe derivada não é considerada uma especialização da classe base, mas uma nova classe cujo comportamento se baseia sobre parte do comportamento de outra classe (a classe base). Isto implica que a classe derivada não precisa herdar todas as características da classe base. Assim, os casos de testes já definidos para a classe base devem ser alterados para suprimir as referências às propriedades não herdadas.

Duas técnicas estão sendo utilizadas para o teste de classes derivadas:

- Técnica baseada no aplanamento da classe
  Nesta abordagem todas as características da classe, tanto novas quanto herdadas ou redefinidas, são consideradas no desenvolvimento do modelo de teste. Uma classe é aplanada (do inglês, *flattened*) adicionando à sua especificação, os atributos e métodos de sua classe base. A principal desvantagem desta abordagem é exigir o reteste completo de classes derivadas, ainda que muitos métodos tenham sido testados na classe base e herdados sem qualquer modificação.

- Técnica baseada na hierarquia de herança
  A técnica incremental hierárquica é uma técnica de teste que explora a natureza hierárquica da relação de herança para testar grupos de classes relacionadas, reutilizando a informação de teste de uma classe base para orientar o teste de uma classe derivada [HM92]. A técnica é dita hierárquica porque é orientada pela ordenação parcial da relação de herança, e incremental porque utiliza os resultados

dos testes de uma classe em um nível na hierarquia para reduzir os esforços necessários em níveis subsequentes. Assim, para projetar a seqüência de teste para uma classe derivada, as informações de teste da classe base são incrementalmente atualizadas para refletir atributos e métodos novos ou redefinidos.

São definidos três tipos de métodos e cada tipo exige uma quantidade de teste diferente, como ilustrado na tabela 1 [Sie96]. É assumido que a especificação ou funcionalidade de métodos redefinidos na classe derivada permaneça a mesma.

Métodos recursivos podem exigir que novos casos de teste sejam construídos para testar eventuais mudanças em suas pré-condições ou pós-condições. Por exemplo, se um método recursivo usa um atributo e a classe derivada acrescenta um novo intervalo de valores para este atributo, então é necessário criar novos testes para este novo intervalo.

A principal vantagem desta abordagem é a redução do esforço de testar novamente cada classe derivada [HM92], pois além de reutilizar a informação de teste da classe base, pode-se também reutilizar seus casos de teste para testar a classe derivada.

Métodos	Descrição	Teste
Novos	Definidos na classe derivada, inclusive aqueles com nome igual ao de métodos na classe base, mas com parâmetros diferentes.	Teste completo.
Recursivos	Definidos na classe base e não sobrecarregados ou reescritos na classe derivada.	Teste somente se o método interage com métodos novos ou redefinidos.
Redefinidos	Definidos na classe base e sobrecarregados ou reescritos na classe derivada.	Teste reutilizando os modelos de teste funcionais da classe derivada.

Tabela 1: Tipos de métodos e teste exigido por cada tipo.

Esta abordagem assume um modelo de linguagem que possua três níveis de visibilidade, como em C++ (*protected, private* e *public*), e cujo mecanismo de herança faça o mapeamento de um atributo da classe base a um atributo na classe derivada com um nível de visibilidade igual ou mais restrito ao que ele possuía. Por exemplo, um método que é público na classe base pode ser público, protegido ou privado na classe derivada, já um método que é protegido na classe base não poderá ser um método público na classe derivada.

- **Problema de interdependência complexa**

  Entre classes é comum existirem relacionamentos complexos, tais como: herança, agregação, associação, instanciação de classes *templates*, aninhamento de classes, criação dinâmica de objetos, invocação de métodos, polimorfismo e ligação dinâmica. Estes relacionamentos dificultam os testes, pois:

- é difícil entender uma classe se ela depende de muitas outras classes;
- o testador pode ficar confuso e não saber onde iniciar os testes em uma biblioteca;
- é muito caro construir *stubs*;
- teste de objetos complexos, ou seja, objetos compostos por outros objetos, implica em saber instanciar os objetos componentes. Além disso, objetos complexos possuem um grande número de estados possíveis pois estes são definidos pelos estados dos objetos que os compõem [SN94];
- é difícil identificar e testar todos os efeitos do polimorfismo e ligação dinâmica.

- **Problema de testar a variação do estado do objeto**
  O efeito de uma operação sobre um objeto depende do estado do objeto e pode modificar este estado. Assim, o efeito combinado de operações deve ser testado, observando o estado do objeto durante a execução destas operações. Além disso, a adição de uma nova operação pode causar impacto sobre as operações existentes e torná-las inúteis. A nova operação pode levar o objeto a um estado inapropriado para as demais. Portanto, cada mudança na implementação de um objeto pode levar a um reteste completo do objeto e não só da operação modificada ou incluída.

- **Problema de ferramentas de suporte**
  A maior parte das ferramentas comerciais existentes implementam métodos de teste convencionais. Embora possam ser utilizados, estes métodos não são totalmente adequados ao teste de sistemas orientados a objetos.

Estes problemas introduzidos por características do paradigma orientado a objetos não são triviais. Alguns deles já foram solucionados, outros ainda representam questões em aberto.

3.3 Algumas Abordagens para o Teste de Software Orientado a Objetos

Nesta seção são apresentadas algumas abordagens para o teste de software orientado a objetos e como elas propõem o uso de diferentes métodos de teste.

3.3.1 FREE (Flattened Regular Expressions)

FREE [Bin94b] é uma abordagem para desenvolver seqüências de teste para software orientado a objetos. Esta abordagem utiliza a técnica baseada no aplanamento da classe, explicada na seção anterior, para testar classes derivadas. Ela cobre os seguintes tipos de testes:

- Teste de unidades
  Para o teste funcional de classes, são gerados casos de teste a partir de um modelo de estados da classe representado através de uma máquina de estados. Estados representam estados legais do objeto e cada transição possui um evento e uma ação associados. Eventos são estímulos externos à classe e ações representam a resposta do objeto a esses estímulos.

  No teste estrutural da classe, primeiramente é construído, para cada método da classe, um grafo que envolve fluxo de dados e controle. Posteriormente estes grafos são agrupados junto com o modelo de estados da classe, formando o grafo de fluxo da classe. É a partir deste grafo que são derivados os casos de teste.

- Testes de integração
  Para os testes de integração de classes para as quais é possível determinar seqüências válidas de invocação de métodos, é construído um modelo de modos. Este modelo é construído a partir dos modelos de estado de cada classe. Para classes que não possuem restrições quanto a invocação de seus métodos, casos de teste são derivados a partir de expressões regulares representando o fluxo de dados de interface entre métodos.

- Testes de sistema
  No teste de sistema o plano de teste é elaborado a partir dos cenários de uso estendidos que contêm informações adicionais aos casos de uso dados na especificação do sistema. Tais informações incluem: fatores quantificáveis que determinam comportamento do usuário (por exemplo, quantidade deve ser maior que zero, ao invés de, quantidade deve ser um número válido) e entradas/saídas visíveis; quais operações são mais utilizadas, e cenários associados com cada operação.

3.3.2 Abordagem Hierárquica

A abordagem hierárquica [Sie96] é baseada na aplicação de várias técnicas de teste em componentes de software. São definidos componentes para vários níveis de teste, por exemplo um objeto, uma classe, um componente básico ou um sistema. Um componente básico pode ser uma hierarquia de classes completa ou algum conjunto de classes que execute uma função importante no sistema.

Cada componente de software é testado separadamente até atingir um determinado nível de confiança estabelecido pela equipe de teste. Quando o componente alcança este nível, ele pode ser integrado com outros componentes que já alcançaram o nível de confiança desejado, para então compor os componentes do próximo nível. Desse modo, inicialmente cada método é testado individualmente. Depois são testadas classes, que se integrarão para formar componentes básicos, que por sua vez serão testados e se integrarão para formar componentes do sistema. E assim sucessivamente até que se tenha o sistema completo e o teste de sistema possa ser realizado.

Para o teste de integração dos componentes básicos é necessário testar somente as

interconexões entre eles, eliminando a necessidade de testar todas as possíveis combinações de estados.

Nesta abordagem os esforços de teste se concentram no teste de componentes de software de menor granularidade. Assim, no teste de sistema o esforço gasto para os testes é mínimo, uma vez que se pode reutilizar os casos e seqüências de teste já definidos anteriormente.

A abordagem hierárquica pode ser integrada a qualquer processo de desenvolvimento de software, mas o ideal é aplicá-la em um processo de desenvolvimento incremental.

3.3.3 Ordem de Teste (Test Order)

Ordem de teste [KGH97] é uma estratégia para testes de unidades e de integração. Ela visa encontrar a ordem de teste ótima para que o esforço requerido na construção de *stubs* e *drivers* seja mínimo.

A estratégia é baseada no ORD (do inglês, *Object Relation Diagram*), um grafo direcionado cujos vértices representam as classes de objetos e os arcos representam os relacionamentos entre as classes, tais como: herança, associação, agregação, instanciação (de uma classe *template*), aninhamento de classes e uso (para instanciar uma classe *template*). Um algoritmo computa a ordem ótima em que as classes devem ser testadas baseado nas dependências indicadas pelo ORD. Este diagrama é obtido automaticamente a partir do código fonte, caso ele esteja em C++, através de uma ferramenta que leva o mesmo nome. A ordem para integração das classes é idêntica à ordem usada para o teste de classes.

A ordem de teste implica em um reuso efetivo dos casos de teste já gerados e complementa a idéia de [HM92] onde o reuso se limitava aos relacionamentos de herança, pois, considera também outros tipos de relacionamento entre classes.

3.4 Ferramentas de Teste de Sistemas Orientados a Objetos

Nesta seção são apresentadas algumas ferramentas de teste estudadas durante o trabalho. Elas não representam trabalhos relacionados, mas exemplificam outros trabalhos desenvolvidos na área de testes de sistemas orientados a objetos.

3.4.1 FOOT (Framework Object-Oriented Testing)

Este *framework* [SR92] permite algum processamento da informação estática presente na classe a fim de permitir a execução estruturada das rotinas quando a classe é instanciada em tempo de execução. Esta informação sobre a classe é obtida através de um *parser* e inclui

como criar e destruir objetos, quais métodos retornam objetos e quais permitem o acesso ao estado da classe, por exemplo. Desta informação é gerado um conjunto das possíveis combinações dos métodos da classe. Além destas, outras informações devem ser providas, tais como quais rotinas estão relacionadas as funções que retornam o valor ou atribuem um valor para um atributo e quais rotinas realizam funções inversas (por exemplo, incrementar e decrementar uma variável inteira por uma constante).

FOOT possui os seguintes componentes:

- Um *driver* que controla o processo de teste e é responsável pela instanciação e invocação de TOE (*Test Object Exerciser*), apresentado abaixo;
- A classe sob teste;
- Os arquivos de informações da classe, que armazenam também informações sobre a execução dos testes e os resultados obtidos;
- As estratégias que orientam o processo de seleção de casos de teste provendo funções que retornam o próximo caso de teste, determinam se ele foi completo e se a seqüência de teste foi completa, e
- TOE, construído a partir da informação da classe e da estratégia de teste selecionada, possui ligações com o *driver* para permitir o retorno dos resultados e a seleção de características. É responsável por aplicar os casos de teste, gerados de acordo com uma determinada estratégia de teste, sobre a classe.

3.4.2 OOTM (Object-Oriented Software Testing and Maintenance Environment)

Este ambiente [KGH+95] possui um modelo de teste que consiste de três diagramas:

- diagrama de relacionamento de objetos (ORD, do inglês *object relation diagram*): representa os relacionamentos de herança, agregação, associação, instanciação de classes *templates*, uso e aninhamento de classes. Ele serve de base para indicar a ordem em que as classes devem ser testadas, de acordo com o nível de dependência entre elas.

- diagrama de ramificação de blocos (BBD, do inglês *block branch diagram*): representa a estrutura de controle de um método e suas interfaces com outros métodos, de modo que o testador saiba quais dados são utilizados e/ou atualizados e quais métodos são invocados por aquele método.

- diagrama de estados do objeto (OSD, do inglês *object state diagram*): similar ao diagrama de estados da metodologia OMT [RBP+94], representa os estados de um objeto. Para cada atributo que pode ser alterado por algum método da classe é construído uma máquina de estados, representando o comportamento deste atributo. O diagrama da classe é representado pela agregação dos diagramas dos atributos.

Cada um destes diagramas é derivado do código da classe por ferramentas que compõem o ambiente. OOTM possui ainda uma ferramenta, chamada *firewall*, que automaticamente identifica o efeito de mudanças e indica as classes afetadas. Estas ferramentas também geram dados de teste.

3.4.3 ClassBench

A metodologia de grafos de teste [HS95, HS97] foi desenvolvida para o teste de classes com interfaces baseadas em chamadas ao invés de classes de interfaces gráficas que utilizam mouse, teclado, vídeo ou arquivos para entrada e saída de dados. Nesta metodologia são definidas três tarefas a serem executadas pelo testador:

- Desenvolver o grafo de teste para a classe sob teste
  O grafo de teste é um grafo direcionado cujos nós representam estados da classe e os arcos correspondem a uma seqüência de uma ou mais transições de estado da classe. Assim, este grafo é menor que o grafo de estados/transições completo da classe, pois os estados representados não cobrem todos os estados possíveis para a classe sob teste, mas somente aqueles que serão alcançados na execução da seqüência de teste.
  Os casos de teste são gerados percorrendo o grafo a partir do nó inicial especificado. São considerados três tipos de cobertura para o grafo: todos os nós, todos os arcos e todos os caminhos. A cobertura dos arcos inclui a cobertura dos nós, já a cobertura de todos os caminhos pode ser impossível de se obter caso o grafo seja cíclico.

- Implementar a classe oráculo
  Esta classe provê as mesmas operações que a classe sob teste, porém ela suporta somente os estados e transições do grafo de teste. As variáveis de estado do oráculo representam os nós do grafo de teste e seus métodos realizam as transições, auxiliando a verificar se o comportamento da classe sob teste está de acordo com o grafo.

- Implementar a classe *driver*
  Esta classe percorre o grafo de teste. Ela provê operações para gerar transições associadas com arcos e verificar os estados associados com os nós através de chamadas aos métodos da classe oráculo.

O *framework ClassBench* [HS97] provê suporte automatizado para esta metodologia, facilitando o desenvolvimento das seqüências de teste e a reutilização destas para os testes de regressão. Ele possui um editor e um algoritmo de percurso de grafos de teste, *templates* de seqüências de teste e controle da configuração de teste.

3.5 Melhoria da Testabilidade de Sistemas Orientados a Objetos

Nesta seção são apresentados aspectos que influenciam a testabilidade de um software e algumas abordagens de DFT voltadas para sistemas orientados a objetos.

3.5.1 Principais Aspectos de Testabilidade de Software

Segundo o padrão IEEE 610-12/90, testabilidade pode ser definida como:

- facilidade que um componente ou sistema apresenta para o estabelecimento de critérios de teste e para a realização de testes que verifiquem se tais critérios foram satisfeitos.

- quão adequada é a definição dos requisitos de modo que se possa estabelecer critérios de teste e realizar testes para verificar se estes critérios foram satisfeitos.

Em [VM95] testabilidade de software é a probabilidade que, caso o software possua uma falha, uma parte do software apresentará um defeito durante os testes com uma determinada distribuição de entrada.

As capacidades de controle e observação são aspectos chave da testabilidade, pois, no teste de um componente é preciso verificar se dada uma entrada (controle da entrada), o componente produz a saída esperada (monitoração da saída). De modo informal, observabilidade é a facilidade de verificar se entradas específicas afetam as saídas; e controlabilidade é a facilidade de produzir uma determinada saída a partir de uma entrada especificada [Fre91]. Segundo R. Binder [Bin94a] alguns fatores que contribuem para a testabilidade de software são:

- Representação
  As especificações do sistema, seja em linguagem natural ou especificações formais, são muito importantes para a fase de 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 documentação atualizada de projeto, implementação e testes (especificação dos casos de teste e resultados esperados), isto contribui para a testabilidade de software.

- Implementação
  Características do código fonte podem influenciar seu nível de testabilidade. Por exemplo, um módulo que contenha funcionalidades da interface e da aplicação, dificulta o teste de somente uma delas. Então, módulos fracamente acoplados e fortemente coesos, com funcionalidades bem específicas, facilitam a realização de testes.

- Seqüência de testes
  A seqüência de testes é a coleção de casos de testes e a estratégia usada para aplicá-los. A existência de um oráculo, a capacidade de reutilização de casos de testes e uma boa documentação dos testes, são características que facilitarão a aplicação dos testes.

- Teste embutido
  Uma classe que possui facilidades para observar e controlar seu estado interno será mais fácil de testar. Adicionar a ela componentes tais como assertivas e/ou métodos especiais para relatar o seu estado interno ou colocá-la em um determinado estado, são meios para se melhorar a testabilidade da classe. Se esses componentes forem utilizados somente para teste, ou seja, se houver uma separação das funcionalidades de teste e da aplicação, então a testabilidade será ainda maior. Em C++, por exemplo, pode-se utilizar diretivas de compilação para acrescentar ou remover componentes de teste.

  Uma assertiva é um teste sobre todo ou parte do estado de um programa em execução, dado pelo valor de suas variáveis [TR94]. Assertivas representam uma valiosa ferramenta de projeto para sistemas orientados a objetos [TR94]. 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 [VM95]. Contudo, a inclusão de muitas assertivas no sistema deve ser evitada pois pode reduzir o desempenho e o risco de adicionar assertivas incorretas ou redundantes é maior.

- Ferramentas de teste
  A não utilização de ferramentas automatizadas (por exemplo, ferramentas de captura de dados de entrada, geradores de dados de entrada e geradores de teste) reduz a testabilidade pois dificulta a realização dos testes e a obtenção do nível de confiabilidade desejado.

- Processo de desenvolvimento de software
  A maturidade do processo no qual o software é desenvolvido pode influenciar em sua testabilidade. Um processo bem definido que integre o processo de teste e a existência de ferramentas automatizadas de apoio ao desenvolvimento de software são fatores importantes.

Um problema com técnicas de verificação baseadas em código é que elas são aplicadas ao final do ciclo de vida do software. Assim, se o software possui falhas devido a decisões de projeto por exemplo, o custo para corrigir essas falhas será alto. Este problema continuará a existir se a testabilidade do software for avaliada somente depois que o código estiver implementado. Porém, a testabilidade pode ser considerada antes da implementação [VM95].

O projeto para testabilidade (DFT, do inglês *Design for Testability*) é uma estratégia para organizar o processo de desenvolvimento de software de modo a maximizar a

testabilidade do software produzido [Bin94a]. Embora este conceito seja relativamente novo na área de teste de software, ele tem sido muito importante no projeto de circuitos VLSI.

Na próxima seção serão apresentadas abordagens para melhorar a testabilidade de software, mais especificamente a testabilidade do código fonte, com a aplicação de estratégias de projeto para testabilidade.

3.5.2 Abordagens de Projeto para Testabilidade de um Software Orientado a Objetos

O teste de circuitos é feito para detectar falhas de fabricação ou decorrentes do efeito de vibrações, calor, por exemplo. São aplicadas cadeias de bits de entrada e observadas as cadeias de bits de saída. Assim, no teste de hardware os aspectos de controle e observação também são importantes para redução dos custos com testes. Segundo [TR94] a observabilidade representa o principal obstáculo em teste de circuitos integrados. Uma forma de melhorar a observabilidade consiste em aumentar a quantidade de pinos a fim de que os pinos extras levem sinais internos que serão verificados durante os testes. Existem três abordagens principais para o projeto para testabilidade em hardware:

- Soluções ad hoc: são soluções específicas para um componente.
- Estruturada: envolve o estabelecimento de regras de projeto que facilitem os testes.
- Teste embutido BIT (do inglês, *Built-in Test*): envolve a utilização de circuitos e pinos padrões de teste, de modo que seja possível colocar o circuito ou placa em modo teste, transmitir as entradas de teste e capturar as saídas. Uma extensão desta abordagem é o BIST (do inglês, *Built-in Self Test*), que permite a geração automática de seqüências de teste.

Como dito na introdução deste capítulo, a testabilidade de um software tem se tornado uma medida importante da qualidade do software. Um software orientado a objetos apresenta obstáculos específicos, além daqueles existentes em software estruturado, para se alcançar uma alta testabilidade. Fazendo-se uma analogia entre circuitos integrados e objetos [Hof89,Bin94a], surgiu a proposta de se aplicar às classes as técnicas de DFT utilizadas no projeto de circuitos. A seguir será apresentado como seria a aplicação das abordagens de DFT na construção de classes.

- Abordagem ad hoc
  Para testar uma classe são construídos *driver* e *stubs* específicos, podem ainda ser adicionadas assertivas e/ou outras formas de instrumentação para melhorar as capacidades de controle e observação. Esta abordagem está ilustrada na figura 3.1. CST indica a classe sob teste e o círculo com um "x" ao centro indica o oráculo. Uma desvantagem desta abordagem é que ela pode exigir longas seqüências de preparação para colocar a classe em um determinado estado, bem como a utilização de ferramentas

de depuração para verificar o estado do objeto. Isto pode aumentar o custo dos testes ou reduzir a confiabilidade dos mesmos.

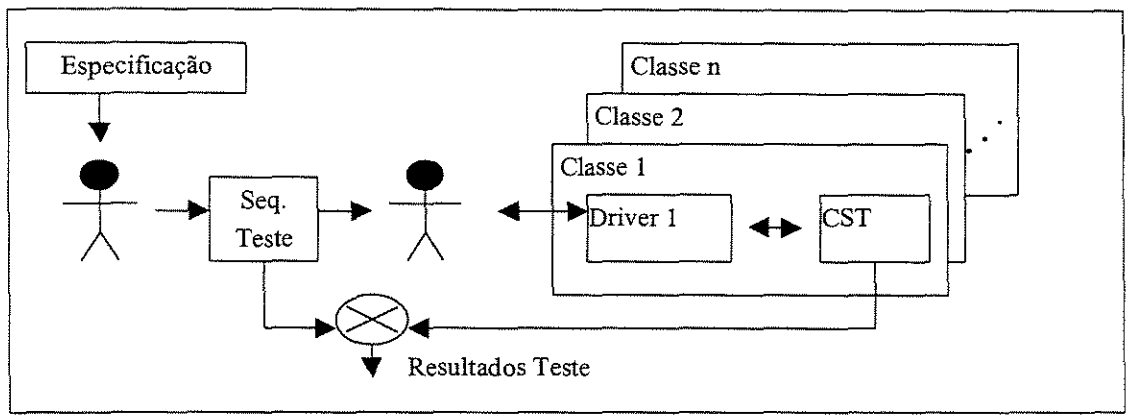


Figura 3.1: Abordagem de DFT ad hoc para classes.

- Abordagem estruturada
  São adicionadas à classe funções padrões de BIT de modo a criar capacidade de observação e controle, tais como funções para colocar a classe em um determinado estado e para relatar o estado interno da classe. Estas funções podem ainda possuir uma interface padrão a fim de facilitar a implementação de *drivers* e sua utilização pelos testadores. Contudo, ainda seria necessário a codificação de um *driver* para cada classe e a geração manual de casos de teste. Esta abordagem está ilustrada na figura 3.2.

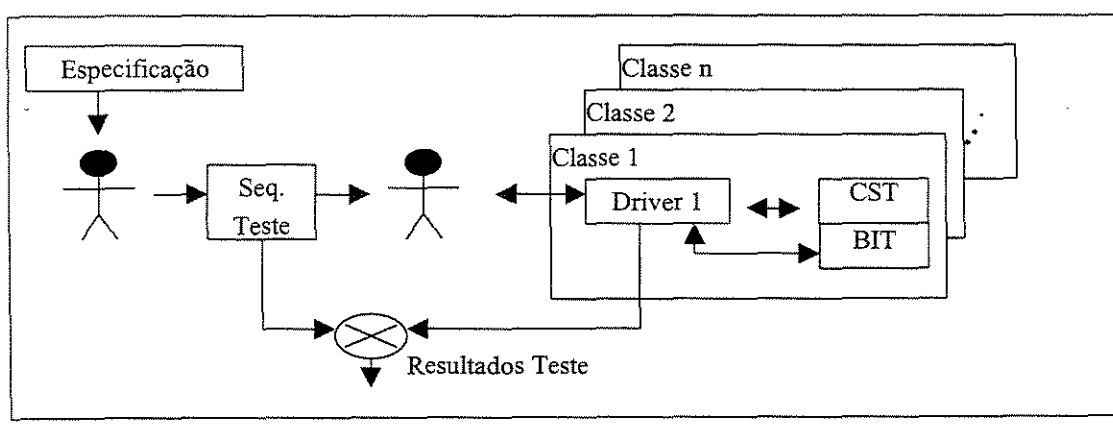


Figura 3.2: Abordagem de DFT estruturada para classes.

- Abordagem BIST
  A abordagem anterior é limitada quanto à automatização da geração, execução e avaliação de testes. Na abordagem BIST existe uma especificação de teste embutida à CST e um *driver* genérico capaz de gerar a seqüência de testes a partir desta especificação, executar esta seqüência sobre a CST e avaliar os resultados dos testes. Esta abordagem está ilustrada na figura 3.3.

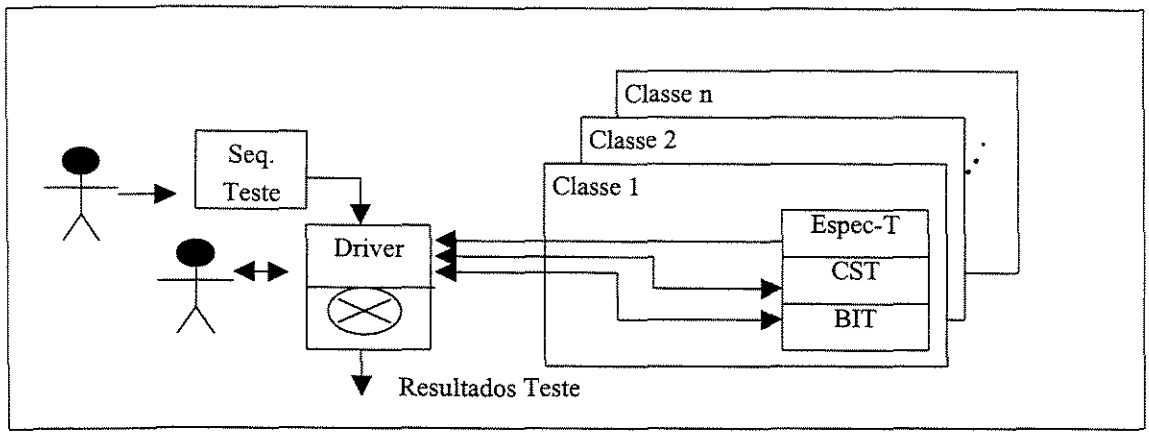


Figura 3.3: Abordagem de DFT BIST para classes.

Desta abordagem surgiu o conceito de classes autotestáveis. Uma classe autotestável é aquela que contém uma especificação de teste e métodos padrões para os testes, como apresentada na figura 3.4.

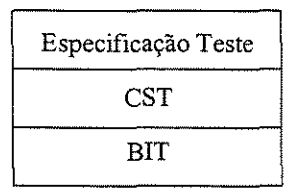


Figura 3.4: Classe autotestável.

A figura 3.4 representa uma abstração de uma classe autotestável. A fim de implementá-la, dois obstáculos devem ser superados:

- É necessário que o *driver* conheça a assinatura dos métodos da CST, além dos métodos de BIT. Isso geralmente é feito através de inspeção manual do código da classe, mas o ideal é que esta atividade possa ser automatizada. R. Binder [Bin94a] sugeriu algumas alternativas para essa associação, entre elas: a utilização de um *driver* genérico que lê a especificação de teste da classe e gera um *driver* específico que interage com ela; e, a geração de um programa pelo *driver* a partir da especificação de teste, sendo este programa aplicado sobre a classe por um interpretador.

- Deve ser definida uma linguagem adequada para construção da especificação de teste, de acordo com uma metodologia de testes escolhida, para que o *driver* possa lê-la e gerar a seqüência de testes a partir dela.

Além destes, o problema do oráculo também persiste e caso uma solução automatizada não seja encontrada, a análise dos resultados dos testes deverá ser feita manualmente.

3.6 Sumário

Na seção 3.1 foram apresentados os conceitos básicos de orientação a objetos. Em especial, a herança que é o principal meio de reutilização de software, pois através dela é possível compartilhar atributos e operações entre classes.

Na seção 3.2 foram descritas as principais dificuldades no teste de sistemas orientados a objetos. Foram apresentadas duas abordagens utilizadas para solucionar o problema relacionado a herança: classes aplanadas e a abordagem incremental hierárquica. A primeira considera a classe com todas suas características, herdadas ou não, no projeto e execução dos testes. A última considera a classe em sua hierarquia de herança com o objetivo de reutilizar a informação de teste da classe base no teste das subclasses.

O teste de software orientado a objetos segue abordagens diferentes do teste de software tradicional. Algumas delas foram apresentadas na seção 3.3. A abordagem FREE é utilizada para desenvolver seqüências de teste de unidades, de integração e de sistema. Ela utiliza a solução de classes aplanadas para testar classes derivadas. Na abordagem hierárquica são definidos componentes de software para vários níveis de teste, por exemplo objeto, classe ou um conjunto de classes. Cada componente é testado individualmente e depois integrado com outros componentes já testados, até que todo o sistema esteja integrado e possa ser testado como um único componente. A abordagem Ordem de Teste é uma estratégia para os testes de unidades e de integração. Seu objetivo é reduzir o esforço requerido para construção de *stubs* e *drivers*, além de promover a reutilização dos casos de testes já gerados.

Na seção 3.4 foram apresentados os *frameworks* FOOT e ClassBench, e o ambiente OOTM, desenvolvidos para testar sistemas orientados a objetos.

Na seção 3.5 foram apresentados o conceito de testabilidade e os aspectos de um software que influenciam em sua testabilidade, tais como sua representação, implementação, seqüência de teste e seu processo de desenvolvimento. Ainda nesta seção, foram apresentados o conceito de DFT e a aplicação das abordagens de DFT para circuitos integrados à construção de classes em sistemas orientados a objetos. Da aplicação da abordagem BIST, surge a idéia de classes autotestáveis utilizada neste trabalho, as quais contêm uma especificação de teste e métodos de teste embutidos para melhorar a observabilidade e a controlabilidade do estado interno de um objeto durante os testes.

O enfoque deste trabalho não foi buscar soluções para os problemas de teste de sistemas orientados a objetos apresentados neste capítulo, mas validar um novo modo de se testar classes utilizando a proposta de classes autotestáveis. Assim, foi implementada a abordagem de classe autotestável sendo desenvolvida uma metodologia para sua construção e utilização, assim como um protótipo para auxiliar o usuário. Ambos serão apresentados nos capítulos a seguir.

Capítulo 4

Metodologia Proposta para Construção e Uso de Classes Autotestáveis

A metodologia descrita neste capítulo tem por objetivo permitir a construção de classes autotestáveis, preparando-as para os testes através do acréscimo de mecanismos de autoteste, e facilitando sua utilização. Ela é composta dos seguintes passos:

1. Construção do modelo de teste para a classe sob teste
2. Construção de uma especificação de teste que descreve o modelo de teste construído e associação desta à classe sob teste
3. Instrumentação da classe
4. Criação de um repositório de teste
5. Geração da seqüência de teste
6. Geração do oráculo
7. Compilação da classe em modo teste e execução da seqüência de teste

A seguir cada um desses passos será detalhado. Para melhor ilustrá-los será usada como exemplo a classe `Produto` descrita na figura 4.1. Esta classe representa um produto em estoque de uma fábrica qualquer, que possui como atributos: nome, quantidade em estoque, preço unitário e fornecedor. `Fornecedor` representa uma classe relacionada, porém, como este trabalho aborda o teste individual de classes, será utilizada apenas a classe `Produto`. Outro exemplo da utilização desta metodologia pode ser encontrado em [TM98].

O capítulo está organizado do seguinte modo: na seção 4.1 são apresentadas algumas considerações feitas na definição da estratégia de teste, da seção 4.2 à seção 4.8 são apresentados os passos da metodologia, e na seção 4.9 é apresentado um sumário do capítulo.

```
class Produto {
    int qtde;
    char* nome;
    float preco_unit;
    Fornecedor* fornec;
  public:
    Produto();
    Produto(int q,char* n, float p, Fornecedor* f);
    Produto(char* n);
    ~Produto();
    void Altera_nome(char* n);
    void Altera_qtde(int q);
    void Altera_preco(float v);
    void Altera_fornecedor(Fornecedor* f);
    void Consulta_dados();
    void Armazena_produto();
    Produto* Remove_produto();
};
```

Figura 4.1: A classe `Produto`.

4.1 Considerações

Os seguintes aspectos foram considerados na definição da estratégia de teste abordada na metodologia:

- **Teste de classe**

A utilização de classes autotestáveis visa, neste trabalho, auxiliar na melhoria da testabilidade de classes fornecendo mecanismos de teste embutidos para melhorar os aspectos de controle e observação de uma classe.

- **Testes baseados na especificação**

Neste trabalho foi considerada a utilização de testes baseados na especificação. Segundo [HM92, Sie96] estes são mais adequados quando classes são reutilizadas, pois geralmente a funcionalidade dos métodos permanece a mesma enquanto o código na maioria das vezes é alterado. Além disso, não é possível gerar testes estruturais para classes abstratas por exemplo, mas pode-se gerar testes funcionais que serão utilizados para testar as classes derivadas concretas da classe. Foi escolhido o modelo de fluxo de transação, descrito no capítulo 2, como modelo de teste. Ele atende a uma das recomendações feitas em [Wey98], segundo a qual os testes de um componente reutilizável, realizados tanto por seus desenvolvedores quanto por seus usuários, devem procurar exercitar todos os usos possíveis desse componente e esse é o objetivo dos testes de fluxo de transação.

- **Reutilização de testes**

Com a utilização de classes autotestáveis este trabalho visa não só promover a reutilização da classe, mas também a de casos de teste. Desse modo, o usuário de uma classe poderia reutilizar os testes gerados pelo desenvolvedor da classe fazendo acréscimos ou alterações quando necessário. Considerando que a herança é o principal mecanismo para reutilização de classes em orientação a objetos, será utilizada a técnica incremental hierárquica para construir seqüências de testes, como proposto em [HM92]. Esta abordagem foi apresentada no capítulo 3.

4.2 Passo 1 - Construção do Modelo de Teste para a Classe sob Teste

Neste passo deve ser construído o modelo de fluxo de transação para a classe sob teste, a partir de agora chamada CsT. Ele é um modelo funcional utilizado para teste de sistema em [Bei90] e adaptado em [Sie96] para o teste de classes. Uma transação pode ser vista como uma unidade de trabalho e representa ciclos de vida de um objeto, desde sua criação até sua destruição. Este modelo mostra os diferentes modos de se criar um objeto, as diferentes tarefas ou processos que o objeto pode realizar e os diferentes modos de se destruir o objeto [Sie96].

O modelo de fluxo de transação é representado por um grafo, onde cada transação é um caminho através deste grafo e representa um objeto de teste. Os arcos representam seqüências de execução e os nós representam processos ou tarefas de interesse, por exemplo a execução de um método ou um conjunto de métodos. É considerada somente a interface pública da classe para a construção do grafo. Este grafo pode ser construído em uma série de iterações a partir da descrição de cenários de uso (*use-cases*) da classe da seguinte forma [Sie96]:

- Eliminar referências externas, como por exemplo referências a bases de dados, arquivos ou métodos de outras classes;
- Simplificar o modelo combinando métodos em um único processo (i.e., nó) quando possível;
- Identificar cenários de uso que não aparecem no modelo em construção pois isto pode indicar que alguns requisitos não foram satisfeitos;
- Identificar métodos públicos que não aparecem no modelo construído pois isto pode indicar que algum requisito não foi satisfeito ou que estes métodos não deveriam pertencer a interface pública da classe.

Por exemplo, considere os seguintes cenários de uso envolvendo a classe `Produto`, descrita na figura 4.1:

1. O usuário cria um novo produto fornecendo nome, preço unitário e quantidade inicial. Ele também associa o produto com seu fornecedor e o armazena no estoque.

2. O usuário consulta os dados cadastrados para um produto e atualiza as informações necessárias. Depois, armazena o produto novamente no estoque.

3. O usuário pode remover um produto do estoque.

Estes cenários de uso podem ser representados de forma mais detalhada pelas seguintes seqüências de ações:

- Seqüência para o cenário de uso 1

1.1. Instanciar um objeto Produto.
1.2. Associar um objeto Fornecedor ao produto.
1.3. Criar uma string contendo o nome do produto.
1.4. Fornecer o seu preço unitário.
1.5. Fornecer a quantidade inicial.
1.6. Armazenar o produto no estoque.
1.7. Atualizar dados do produto.
1.8. Destruir o objeto quando o remove do estoque ou fechar a aplicação/janela.

- Seqüência para o cenário de uso 2

2.1. Instanciar um objeto Produto.
2.2. Recuperar os dados do produto em estoque.
2.3. Consultar dados do produto.
2.4. Atualizar dados do produto.
2.5. Destruir o objeto quando o remove do estoque ou fechar a aplicação/janela.

- Seqüência para o cenário de uso 3

3.1. Instanciar um objeto Produto.
3.2. Recuperar os dados do produto em estoque.
3.3. Remover o produto do estoque.
3.4. Destruir o objeto.

A partir destas seqüências pode ser construído um diagrama de fluxo de dados para a classe `Produto`, representado na figura 4.2.

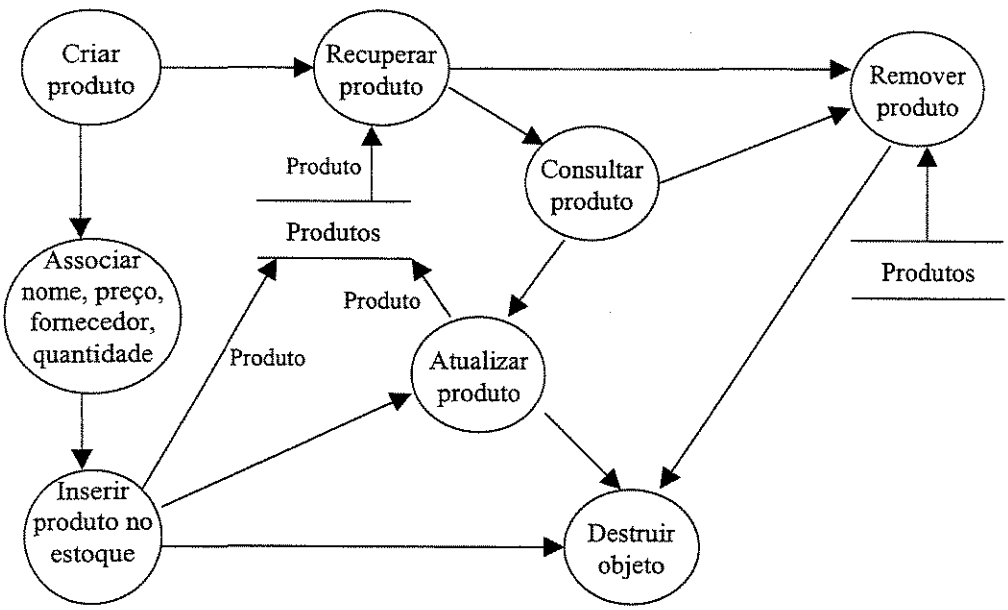


Figura 4.2: Diagrama de fluxo de dados para a classe `Produto`.

Com este diagrama podem ser aplicados os passos citados anteriormente para construir uma versão inicial do grafo de fluxo de transação. A figura 4.3 apresenta esta primeira versão, na qual foram removidas as referências à base de dados e os nomes de alguns processos foram convertidos aos métodos correspondentes.

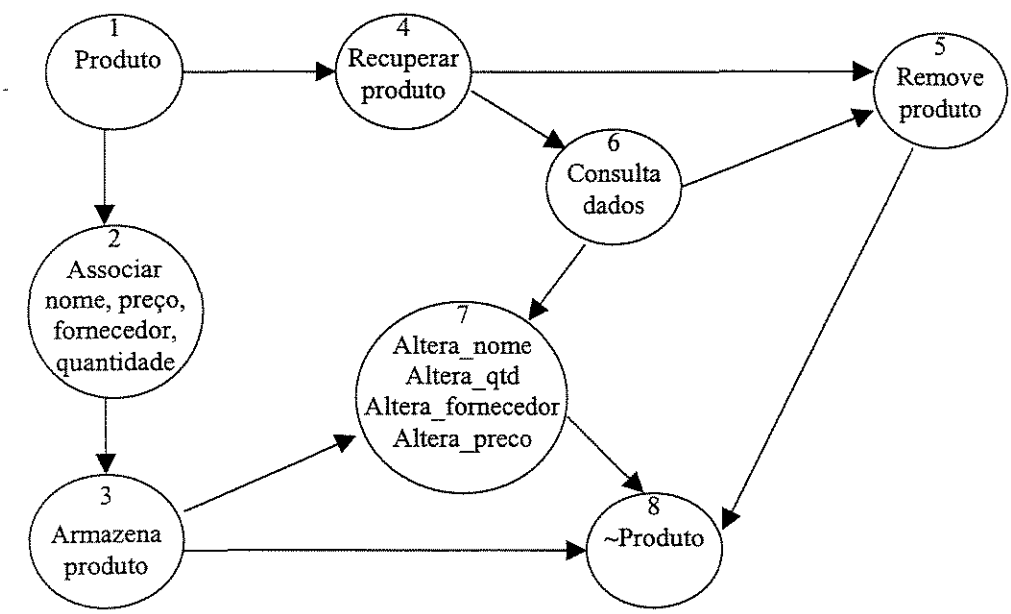


Figura 4.3: Versão inicial para o grafo de fluxo de transação da classe `Produto`.

Neste grafo verifica-se que os processos 1 e 2 podem ser representados pelo construtor da classe com parâmetros; e os processos 1 e 4 podem também ser representados pelo método construtor que recupera os dados de um determinado produto em estoque. Assim, são representados dois nós iniciais adicionais, um para cada construtor.

Note que o processo 5 poderia ser executado após os processos 3 ou 7 também, pois a execução destes indica que o produto já foi armazenado no estoque. Assim, geramos o modelo final, ilustrado na figura 4.4.

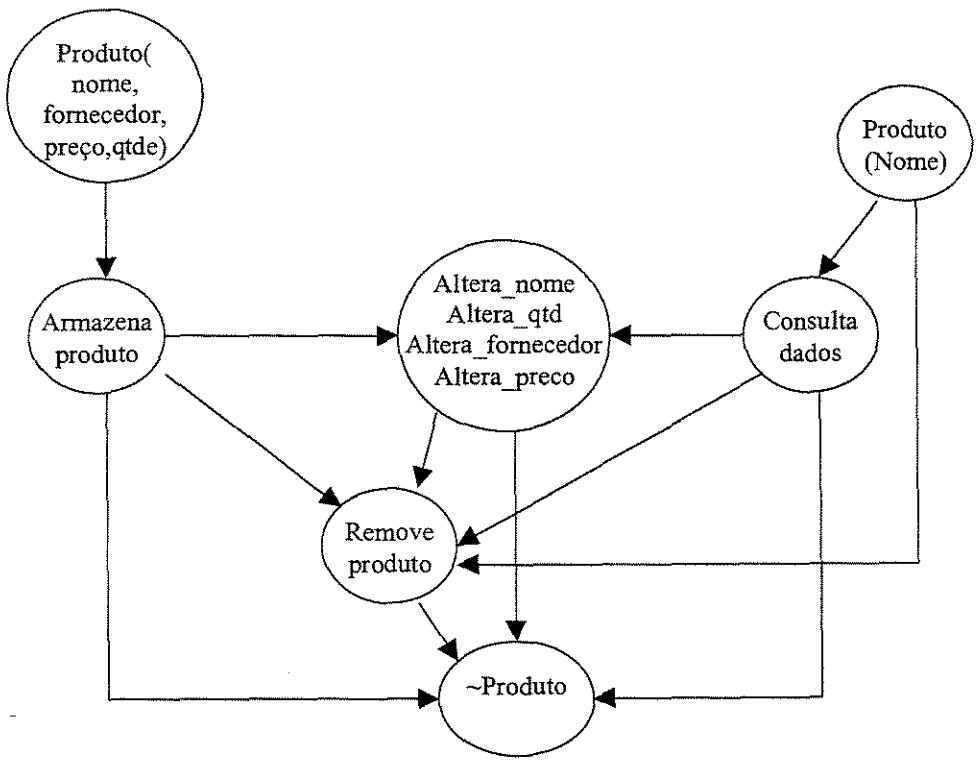


Figura 4.4: Modelo de fluxo de transação da classe `Produto`.

Algumas transações derivadas deste modelo são:

- Produto(nome,fornecedor,preço,qtde), Armazena_produto, ~Produto
- Produto(nome,fornecedor,preço,qtde), Armazena_produto, Remove_produto, ~Produto
- Produto(nome), Consulta_dados, Remove_produto, ~Produto
- Produto(nome), Consulta_dados, Altera_nome, ~Produto

4.3 Passo 2 - Construção da Especificação de Teste

A especificação de teste deve descrever o modelo de teste. 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. Por exemplo, apesar dos

modelos de transição de estados e de fluxo de transação serem representados através de grafos, cada nó e arco têm significados e informações diferentes em cada um deles.

O formato da especificação de teste desenvolvida neste trabalho para o modelo de fluxo de transação e as informações que ela contém são mostrados na figura 4.5.

```
Classe(     Nome,
            Abstrata/concreta,          - indica se a classe é abstrata ou não
            Template/normal,            - indica se a classe é template ou não
            Nome_superclasse,
            Lista_arq_include )         - lista de arquivos a serem incluídos na compilação

Parametros_template( Lista_parâmetros ) - lista de parâmetros necessários para instanciar uma classe
                                          template

Atributo(   Nome,
            Tipo,                       - pode ser intervalo, string, conjunto, objeto ou ponteiro
            Dado1,                      - para o tipo intervalo, indica o valor mínimo
                                          string e conjunto, indica uma lista de valores permitidos
                                          objeto e ponteiro, indica a classe envolvida
            Dado2 )                     - para o tipo intervalo, indica o valor máximo
                                          para os demais não é utilizado

Método(     Identificador,              - identificador único para o método
            Nome,
            Tipo_de_retorno,
            Classificação,              - classificação conforme tabela 1 (seção 3.2)
            Num_de_parâmetros )

Parametro(  Nome,
            Método_a_que_pertence,      - identificador do método
            Tipo,                       - similar ao de atributo
            Dado1,
            Dado2 )

No(         Identificador,
            Nó_inicial,                 - indica se é um nó inicial ou não
            Arcos_saída,                - número de arcos de saída
            Lista_de_métodos )          - lista de identificadores de métodos que compõem o nó

Arco(       Nó_origem,                  - identificador do nó
            Nó_destino )                - identificador do nó
```

Figura 4.5: Formato da especificação de teste.

Além das informações sobre cada arco e nó do grafo, são fornecidas informações sobre a classe, seus atributos e métodos, para que se possa derivar mensagens aceitas pelos objetos da CsT.

As informações sobre a classe auxiliarão na geração automática de casos e seqüências de testes, indicando as bibliotecas necessárias para compilar a classe e como manipulá-la. Por exemplo, se a classe é abstrata não é possível instanciar um objeto a partir dela; se é uma classe *template*, parâmetros adequados devem ser utilizados para instanciar um objeto desta classe; se é uma classe derivada, então a informação de teste da classe base poderá ser reutilizada. No caso de classes *templates*, os parâmetros necessários para

instanciar um objeto da classe podem assumir inúmeras possibilidades de valores. Como citado no capítulo 3, ainda não se sabe ao certo como testar este tipo de classe. A solução dada neste trabalho foi incluir uma cláusula na especificação de testes (Parametros_template) onde o usuário indique os valores dos parâmetros para os quais ele deseja que a classe seja testada.

As informações sobre os parâmetros e atributos são basicamente as mesmas e são importantes para a geração de dados de teste. Foram definidos cinco tipos de dados:

- intervalo_int: representa o tipo inteiro e define um intervalo de valores;
- string: representa o tipo string e define um conjunto de cadeias de caracteres válidas para o parâmetro/atributo;
- conjunto: representa os tipos inteiro ou *float* e define um conjunto de valores válidos;
- objeto: representa qualquer objeto e define a classe da qual este objeto é derivado;
- ponteiro: representa um ponteiro para qualquer tipo/classe e define o tipo/a classe referenciado pelo ponteiro.

A classificação dos métodos permite verificar a possibilidade de reutilização de casos de teste. O número de parâmetros auxilia na geração de chamadas para os métodos.

Esta especificação de teste foi desenvolvida para representar o modelo de fluxo de transação, dado por um grafo. Desse modo, os arcos representam apenas transições válidas de um nó para o outro; e cada nó representa a execução de um ou mais métodos (descritos por Lista_de_métodos da cláusula no). Informações adicionais sobre cada nó (no_inicial e num_arcos_saida) auxiliarão a fazer o percurso do grafo para derivar transações.

Uma descrição da gramática que representa a especificação está descrita no Apêndice A.

A especificação construída para a classe `Produto`, a partir do modelo de fluxo de transação mostrado na figura 4.4, está ilustrada figura 4.6.

Após a elaboração da especificação de teste, esta deve ser associada a classe sob teste, de forma que toda vez que a classe seja reutilizada sua especificação esteja disponível.

4.4 Passo 3 - Instrumentação da Classe sob Teste

A CsT e seus métodos devem ser acrescidos de mecanismos de teste embutido (BIT, do inglês *Built-in Test*): métodos relatores e predicados (pré e pós-condições e invariantes de classe).

Os métodos BIT devem ter visibilidade sobre os membros privados da classe. De acordo com a abordagem de classes autotestáveis, esses métodos devem ter uma interface padrão para facilitar a geração da seqüência de teste. A visibilidade dos membros privados é necessária para facilitar a análise dos resultados dos casos de teste aplicados.

```
classe('Produto', n, n, _ , [ ]).                    -- nome, ind. abstrata, ind. template, nome super., lista arq

parametros_template( [ ] ).                          -- lista de parâmetros

atributo('qtd', intervalo_int, 10000, 1).            -- nome, tipo, dado1, dado2
atributo('nome', string, ['farinha','detergente','sabao'], _ ).
atributo('preco_unit', conjunto, [2.23, 3.3, 4.5, 11.1],_ ).
atributo('fornec', ponteiro, 'Fornecedor',_).

metodo(m1, 'Produto', _ , construtor,0).             -- id. met., nome, tipo ret., classificação, num. param.
metodo(m2, 'Produto', _ , construtor,4).
metodo(m3, 'Produto', _ , construtor,1).
metodo(m4, '~Produto', _ , destrutor,0).
metodo(m5, 'Altera_nome', 'void', novo,1).
. . .
metodo(m10,'Armazena_produto','void',novo,0).
metodo(m11,'Remove_produto','void',novo,0).

parametro('q',m2, intervalo_int, 10000, 1).          -- nome, id. met., tipo, dado1, dado2
parametro('n', m2, string, ['farinha','detergente','sabao'], _ ).
parametro('p', m2, conjunto, [2.23, 3.3, 4.5, 11.1],_ ).
parametro ('f', m2, ponteiro,'Fornecedor',_).
parametro ('n', m3, string, ['farinha','detergente','sabao'], _ ).
parametro ('n', m5, string, ['farinha','detergente','sabao'], _ ).
parametro ('q', m6, intervalo_int, 10000,1).
parametro ('v', m7, conjunto, [2.23, 3.3, 4.5, 11.1],_ ).
parametro ('f', m8, ponteiro, 'Fornecedor',_).

no(n1,sim,1,[m2]).                                   -- id. no, ind. no inicial, num. arcos saída, lista met.
no(n2,sim,1,[m3]).
no(n3,nao,0,[m4]).
no(n4,nao,3,[m10]).
no(n5,nao,2,[m5,m6,m7,m8]).
no(n6,nao,3,[m9]).
no(n7,nao,1,[m11]).

arco(n1,n4).                                         -- id. no inicial, id. no final
arco(n2,n6).
arco(n2,n7).
. . .
arco(n5,n7).
arco(n5,n3).
arco(n7,n3).
```

Figura 4.6: Especificação de teste para a classe `Produto`.

Os predicados descrevem as cláusulas ou condições do contrato entre a classe e seus clientes [Mey00]. Este contrato estabelece as relações entre a classe e seus clientes, ou seja, o que ela deve fornecer para eles e o que ela deve receber deles. São utilizados três tipos de predicados:

- Invariante de classe: são restrições de consistência que caracterizam a semântica da classe [Mey00]. A invariante de classe deve ser verdadeira para qualquer método da classe, qualquer instância da classe e também para suas classes base e derivadas. Geralmente são implementadas como métodos que validam o estado da classe.

- Pré-condição: Aplica-se a métodos individuais e significa uma condição que deve ser verdadeira antes da execução do método.

- Pós-condição: Também se aplica a métodos individuais e significa uma condição que deve ser verdadeira quando o método termina sua execução. Geralmente é criada sobre os dados de saída do método.

Na construção de pré e pós-condições deve-se analisar todos os dados que o método tem disponível e utiliza, tais como variáveis globais, parâmetros, atributos e variáveis locais. Geralmente, elas são implementadas como exceções ou comandos como a macro *assert* de C++ que abortam o programa se a condição testada é falsa.

Para a classe `Produto` podem ser consideradas as seguintes condições como invariante de classe:

- Se fornecedor não for um objeto válido, então o objeto Produto é inválido.
- Se a quantidade não for maior ou igual a 0, então o objeto Produto é inválido.
- Se o preço unitário não for maior ou igual 0, então o objeto Produto é inválido.

Se o tamanho do campo "nome do produto" na base de dados tiver tamanho 100, por exemplo, poderia ser acrescentada a seguinte condição:

- Se o nome do produto tiver tamanho maior que 100, então o objeto Produto é inválido.

Considere o método `Altera_qtd` da classe `Produto`:

```
void Produto::Altera_qtd(int q)
{
   qtde = q;
}
```

Ele é bastante simples, possui como entrada o parâmetro "q" e altera a quantidade do produto com o seu valor. Como pré-condição pode ser considerada a seguinte condição:

- Se o valor do parâmetro "q" maior que 0, então o método pode ser executado.

O método não possui nenhum atributo ou variável de saída, assim pode ser considerada como pós-condição do método a seguinte condição:

- Se o valor do atributo "qtde" for diferente do valor do parâmetro "q", então o método é inválido.

As condições identificadas devem ser avaliadas, pois dependendo do contexto de uso da classe condições triviais podem não representar condições de teste relevantes [Sie96].

4.5 Passo 4 - Criação de um Repositório de Teste

É necessária a criação de um repositório onde fiquem armazenados todos os objetos relacionados aos testes da classe. Por exemplo: o código da classe, sua especificação de teste, as seqüências e os casos de teste gerados, os resultados obtidos e *stubs*.

A utilização efetiva deste repositório aumentaria a produtividade dos testes pois facilitaria o armazenamento e recuperação das informações.

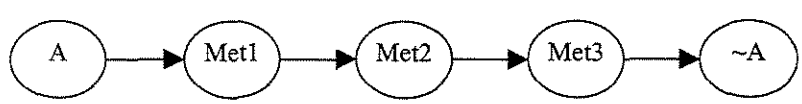
Este repositório, segundo [Sie96], não precisa ser necessariamente uma base de dados relacional ou orientada a objetos. Ele pode ser também um sistema de arquivos estruturado ou algum outro tipo de armazenamento organizado.

4.6 Passo 5 - Geração Seqüência de Testes

A seqüência de testes funciona como um *driver* para a classe (o conceito de *driver* foi descrito na seção 2.3) e cada caso de teste deve testar uma transação do modelo de teste.

Durante a geração dos casos de teste deve ser considerado quando a reutilização é possível. Para isso é utilizada a técnica incremental hierárquica (seção 3.3) baseando-se nas informações descritas na especificação de teste.

A abordagem utilizada para aplicar a técnica incremental hierárquica ao modelo de fluxo de transação é considerar a classificação da transação como um todo (exceto métodos construtores e destrutores). Assim são consideradas as classificações de cada método componente da transação para classificá-la. Por exemplo, considere o objeto de teste:



Caso Met1, Met2 e Met3 sejam recursivos, então não será necessário testar esse objeto de teste novamente. Caso pelo menos um deles seja redefinido e os demais recursivos, então o caso de teste gerado para esse objeto na classe base poderá ser reutilizado. Caso algum deles seja novo, então um novo caso de teste deverá ser gerado.

4.7 Passo 6 - Geração de um Oráculo

A necessidade de um oráculo é imprescindível para que os testes possam ser avaliados. Porém, nem sempre é possível se obter um oráculo pois às vezes não existe uma referência confiável para determinar as saídas corretas para um determinado caso de teste [Mar98]. Isto é chamado o problema do oráculo. O oráculo aqui seria a saída esperada para cada seqüência de métodos, ou uma exceção, caso o teste viole uma das assertivas.

4.8 Passo 7 - Compilação da Classe em Modo Teste e Execução da Seqüência de Teste

Os mecanismos BIT violam o encapsulamento da classe, assim, por motivos de segurança devem existir na classe somente durante os testes. A compilação condicional pode tornar isto possível, além de não alterar o comportamento da classe em modo de produção.

Após a classe ser compilada em modo teste e os mecanismos BIT serem habilitados, a seqüência de testes pode ser aplicada à classe.

4.9 Sumário

Neste capítulo apresentamos a metodologia desenvolvida para construção e uso de uma classe autotestável. Foram definidos: o modelo de teste, baseado no grafo de fluxo de transação, bem como um modo de adicionar mecanismos de teste embutido à classe, aumentando sua observabilidade e controlabilidade.

No próximo capítulo é descrita uma ferramenta que auxilia o usuário na criação e uso de classes autotestáveis.

Capítulo 5

ConCAT: Ferramenta de apoio a Construção e utilização de Classes Autotestáveis

No capítulo anterior foi descrita a metodologia proposta para construção e uso de uma classe autotestável. Porém, um auxílio automatizado para geração e execução dos testes é fundamental para melhoria da testabilidade pois mais testes podem ser gerados e aplicados. Com esse objetivo foi desenvolvido um protótipo de uma ferramenta de apoio à construção e utilização de classes autotestáveis chamada ConCAT [MT99] que automatiza parte desta metodologia. Neste capítulo é apresentado este prototótipo, seus componentes, características e limitações, bem como algumas considerações ao uso.

O capítulo está organizado do seguinte modo: na seção 5.1 é apresentada uma descrição geral da ferramenta; na seção 5.2 são apresentados os principais componentes de ConCAT; na seção 5.3 são apresentados os passos para sua utilização: a construção da especificação de teste, a instrumentação da classe e sua preparação para os testes, a geração do *driver* específico e sua execução; na seção 5.4 são apresentadas algumas considerações ao uso de ConCAT em certos tipos de classe; e, finalmente, na seção 5.5 é apresentado um sumário do capítulo com as vantagens e limitações da ferramenta.

5.1 Descrição Geral

Um modelo de objetos da fase de análise da ferramenta está descrito na figura 5.1. A seguir é descrita cada classe representada no modelo.

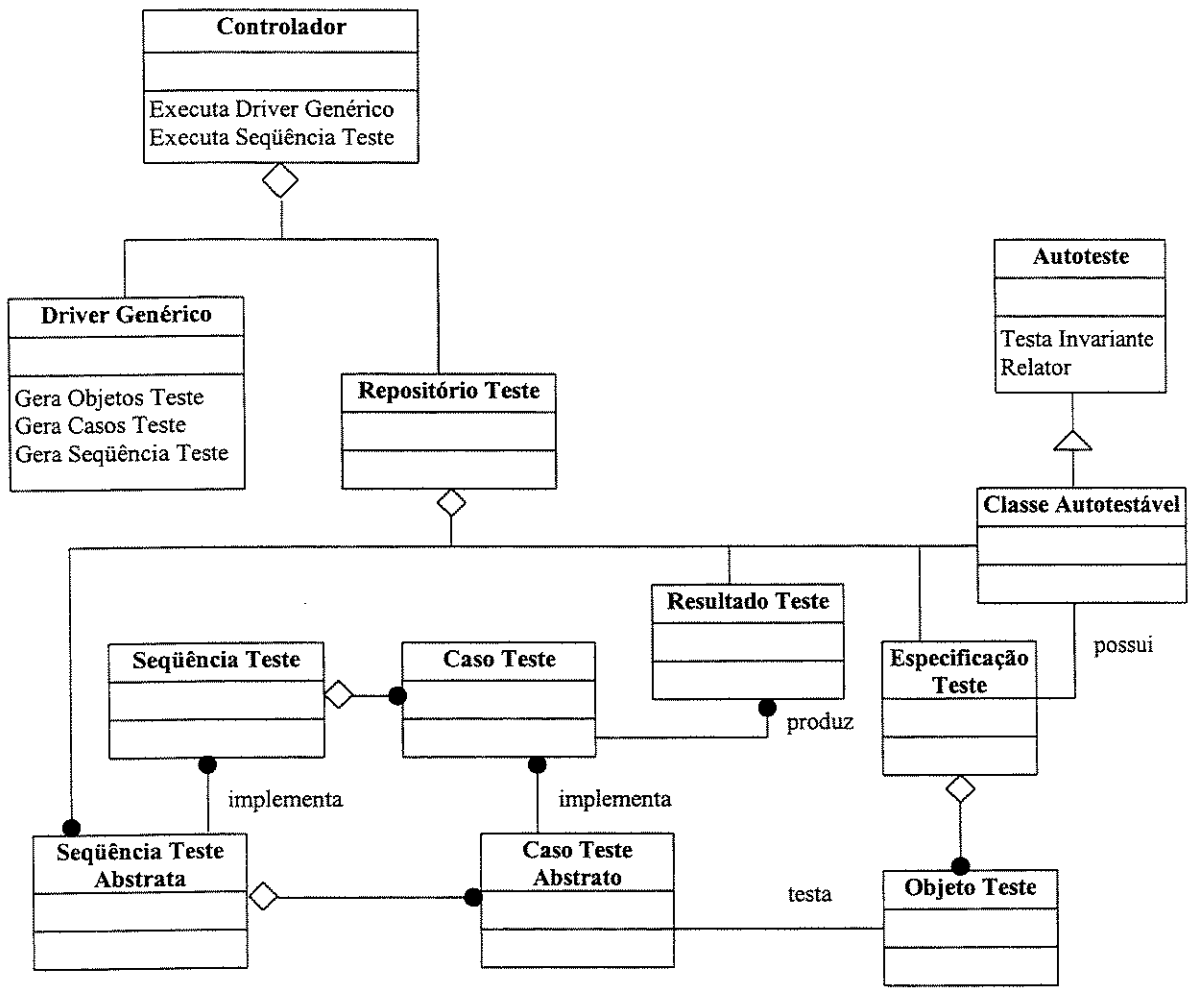


Figura 5.1: Modelo de objetos da ferramenta ConCAT.

- Controlador
  O Controlador representa a ferramenta. Ela possui um *driver* genérico para gerar os testes e um repositório onde ficarão armazenadas as informações de teste. Ele deve fornecer uma interface para o usuário fazer acesso ao *Driver* Genérico, executar uma seqüência de teste ou associar a especificação de teste à classe sob teste. Ele foi implementado em C++ e foi dividido em várias classes. Ele será mais detalhado na seção 5.2.2.

- *Driver* Genérico
  O *Driver* Genérico é o componente responsável pela geração dos objetos, casos e seqüência de teste. Ele foi implementado como um programa em Prolog e será detalhado na seção 5.2.1.

- Repositório de Teste
  O repositório de teste é o lugar onde estarão armazenadas todas as informações necessárias para os testes: objetos, casos, seqüências, especificações, resultados, a classe sob teste e outras informações não representadas no modelo como *stubs* por exemplo. O repositório de teste foi implementado em ConCAT como um sistema de arquivos, onde todos os arquivos relacionados a uma classe ficam no mesmo diretório.

- Classe Autotestável
  Esta classe representa a classe sob teste já instrumentada com os mecanismos de teste embutido. Uma classe autotestável deve ter uma especificação de teste associada e ser derivada da classe Autoteste que define a assinatura padrão dos métodos BIT.

- Autoteste
  A classe Autoteste, como citado acima, define os métodos BIT e a assinatura padrão deles. A classe Autoteste e seu relacionamento com a Classe Autotestável serão detalhados na seção 5.3.2.

- Especificação de Teste
  A especificação de teste representa o modelo de fluxo de transação da classe autotestável. Uma especificação de teste contém vários objetos de teste. Em ConCAT ela foi implementada como um arquivo e deverá ser gerada pelo usuário.

- Objeto de Teste
  Um objeto de teste representa um caminho no grafo do modelo de fluxo de transação da classe representado pela Especificação de Teste. Cada objeto de teste é testado por um Caso de Teste Abstrato. Os objetos de teste são gerados pelo *Driver* Genérico no formato de um arquivo. Eles serão mais detalhados na seção 5.2.1 que fala sobre o *Driver* Genérico.

- Caso de Teste Abstrato
  Um caso de teste abstrato representa o caso de teste gerado pelo *Driver* Genérico para testar um determinado objeto de teste. Ele pode não estar pronto para ser executado sendo necessário que o usuário o parametrize, isto é, complemente com dados que não foram gerados, por exemplo parâmetros que são ponteiros para outros objetos.

- Seqüência de Teste Abstrata
  Uma seqüência de teste abstrata representa a seqüência gerada pelo *Driver* Genérico. Ela contém vários casos de teste abstratos e também precisa de informações suplementares para que possa ser executada, como por exemplo, a ativação de outros objetos. Estas informações devem ser fornecidas pelo usuário. Uma seqüência de teste abstrata é implementada por uma ou mais seqüências de teste, dependendo do valor atribuído às informações que faltam.

- Caso de Teste
  Esta classe representa o caso de teste abstrato parametrizado pelo usuário e pronto para ser executado. Quando um caso de teste é executado ele produz um resultado observado que será utilizado para avaliar se o caso de teste detectou uma falha ou não.

- Seqüência de Teste
  A seqüência de teste representa uma seqüência de teste abstrata completada com as informações que permitam sua execução. Ela contém um ou mais casos de teste.

- Resultado de Teste
  Ao ser executado um caso de teste produz um resultado observado. Este valor deve ser armazenado no repositório de teste.

Na próxima seção serão detalhados dois principais componentes da ferramenta: o Controlador e o *Driver* Genérico.

5.2 Componentes de ConCAT

O *Driver* Genérico e o Controlador são os dois componentes principais de ConCAT. Na figura 5.2 está ilustrado o fluxo de dados entre esses componentes.

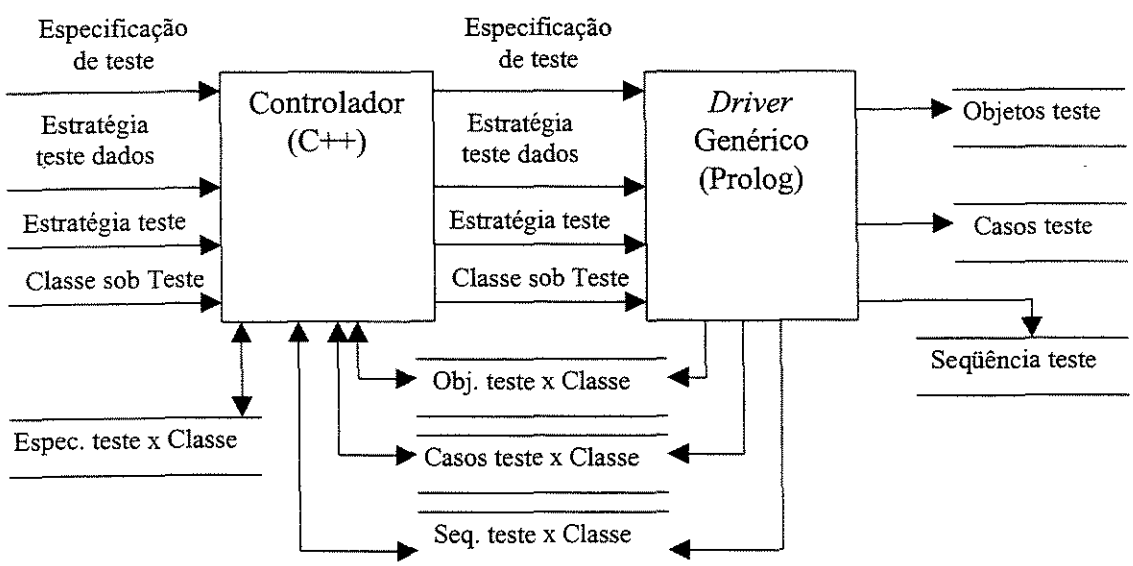


Figura 5.2: Fluxo de dados entre o Controlador e o *Driver* Genérico.

O *Driver* Genérico é responsável pela construção do *Driver* Específico (seqüência de teste abstrata) e geração dos objetos e casos de teste. O Controlador fornece a interface com o

usuário e é responsável pela gerência dos relacionamentos entre classe e especificação de teste, seqüências, objetos e casos de teste. Além disso, também é responsável pelo acionamento do *driver* genérico com os parâmetros corretos. O resto da seção apresentará detalhes sobre cada componente.

5.2.1 Driver Genérico

Um dos obstáculos para a implementação de classes autotestáveis foi como um único *driver* reconheceria a interface de todas as classes a serem testadas. Algumas soluções para este problema foram propostas em [Bin94a]. Neste trabalho utilizamos a solução baseada em um *driver* genérico, ilustrada na figura 5.3.

O *Driver* Genérico é responsável por ler a especificação de teste embutida na CsT e gerar um *driver* específico a partir dela, o qual será o responsável por aplicar os testes e armazenar os resultados. O *Driver* Genérico foi implementado como um programa em Prolog que tem como entradas: o nome do arquivo contendo a CsT, a especificação de teste, a estratégia para teste de dados e a estratégia para teste de fluxo de transação; e tem como saídas: objetos de teste, casos de teste e a seqüência de teste para a CsT que representa o *driver* específico, como ilustrado na figura 5.4.

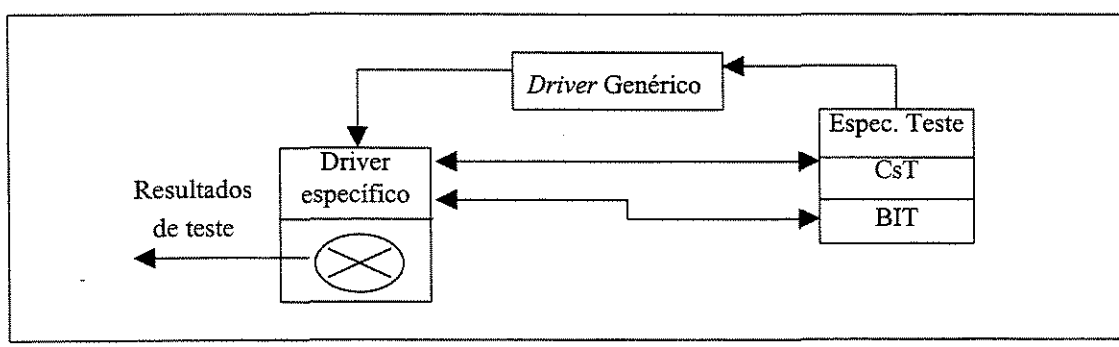


Figura 5.3: Solução baseada em *driver* genérico.

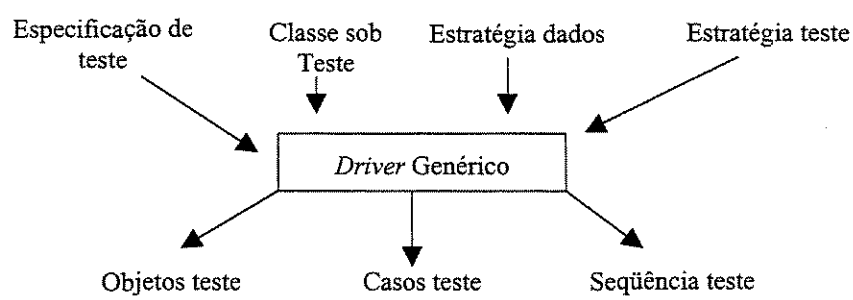


Figura 5.4: O *Driver* Genérico.

A especificação de teste será utilizada como base de conhecimento pelo *Driver* Genérico para gerar os objetos, casos e seqüência de teste para a classe. A estratégia de dados é utilizada na geração de mensagens aceitas pela classe, enquanto que a estratégia de teste é utilizada para orientar a geração dos objetos de teste. Estas estratégias serão mais detalhadas nos itens a seguir. Os objetos, casos e seqüência de teste são gerados em arquivos separados no diretório específico para a classe. O formato de cada arquivo será descrito no restante do capítulo.

No item a seguir é apresentada a utilização da linguagem Prolog na área de testes a fim de justificar sua escolha para implementar o *Driver* Genérico.

❖ Utilização de Prolog em Testes

Prolog é uma linguagem de programação lógica baseada em fatos e regras, os quais definem declarações sobre objetos ou relacionamentos entre eles. A sintaxe dos fatos e regras em Prolog é na forma de *cláusulas de Horn*. Neste trabalho foram utilizados fatos para definir a especificação de teste e regras para gerar casos de teste. Maiores detalhes da linguagem podem ser encontrados no Apêndice D.

Prolog tem sido utilizada na área de testes para avaliação de resultados de casos de teste, mas principalmente na construção de especificações de teste e implementação de geradores de teste.

Em [PSS+85] Prolog foi utilizada na implementação de um protótipo de ambiente de teste baseado em conhecimento, para validar chamadas ao *kernel* por sistemas operacionais Unix. As especificações de teste foram formalizadas como fatos e regras, e interpretadas por um gerador também em Prolog. Em [HS91] sua utilização foi na escrita de *scripts* de teste para módulos escritos em C e na construção de Protest, um sistema de apoio ao desenvolvimento e aplicação de testes para módulos em C, que aborda o problema de geração de entradas para casos de teste e o problema do oráculo. O problema do oráculo é tratado de dois modos: o testador fornece o resultado esperado para cada caso de teste, ou a saída do caso de teste é validada por um programa, também em Prolog, capaz de gerar o resultado esperado a partir de uma entrada. Em [UP83] Prolog foi utilizada no teste de protocolos de comunicação para geração de seqüências de teste a partir da especificação destes em gramáticas livres de contexto com atributos.

Algumas vantagens de se utilizar esta linguagem em testes são que ela:

- possibilita a definição de especificações de teste mais formais que uma descrição narrativa e que podem ser interpretadas diretamente sem necessitar de um *parser* ou tradutor mais elaborado;

- possui um nível de abstração mais alto que linguagens imperativas, como C por exemplo, sendo mais adequada para o desenvolvimento rápido de protótipos, além de facilitar a manutenção;

- muitas notações de especificação formal como tipos abstratos de dados e especificações algébricas, grafos de fluxo de dados, gramáticas formais e máquinas de estado, podem ser implementadas facilmente como programas lógicos;

- possibilita a definição de especificações executáveis;

- permite a definição de *templates* de casos de teste;

- facilita o acréscimo de novas estratégias de teste.

A pessoa responsável pela definição da especificação de teste não precisa necessariamente conhecer detalhes sobre o processamento de regras ou a ordem correta dos termos em uma regra. Isto pode implicar em alguns problemas, abordados em [Den91], os quais incluem a recursão, ordenação incorreta de termos em uma regra e a avaliação de predicados insuficientemente instanciados. A recursão não controlada pode levar a geração de infinitos casos de teste. A execução de uma regra instanciada com termos em uma ordem incorreta também pode levar ao problema da recursão, *overflow* ou, o que é pior, pode gerar um resultado errado. A avaliação de predicados com variáveis insuficientemente instanciadas ou instanciadas com valores incorretos pode produzir os mesmos efeitos da ordenação incorreta de termos.

Para evitar estes problemas, foi adotada a seguinte estratégia:

- a especificação, definida pelo usuário, é composta somente de fatos descritos conforme a sintaxe estabelecida pela gramática apresentada no Apêndice A, que fornece a ordem correta dos termos nos fatos para a pessoa que irá construí-la;

- o *Driver* Genérico, parte integrante da ConCAT, é que contém as regras necessárias para implementação dos métodos de teste descritos, usando a especificação de teste.

Assim, o usuário de ConCAT não precisa conhecer a linguagem Prolog para utilizá-la, pois terá somente que definir a especificação que descreve a classe e o modelo de teste, de acordo com a sintaxe descrita no Apêndice A.

❖ Estratégia de Geração dos Dados de Teste

Na geração dos casos de teste de acordo com o grafo de fluxo de transação, é necessário selecionar os valores dos parâmetros dos métodos. Uma estratégia de teste de dados (classes de equivalência, valor-limite ou outra) deve então ser usada. Na implementação atual a estratégia consiste em gerar dados válidos aleatoriamente, de acordo com a descrição dos valores possíveis como consta na especificação de teste, de acordo com o tipo do parâmetro. Por exemplo, considere a especificação dos parâmetros abaixo pertencentes

ao método identificado por "m1":

```
parametro('P1',m1, intervalo_int, 1000, 1).          //para o tipo intervalo_int são definidos
                                                       um valor máximo e um mínimo
parametro('P2', m1, string, ['farinha','detergente','sabao'], _ ). //para o tipo string é definida uma lista
                                                       de valores possíveis.
parametro('P3', m1, conjunto, [2.23, 3.3, 4.5, 11.1],_ ).  //para o tipo conjunto é definida uma lista
                                                       de valores possíveis
parametro ('P4', m1, ponteiro,'ClasseB',_ ).         //para o tipo ponteiro é indicado o tipo
                                                       apontado
parametro ('P5', m1, objeto,'ClasseB',_ ).           //para o tipo objeto é indicado o nome
                                                       da classe
```

Os seguintes valores podem ser gerados para cada um dos parâmetros:

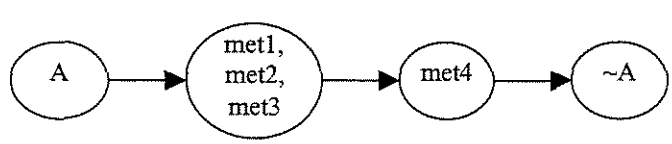
P1 ⇨ Qualquer valor entre 1 e 1000
P2 ⇨ 'farinha', 'detergente' ou 'sabao'
P3 ⇨ 2.23, 3.3, 4.5 ou 11.1
P4 ⇨ mensagem: "Parametro deve ser um ponteiro para ClasseB"
P5 ⇨ mensagem: "Parametro deve ser um objeto do tipo ClasseB"

Desse modo, estão sendo gerados valores para parâmetros do tipo inteiro, real e cadeia de caracteres, e mensagens de aviso ao usuário para ponteiros ou objetos. Parâmetros que são ponteiros ou objetos não foram tratados pela dificuldade em gerar dinamicamente um objeto de determinada classe e atribuir valores adequados a seus atributos.

❖ Estratégia para Teste de Fluxo de Transação

O modelo de fluxo de transação, como dito no capítulo anterior, é dado por um grafo onde cada nó pode conter um ou mais métodos, e os arcos representam chamadas a esse(s) método(s). As transações são geradas percorrendo-se este grafo. Cada nó do grafo é visitado, sendo feita a escolha de um (ou todos) os métodos que compõem o nó, de acordo com a estratégia escolhida pelo usuário. As estratégias implementadas são: "Primeiro" ou "Todos".

Considere a seguinte transação:



Se o usuário escolheu como estratégia "Primeiro", então a seguinte seqüência de métodos será derivada:

A ⇨ met1 ⇨ met4 ⇨ ~A

Caso a estratégia escolhida tenha sido "Todos", então as seqüências de métodos derivadas serão:

A ⇨ met1 ⇨ met4 ⇨ ~A
A ⇨ met2 ⇨ met4 ⇨ ~A
A ⇨ met3 ⇨ met4 ⇨ ~A

Cada seqüência de métodos é chamada um "objeto de teste" e será testada por um determinado caso de teste. Um objeto de teste é implementado com o seguinte formato:

objeto_tst(Identificador, Lista_métodos, Caso_teste, Arq_Casos_teste).

onde:

Identificador é um número que identifica o objeto de teste;
Lista_métodos representa uma lista dos métodos que compõem o objeto de teste;
Caso_teste indica o caso de teste que testa este objeto, e
Arq_Casos_teste representa o arquivo onde o caso de teste está implementado.

Todos os objetos de teste gerados para a classe são armazenados em arquivos nomeados com o nome da classe e a extensão ".otN", onde N é um número identificador.

5.2.2 Controlador

O Controlador é um componente da ferramenta que gerencia os relacionamentos entre a classe e seus objetos, casos, seqüência e especificação de teste. Além disso, também fornece a interface com o usuário. Ele foi codificado em C++ visando facilitar uma manutenção evolutiva na interface (para uma interface gráfica, por exemplo) ou para um outro tipo de repositório. Além disso, C++ é uma linguagem bastante conhecida e oferece mais recursos que Prolog, por exemplo. O modelo de objetos gerado no projeto deste componente está ilustrado no Apêndice B.

As principais funcionalidades fornecidas pelo Controlador, são:

- Gerência da entrada e saída de dados. A interface implementada atualmente é feita através de linha de comando. O usuário interage fornecendo os parâmetros solicitados pela ConCAT.

- Criar e persistir o relacionamento entre a classe e sua especificação de teste. Este relacionamento é persistido em um arquivo chamado "modelos.tst".

- Invocar o *Driver* Genérico, fazendo a consistência dos parâmetros passados. Essa

consistência verifica se já foram derivados para a classe objetos e casos de teste com os mesmos parâmetros. Para que essa consistência possa ser feita, a cada geração de objetos de teste para a classe e a cada geração de casos de teste são armazenadas, nos arquivos chamados "objetos.tst" e "casos.tst" respectivamente, as seguintes informações:

objeto_teste(classe, arq_objetos, estratégia_teste).

caso_teste(arq_objetos, arq_casos, estratégia_dados).

Onde:

classe é o nome da classe sob teste;

arq_objetos é o nome do arquivo que contém os objetos de teste;

arq_casos é o nome do arquivo que contém casos de teste para um determinado conjunto de objetos de teste;

estratégia_teste é a estratégia utilizada para derivar os objetos de teste ('Primeiro' ou 'Todos'), e

estratégia_dados é a estratégia utilizada para derivar os parâmetros das mensagens para classe.

- Executar uma seqüência de teste escolhida pelo usuário. Para que o Controlador saiba todas as seqüências de teste já geradas para a classe, são armazenadas em um arquivo chamado "sequencias.tst" as seguintes informações:

  sequencia_teste(classe, seqüência).

  Onde:

  classe é o nome da classe sob teste, e

  seqüência é nome do arquivo contendo a seqüência de testes para a classe.

Os arquivos "objetos.tst", "casos.tst" e "sequencias.tst" são criados pelo Controlador, mas atualizados pelo *Driver* Genérico.

A implementação atual de ConCAT não permite a manipulação de mais de uma classe por vez e, também, que uma classe possua mais de uma especificação de teste. Mas é flexível para que essas extensões possam ser feitas.

5.3 Passos para Utilização de ConCAT

Nesta seção são descritos os passos para utilização de ConCAT de acordo com a metodologia proposta no capítulo anterior:

- Construção da especificação de teste e associação desta à classe
- Instrumentação da classe sob teste
- Geração dos casos de teste e seqüência de teste
- Geração e execução do *driver* específico

5.3.1 Construção da Especificação de Teste

Depois de construído o modelo de fluxo de transação, a especificação de teste deve ser feita de acordo com o formato apresentado no capítulo 4 (figura 4.5). Este formato é compreendido pelo *Driver* Genérico.

O próximo passo é a associação da especificação à classe sob teste. Foi definido um padrão para nomear os arquivos contendo a especificação: o nome da classe e a extensão ".mt" indicando "modelo de teste". Por exemplo, o arquivo contendo a especificação de teste da classe `Produto` se chamaria "Produto.mt". Além do nome padrão, também deve ser definido um relacionamento persistente entre a especificação e sua classe. Este relacionamento é criado pelo usuário através do Controlador e armazenado em um arquivo chamado "modelos.tst" no seguinte formato:

```
modelo_teste(classe, especificacao_classe.mt).
```

Para a classe `Produto` seria armazenado o seguinte relacionamento:

```
modelo_teste(Produto,Produto.mt).
```

5.3.2 Instrumentação da Classe sob Teste

O usuário deve adicionar manualmente à CsT os mecanismos de teste embutido que são os métodos de teste de Invariante de classe e o método Relator. Para realizar esta adição, dois mecanismos foram considerados: a herança e a reflexão computacional.

A reflexão computacional pode ser definida como a capacidade de um sistema computacional desviar do seu processo normal de execução, fazer deduções ou computações em um outro nível de processamento e retornar ao nível de execução traduzindo o impacto das decisões, para então retomar o processo de execução [Lis97]. A reflexão computacional foi considerada principalmente por permitir a separação das funcionalidades de teste com as da própria aplicação de uma forma transparente ao usuário. Porém, a linguagem disponível para testes no momento em que o protótipo foi desenvolvido, Open C++1.2 [Chi93], não permitia o acesso aos atributos privados do objeto em teste na execução de um método, o que impedia o teste da invariante de classe ou qualquer outra condição envolvendo seus atributos.

Por essa razão, o meio utilizado para adicionar os métodos BIT à classe foi a herança, pois ela satisfaz os requisitos para uma interface padrão para esses métodos e a visibilidade requerida por eles. Foi definida uma classe abstrata, denominada `Autoteste`, que define a interface padrão para os métodos de teste embutido. Desse modo, a CsT deve se tornar uma classe derivada de `Autoteste`. Dois métodos BIT foram definidos: um relator (para observar o estado do objeto) e outro para o teste do predicado invariante da classe. A classe `Autoteste` está ilustrada na figura 5.5.

```
class Autoteste {
public:
      Autoteste ( ) { }
      ~Autoteste ( ) { }
      virtual void Testa_Invariante ( ) { }
      virtual void Relator (char* arq)
};
```

Figura 5.5: A classe `Autoteste`.

O usuário deve também introduzir os predicados referentes às pré-condições e pós-condições em cada método. Para o teste desses predicados foram definidas macros, mostradas na figura 5.6. Note que a macro `assert` foi redefinida também para que a seqüência de testes não fosse interrompida caso a condição neste comando fosse violada durante a execução de um caso de teste. Desse modo, o erro é registrado e o próximo caso de teste da seqüência será executado.

```
#define Invariante(exp) \
  if (!(exp))                \
     throw Erro("Invariante falhou...")

 #define Pre_condicao(exp) \
   if (!(exp))             \
      throw Erro("Pre-condicao falhou...")

 #define Pos_condicao(exp) \
   if (!(exp))             \
      throw Erro("Pos-condicao falhou...")

 #define assert(exp) \
   if (!(exp))       \
       throw Erro("Comando Assert falhou...")
```

Figura 5.6: Macros para avaliar invariante de classe, pré-condições e pós-condições.

O mecanismo de tratamento de exceção foi o meio utilizado para verificar se algum predicado foi violado durante a execução de um método da classe. A classe `Erro` foi definida para diferenciar o tratador para estas exceções de outros que já pudessem existir na classe. Existe um problema ao se utilizar ConCAT para testar uma classe que possui um tratamento de exceção, pois o mecanismo de tratamento de exceções da linguagem C++ busca primeiramente o tratador mais próximo do bloco *try* onde a exceção foi sinalizada. Caso não encontre algum, ele busca no bloco *try* mais externo e assim por diante. O bloco *try* que trata os predicados durante os testes é o mais externo, assim se a classe possuir um tratador genérico, as exceções sinalizadas pela violação destes não serão devidamente tratadas durante os testes.

Como exemplo, a figura 5.7 ilustra a classe `Produto` instrumentada. Deve-se

observar que os métodos BIT devem ser definidos entre diretivas de compilação para que existam somente durante os testes. Isto não é necessário para os predicados pois as macros foram definidas de modo que, em produção, funcionem como a macro `assert` de C++.

```
class Produto : public Autoteste{
    int qtd;
    char* nome;
    float valor_unit;
    Fornecedor* fornec;
 public:
    Produto();
    Produto(int q, char* n, float v, Fornecedor* f);
    Produto(char* nome);
    ~Produto();
      . . .
    #ifdef TESTE
      void Testa_Invariante();
      void Relator(char* arquivo);
    #endif
};

void Produto::Altera_qtd(int q)
{
   Pre_condicao(q > 0);
   qtd = q;
}
 // faz o mesmo para os demais métodos
```

Figura 5.7: Classe `Produto` instrumentada.

Os métodos BIT herdados da classe `Autoteste` devem ser redefinidos pelo usuário. Para redefinir o método que testa invariante de classe deve ser utilizada a macro mostrada na figura 5.6. O método relator possui um formato pré-definido e pode ser estendido para incluir outros dados que o usuário desejar. O essencial é que todos os atributos da classe sejam passados a um arquivo que conterá os resultados dos testes. A figura 5.8 ilustra os métodos de teste redefinidos para a classe `Produto`. Este é apenas um exemplo de como poderia ser feita a redefinição do método Relator, outros dados poderiam ser acrescentados pelo testador.

5.3.3 Geração dos Casos de Teste e Seqüência de Teste

Para realização deste passo, o usuário deve fornecer as estratégias de teste e dados para que o Controlador acione o *Driver* Genérico. Este derivará os objetos de teste de acordo com o modelo de fluxo de transação descrito na especificação de teste da classe, depois gerará casos de teste que exercitarão cada objeto de teste, e finalmente a seqüência de teste que define quais e em que ordem os casos de teste serão executados.

```
void Produto::Testa_Invariante()
{   Invariante((fornec != NULL) && (qtd >= 0) && (valor_unit >= 0)); }

void Produto::Relator(char* arquivo)
{
    ofstream resultados(arquivo,ios::app);
    if (!resultados) {
       cout << "Erro abrindo arquivo!" << endl;
       exit;
    };
    resultados << "qtd: " << qtd << "\n";
    resultados << "nome: " << nome << "\n";
    resultados << "valor unitario: " << valor_unit << "\n";
    resultados << "fornecedor: " << fornec << "\n";
    resultados.flush();
}
```

Figura 5.8: Redefinição dos métodos BIT para a classe `Produto`.

Além da reutilização de casos de teste quando classes derivadas são testadas, o *Driver* Genérico também permite que casos de teste possam ser reutilizados se um objeto de teste é re-selecionado. Por exemplo, se inicialmente o usuário gerou testes para uma classe utilizando a estratégia de teste "Primeiro" e depois gerou mais testes utilizando a estratégia "Todos", então todos os casos de teste gerados para os objetos derivados com a primeira estratégia não precisarão ser gerados novamente.

Um exemplo de caso de teste gerado está ilustrado na figura 5.9. Cada caso de teste foi definido como uma função *template* em C++ para permitir sua reutilização no teste de classes derivadas. Se fosse definido como uma função tendo como parâmetro um ponteiro para a classe base, na reutilização do caso de teste os métodos redefinidos não seriam executados pois seria considerado o método definido na classe base.

Os métodos são chamados dentro de um bloco "try" para que se possa capturar alguma exceção sinalizada pela violação de pré-condições, pós-condições ou invariante de classe. Caso uma exceção seja capturada são registrados: uma mensagem contendo o nome e os parâmetros do método onde a exceção foi sinalizada; o caso de teste que estava sendo executado; a condição violada, e o estado interno da classe, através do método Relator da CsT. Um caso de teste deveria testar um objeto de teste completo, desde a criação do objeto até sua destruição, mas para que eles pudessem ser reutilizados a construção do objeto é feita fora do caso de teste, pois, de acordo com a classificação dos métodos, o método construtor sempre será novo.

Finalmente é gerada uma seqüência de teste com a chamada de todos os casos de teste gerados e reutilizados, como mostrado na figura 5.10. A variável TESTE definida no início do programa indica à classe que ela está em modo teste e habilita os mecanismos de teste embutidos. O arquivo "classe.cc" é o arquivo contendo a classe já preparada para os testes e o arquivo "classe_ct0.cc" é o arquivo contendo os casos de teste gerados. Para cada caso de teste que será executado é criado um objeto da classe sob teste, se necessário com

parâmetros para testar construtores com atributos.

```
template <class Tipo>
void Caso_teste2_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 = "Metodo1(321,594,\"Maria\")";
      cst->Metodo1(321,594,"Maria");
      cst->Testa_Invariante();
      met = "Metodo2(1.3)";
      cst->Metodo2(1.3);
      cst->Testa_Invariante();
      arq << "Caso_teste2_0 OK!\n";
      arq.flush();
      cst->Relator("Result.txt");
      arq << "\n";
      arq.close();
      delete cst;
   }
   catch(Erro& er) {
      arq << "Caso_teste2_0 \n";
      arq.flush();
      cout << "...";
      arq << er.msg << "\n";
      arq << "Metodo chamado: " << met << "\n";
      arq.flush();
      cst->Relator("Result.txt");
      arq << "\n";
      arq.close();
   }
   catch(...) {
      arq.close();
      }
}
```

Figura 5.9: Exemplo de um caso de teste.

```
#include <fstream.h>
#include <iostream.h>
#define TESTE
#include "classe.cc"
#include "classe_ct0.cc"

void main() {
   classeA* obj1 = new classeA;
   Caso_teste1_0(obj1);
   classeA* obj2 = new classeA;
   Caso_teste2_0(obj2);
   classeA* obj3 = new classeA(848,1600,"Jose",7876);
   Caso_teste3_0(obj3);
      . . .
   classeA* objN = new classeA;
   Caso_testeN(objN);
}
```

Figura 5.10: Exemplo de uma seqüência de teste.

5.3.4 Geração e Execução do Driver *Específico*

Antes de iniciar os testes o usuário deve completar a seqüência gerada com elementos adicionais ou que não puderam ser gerados automaticamente pelo *Driver* Genérico, tais como:

- novos casos de teste;
- *stubs*, que substituem objetos usados pelo objeto em teste, e
- ponteiros ou tipos estruturados usados em atributos ou parâmetros de métodos.

O usuário pode ainda retirar casos de teste que não considere importantes ou alterar o valor gerado para os parâmetros. O próximo passo é compilar e ligar a seqüência de teste aos arquivos necessários: casos de teste, classe sob teste, classe `Autoteste` e macros.

O arquivo executável funciona como um *driver* específico para a classe sob teste. Este *driver* ao ser executado gera um arquivo de resultados contendo: a identificação dos casos de teste aplicados, uma indicação da ocorrência de erros detectados pelos predicados e os valores dos atributos dos objetos.

Se nenhum erro foi detectado, os resultados devem ser analisados manualmente pelo testador para saber se o resultado obtido é válido ou não.

5.4 Considerações ao Uso de Classes Autotestáveis

Esta seção apresenta algumas considerações feitas na aplicação da metodologia e utilização de ConCAT no caso de classes abstratas, *templates*, aninhadas e derivadas.

Relacionamentos *friend* e classes polimórficas não foram considerados, pois estas características devem ser tratadas durante os testes de integração e neste trabalho é considerado somente o teste de classes individuais.

5.4.1 Classes Abstratas

Uma classe abstrata não pode ser instanciada diretamente, então quando o *Driver* Genérico encontra uma cláusula da forma

classe('classe', **s** , n, _, []).

na especificação de teste, ele gera somente os objetos e os casos de teste para a classe. O formato destes é o mesmo que para classes comuns. Posteriormente, os objetos e casos de teste poderão ser reutilizados no teste de uma classe derivada concreta para a qual o *Driver* Genérico irá gerar uma seqüência de teste. Um exemplo da utilização de ConCAT com uma classe abstrata está ilustrado no Apêndice C.

5.4.2 Classes Aninhadas

Uma classe aninhada possui em sua especificação a definição de uma outra classe, como ilustrado a seguir.

```
Class A {
  Private:
        Class B {
         ...
        };
  Public:
         A ( ) { }
         ~A ( ) { }
         ...
};
```

```
Class C {
  Private:
        ...
  Public:
        Class D {
           ...
        };
        C ( ) { }
        ~ C ( ) { }
        ...
};
```

Este tipo de classe restringe os testes, pois a classe mais externa tem acesso somente à interface pública da classe mais interna; as demais classes podem não saber da existência da classe mais interna (como no exemplo da Classe A), ou podem ter acesso à ela (como no exemplo da Classe C) se ela for definida como pública. Como neste trabalho é considerado o teste de classes, o fato da classe interna ser definida como pública ou não na classe externa é indiferente, consideraremos nos exemplos somente a classe A.

A abordagem aplicada para o teste de classes aninhadas foi tratar a classe interna como uma classe em separado. Ela deve ser testada primeiro e se tornar uma classe autotestável, depois a classe externa deve ser testada. Assim, de acordo com o exemplo, a classe B deveria ser testada primeiro:

```
class B : public Autoteste {
   . . .
public:
   . . .
#ifdef TESTE
    void Testa_Invariante();
    void Relator(char* arquivo);
#endif
};
```

O usuário deve construir o modelo de teste e a especificação para a classe B, gerar os casos de teste e aplicá-los. Após analisar os resultados e verificar que a classe alcançou o nível de confiabilidade desejado, então deve testar a classe A seguindo os mesmos passos. O modelo de fluxo transação da classe A deve incluir somente os métodos públicos de A, uma vez que a classe B é um atributo privado. Porém, no exemplo da classe C, consideramos que a interação dos métodos públicos da classe interna D com a classe C poderia ser testada durante os testes de integração.

Os métodos BIT da classe B podem ser utilizados pelos métodos BIT da classe A, afinal o estado da classe B faz parte do estado de A. Quando a classe A é compilada em modo teste, a classe B também fica em modo teste e habilita seus mecanismos BIT. Desse modo, a classe A instrumentada ficaria do seguinte modo:

```
Class A : public Autoteste {
  Private:
        class B : public Autoteste {
        . . .
        public:
        . . .
        #ifdef TESTE
            void Testa_Invariante();
            void Relator(char* arquivo);
        #endif
        };
  Public:
        A ( ) { }
        ~A ( ) { }
        . . .
        #ifdef TESTE
            void Testa_Invariante();
            void Relator(char* arquivo);
        #endif
};
```

5.4.3 Classes Template

Classes *template* não podem ser instanciadas diretamente, elas necessitam de alguma informação adicional. Para que o *Driver* Genérico possa gerar testes para uma classe *template* o testador deve colocar na especificação de teste da classe esta informação, assim como os parâmetros necessários para instanciar um objeto desta classe, da seguinte forma:

classe('Classe', n, **s**, _ , []).
parametros_template(['**parametro1**', **...**]).

Deste modo, ao ler estas cláusulas da especificação de teste o *Driver* Genérico irá gerar uma seqüência de teste diferenciada, como a ilustrada na figura a seguir. Os casos de teste são gerados normalmente como o exemplo da figura 5.9. Um exemplo da utilização de ConCAT com classes *templates* está ilustrado no Apêndice C.

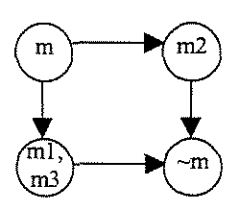
```
#include <fstream.h>
#include <iostream.h>
#define TESTE
#include "Classe.cc"
#include "Classe_ct0.cc"

void main() {
   Classe<parametro1,...>* obj1 = new Classe<parametro1,...>;
   Caso_teste1_0(obj1);
      . . .
   Classe<parametro1,...>* objN = new Classe<parametro1,...>;
   Caso_testeN_0(objN);
}
```

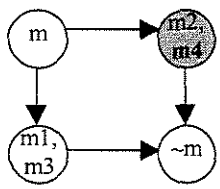
5.4.4 Classes Derivadas

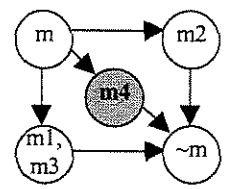
Para testar uma classe derivada é necessário que as seguintes informações de teste de sua classe base estejam disponíveis no mesmo diretório: casos de teste e objetos de teste. O *Driver* Genérico as utiliza para reutilizar casos de teste, por isso estes arquivos devem estar no mesmo diretório onde os testes para a classe derivada estão sendo gerados.

Após testar a classe base o usuário pode reutilizar seu modelo de teste para gerar o modelo da classe derivada. Para isso, devem ser feitas algumas considerações. Por exemplo, seja o grafo abaixo o modelo de fluxo de transação da classe base.

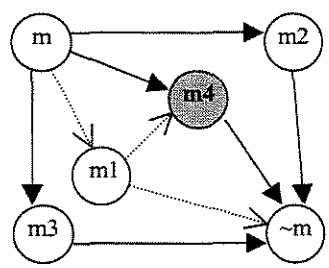


- Se foi criado um método novo, ele deve ser incluído no modelo criando um novo nó ou adicionando-o a um nó existente, dependendo de sua funcionalidade pois ela pode implicar ou não na existência de novos casos de uso. Considerando que o método novo não gera novos casos de uso ou a representação destes não afeta os nós existentes então temos a situação descrita nas figuras a seguir. Elas ilustram como ficaria o modelo para a classe derivada caso seja criado um método **m4**. Note que se for criado um novo nó, também devem ser criados os arcos representando as transações das quais este método participa.

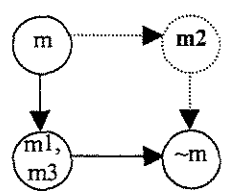




- Considere agora que a inclusão de um novo método gera novos casos de uso e a representação destes leve a divisão de um nó composto existente. Isto pode ocorrer quando o caso de uso envolve apenas algum dos métodos que compõem o nó. As transações das quais o nó composto participava, devem ser repetidas para o nó contendo método que foi separado, pois sua funcionalidade não foi alterada. A figura a seguir ilustra como ficaria o modelo da classe se a inclusão do método **m4** implicasse na criação de uma nova transação: m, m1, m4 e ~m. Os métodos m1 e m3 deveriam ser separados em nós diferentes e novos arcos para o nó contendo o método m1 deveriam ser criados, como ilustrado pelos arcos pontilhados.



- A abordagem incremental hierárquica (seção 3.3) assume que um método possa ter seu nível de visibilidade alterado para um nível mais restrito. Se algum método herdado ou redefinido deixou de ser público, ele deve ser removido do modelo pois este só espelha os métodos da interface pública da classe. Caso este método pertença a um nó individual este nó deve ser removido do modelo. A figura abaixo ilustra como ficaria o modelo se o método **m2** se tornasse um atributo privado da classe derivada.



Se um método for redefinido o modelo não sofrerá alterações, pois a abordagem incremental hierárquica assume que a funcionalidade do método deve ser mantida quando ele for redefinido para uma classe derivada.

Depois de gerado o modelo para a classe derivada, o usuário deve construir a especificação de teste para a classe indicando o nome da classe base, como ilustrado abaixo:

classe('Classe', n, n, **ClasseBase** , []).

Também deve ser indicada a classificação dos métodos de acordo com a técnica incremental hierárquica. A partir destas informações o *Driver* Genérico gerará os objetos de teste e, para aqueles objetos compostos de métodos redefinidos ou herdados, ele reutilizará o caso de teste definido nos arquivos da classe base. A seqüência de teste gerada para uma classe derivada se diferencia de outras porque pode conter mais de um arquivo de casos de teste, como ilustrado na figura a seguir. Um exemplo da utilização de ConCAT com classes derivadas está ilustrado no Apêndice C.

```
#include <fstream.h>
#include <iostream.h>
#define TESTE
#include "classeDerivada.cc"
#include "classeBase_ct0.cc"
#include "classeDerivada_ct0.cc"

void main() {
   classeDerivada* obj1 = new classeDerivada;
   Caso_teste1_0(obj1);              // caso de teste reutilizado
   classeDerivada* obj2 = new classeDerivada;
   Caso_teste2_1(obj2);              //caso de teste gerado
    . . .
}
```

5.5 Sumário

ConCAT apresenta um meio automatizado para gerar casos de teste e uma seqüência de teste para classes autotestáveis escritas em C++. Ela implementa a técnica de teste de fluxo de transação, que parece ser pouco utilizada no teste de classes pelos estudos feitos nesta dissertação, combinada com a técnica incremental hierárquica para reutilização de informações de teste. Assim, a reutilização de casos de teste pode ser feita no teste de classes derivadas e na utilização de diferentes estratégias de derivação de casos de teste para uma mesma classe. Além disso, ela fornece após a aplicação dos testes um arquivo

com informações para analisar se os resultados são válidos ou não.

Certamente, ela possui limitações: outras estratégias de teste de dados e de classes poderiam ser implementadas, um oráculo automatizado e um repositório de teste mais adequado deveriam ser providos, e uma interface gráfica auxiliaria o usuário.

Mas sua contribuição para a testabilidade da classe é importante, pois o usuário não precisa se preocupar com: a geração dos casos de teste e de uma seqüência para aplicá-los; a aplicação dos testes; e, a coleta dos resultados, que ficam disponíveis em um arquivo. Erros durante a execução dos métodos são capturados pelos predicados e registrados pela ferramenta.

Foram realizados alguns experimentos com ConCAT utilizando a metodologia descrita no capítulo 4. Estes experimentos estão ilustrados no Apêndice C.

A estratégia de dados implementada foi a geração aleatória e as classes utilizadas são simples. Foram consideradas classes abstratas, *templates* e aninhadas, classes base e classes derivadas.

Através destes experimentos foi validada a aplicabilidade da metodologia definida, bem como da ferramenta, aos tipos de classe consideradas.

Capítulo 6

Trabalhos Relacionados

Neste capítulo são apresentados alguns trabalhos apresentados na literatura que utilizam abordagens semelhantes aquelas utilizadas neste trabalho: projeto visando testabilidade e abordagem baseada em sistemas especialistas.

O capítulo está organizado do seguinte modo: na seção 6.1 é apresentado o método TOOP; na seção 6.2 é apresentada a abordagem PACT; na seção 6.3 é apresentada uma ferramenta de teste baseada em sistemas especialistas; na seção 6.4 é apresentada uma abordagem de teste baseada na comparação de componentes de software com circuitos integrados de hardware; na seção 6.5 é apresentado uma abordagem de teste baseada na inclusão de assertivas, e finalmente, na seção 6.6 é apresentado um sumário do capítulo que faz uma análise comparativa dos trabalhos apresentados.

6.1 TOOP (*Testable Object-Oriented Programming*)

TOOP [WKC+97] é um método para construir programas orientados a objetos que facilitem os testes.

Um conjunto de mecanismos testáveis embutidos foi construído para melhorar a testabilidade de sistemas orientados a objetos, considerando os aspectos de observação e controle. Estes mecanismos foram desenvolvidos baseados no conceito de estruturas de controle básicas, BCSs (do inglês, *Basic Control Structures*). BCSs são comandos fundamentais de controle do fluxo de programa, tais como iteração, condição e seqüência.

O trabalho deu ênfase ao aspecto de controlabilidade de uma BCS, definida como a capacidade de associar valor à variável de controle ou expressão da BCS, de modo que todos os caminhos indicados por ela possam ser percorridos independentemente.

Um sistema orientado a objetos testável é definido como aquele que possui mecanismos testáveis embutidos em todas as BCSs de seu código, os quais podem ser ativados mesmo que o sistema não esteja em modo teste.

A forma geral para construir as BCSs de um objeto incluindo os mecanismos de teste embutidos, é:

$$Exp \Rightarrow modo \wedge exp \vee teste$$
$$= (m_g \vee m_i) \wedge exp \vee (t_g \vee t_i)$$

onde:

exp - expressão booleana da BCS antes da inclusão dos mecanismos de teste

modo - seletor de modos, expressão booleana formada pelos valores de m_g e m_i (*False*→ modo teste, *True* → modo normal)

teste - seletor de teste, expressão booleana formada pelos valores de t_g e t_i (*False* → modo normal, *True* → modo teste)

m_g - seletor de modo global, caso seja *True* então o sistema deve ser executado normalmente.

m_i - seletor de modo individual da BCS_i , caso seja *True* então a BCS específica deve ser executada normalmente.

t_g - seletor de teste global, caso seja *True* então o sistema está em modo teste.

t_i - seletor de teste individual do BCS_i , caso seja *True* então a BCS específica está em modo teste.

Os seletores de modo e teste permitem que cada BCS possa ser controlada em modo teste, independente do valor de sua expressão original. Por exemplo, um comando *if-then* poderia ser definido como:

If exp *then* P $\Rightarrow$ *If* $[((m_g \vee m_i) \wedge exp) \vee (t_g \vee t_i)]$ *then* P;

No caso de uma estrutura *for-do*, são incluídas duas variáveis de teste, t_a e t_b, para controlar o número de iterações do loop. Ela poderia ser definida como:

```
For i := i_a to i_b do
        P;
⇒
If [(m_g ∨ m_i) ∧ (¬(t_g ∨ t_i))]
  Then                          // modo normal
       For i := i_a to i_b do
              P;
  Else                          // modo teste
   {
       i_a := t_a ;
       i_b := t_b ;
       For i := i_a to i_b do
              P;
    };
```

Objetos testáveis podem ser construídos utilizando estes mecanismos. A estrutura geral de uma classe testável é ilustrada na figura 6.1. Os mecanismos testáveis do objeto

(variáveis, casos de teste e resultados esperados) devem estar explicitamente declarados na interface do objeto.

```
Class nome-classe {
        // interface
        dados;
        construtor;
        destrutor;
        interface teste {
                variáveis de controle de teste
                Boolean m_g , // seletor de modo global
                        t_g , // seletor de teste global
                        m_i , // seletor de modo individual, 1≤ i ≤ n
                              // n é o número de BCSs no objeto
                        t_i ; // seletor de teste individual, 1≤ i ≤ n
                testes
                T_τ ;   // testes embutidos, 1≤ τ ≤ k, k é o número de testes embutidos
                respostas dos testes
                R_τ ;   // respostas esperadas dos testes embutidos
        }
        // implementação das funções
        if (m_g ∧ (¬ t_g))
                then                    // modo normal
                    funções;
                else                    // modo teste
                    { if ¬ m_i ∧ t_i    // i = 1, 2, ..., n
                         then BCS_i testável;
                         else BCS_i normal;
                         ...
                    }
} [lista de objetos];
```

Figura 6.1: Especificação de um objeto testável.

A utilização destes mecanismos permite que uma equipe realize testes sem conhecer a estrutura interna do objeto, pois em modo teste, este já aciona suas funcionalidades de teste embutidas. O método TOOP adota o conceito de projeto para testabilidade, de modo que sua aplicação permite que os testes embutidos e seu código possam ser herdados e reutilizados. Assim os testes e a manutenção do sistema são simplificados.

6.2 PACT (*Parallel Architecture for Component Testing)*

Este trabalho [McG96] apresenta uma abordagem para realizar o teste de componentes de software individuais. Em um software orientado a objetos, o componente é, portanto, um objeto. A abordagem adotada auxilia no aumento da capacidade de observação do

componente testado, bem como integra e apoia as estratégias de desenvolvimento e teste necessárias para uma alta qualidade do software.

A abordagem propõe a criação e manutenção de uma arquitetura paralela, chamada PACT, como ilustrada na figura 6.2. Nesta arquitetura existem classes que manipulam casos de teste; para cada classe da aplicação é desenvolvida uma classe deste tipo, e um conjunto de classes genéricas de auxílio aos testes. A abordagem assume que uma equipe de teste construirá este conjunto de classes genéricas.

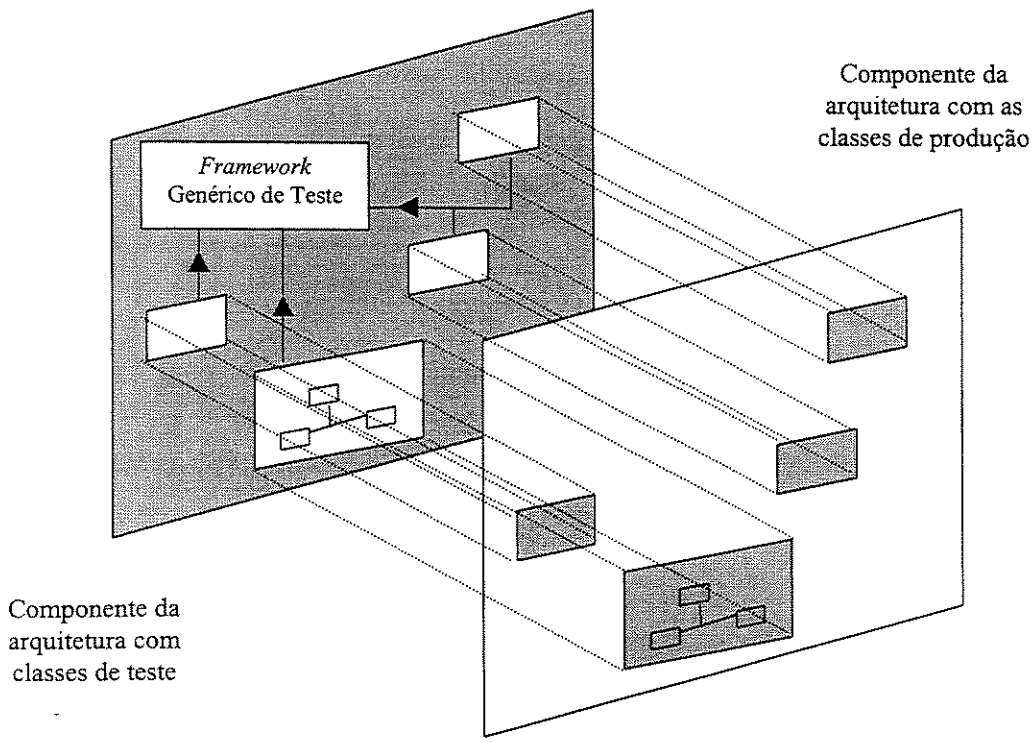


Figura 6.2: Estrutura de PACT.

As classes de teste devem ser obtidas especializando-se alguma das classes pertencentes ao conjunto de classes de teste genéricas (do inglês, *Generic Test Harness*). Este conjunto de classes deve oferecer:

- ambiente para realização dos testes;
- interface para ferramentas de teste externas, como ferramentas de *capture/playback* para teste de interfaces por exemplo;
- um sistema de arquivos de log compartilhado, para armazenar resultados de teste, saídas capturadas dos programas, etc.;
- um mecanismo de tratamento de exceção que possa capturar uma exceção sinalizada pelo objeto sob teste e verificar se ela era esperada ou não;
- mecanismos para inicialização de *hardware* necessário para algum teste específico;
- *templates* de casos e seqüências de teste.

PACT permite o mapeamento de um relacionamento de herança entre classes de produção para as classes de teste correspondentes, como ilustrados na figura 6.3. Desse modo, são criadas hierarquias de classes de teste permitindo que os casos de teste definidos na classe base sejam herdados pelas classes de teste derivadas. Assim, somente a classe base no nível mais alto da hierarquia será derivada de uma classe do conjunto de classes genéricas.

Com relação a ocultação de informação entre a classe de teste e a classe de produção correspondente, PACT propõe a utilização de mecanismos específicos da linguagem na qual o sistema está implementado. Por exemplo, no caso de C++ utiliza-se o conceito de classes *friend* e em Ada 95 utiliza-se o conceito de unidades filhas.

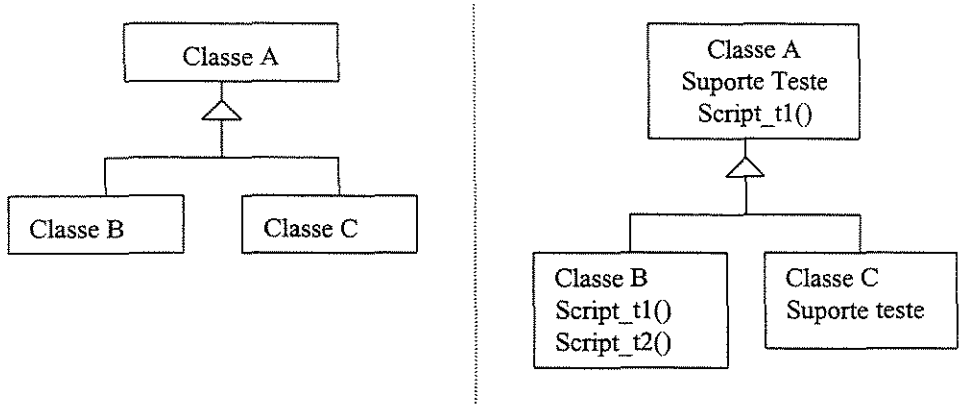


Figura 6.3: Reutilização de código na hierarquia de teste paralela.

A abordagem PACT pode ser usada também para o teste: de sistemas embutidos, que devem ter códigos de produção tão pequenos quanto possível; de grandes projetos que provavelmente irão reutilizar casos de teste, e de projetos de *frameworks* cujas aplicações instanciadas provavelmente irão reutilizar casos de teste. Algumas vantagens apresentadas por PACT são:

- Vários níveis de cobertura de teste podem ser considerados;
- As funcionalidades de teste e da aplicação são claramente separadas;
- Não aumenta o tamanho do código executável entregue ao usuário;
- As informações de teste são organizadas de modo a facilitar sua reutilização;
- O esforço necessário para atualizar as informações de teste após uma alteração do software é minimizado, e
- A interação entre as equipes de teste e desenvolvimento se torna mais fácil.

6.3 Ferramenta de Teste de Programas O-O Baseada em Técnica de Sistemas Especialistas

Em [CCC99] é proposta uma ferramenta para o teste de sistemas orientados a objetos baseada em sistemas especialistas. A ferramenta é composta de três partes principais, como ilustrado na figura 6.4.

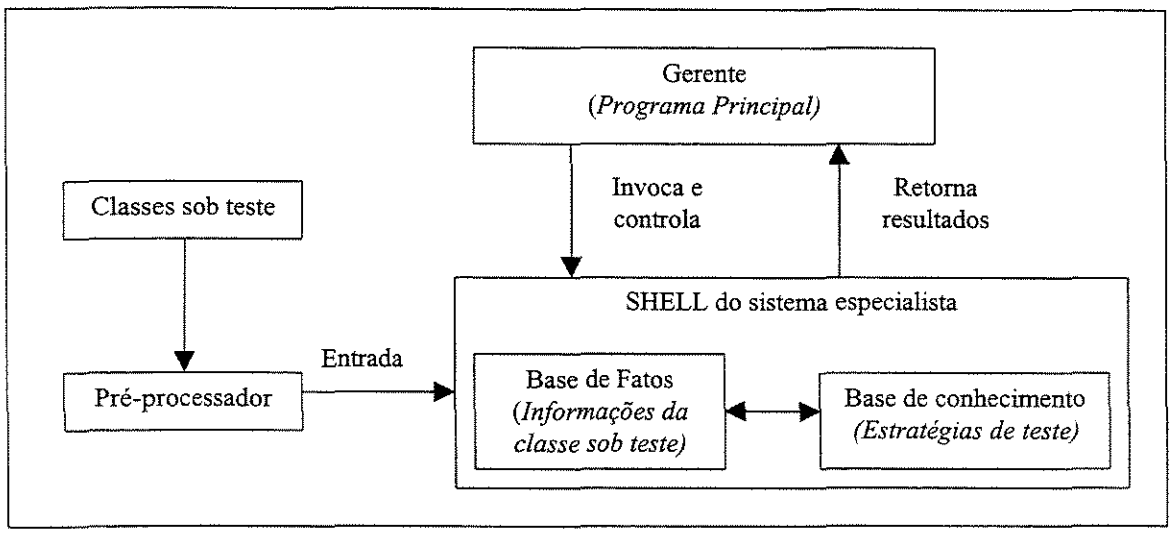


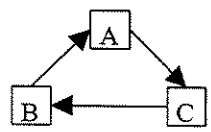
Figura 6.4: Componente principais da ferramenta.

O Gerente é responsável por iniciar o sistema e realizar a sessão com o engenheiro de teste. Ele possui mecanismos para invocar as estratégias de teste, uma interface gráfica para o sistema especialista e um editor de texto para as regras ou o código fonte da classe.

A classe sob teste representa o arquivo com o código fonte da classe a ser testada. O pré-processador é um tipo de *parser* que obtém as informações da classe sob teste e as transforma em fatos que são colocados na base de fatos dentro do sistema especialista. O sistema especialista é composto de uma shell, uma base de fatos e uma base de conhecimento. A ferramenta CLIPS, utilizada como shell, oferece os meios de acesso à base de conhecimento e à base de fatos, e de execução dos procedimentos de teste. A base de conhecimento contém regras que definem as estratégias de teste. Tais estratégias abordam os seguintes problemas:

- Problemas de herança
  - ⇒ Duplicação de métodos: quando ocorre a herança múltipla e as classes base derivam de uma classe em comum ou derivam uma da outra;
  - ⇒ Duplicação de nomes de métodos: quando ocorre herança múltipla e as classes base possuem métodos com assinaturas idênticas;

⇒ Herança cíclica: quando uma classe base possui como uma de suas classes ancestrais uma classe que foi derivada dela, como ilustrado abaixo.



- Problemas de classes amigas
  ⇒ Relacionamento mútuo: quando as classes são amigas uma da outra;
  ⇒ Relacionamento de herança entre classes amigas: quando uma classe derivada é amiga de sua classe base.

- Problemas de ordem de teste
  Seqüências incorretas de chamada de métodos podem causar um erro em tempo de execução.

- Problemas de ligação dinâmica
  A determinação do código a ser executado em tempo de execução.

- Problemas de alocação de memória
  São abordados os problemas com a alocação e liberação de espaço de memória para a criação e destruição do objeto, especialmente em linguagens como C++ onde tal controle é feito pelo programador.

As estratégias utilizadas na ferramenta podem ser classificadas em três categorias: análise estática, testes e definidas pelo usuário.

As falhas relacionadas a herança e classes amigas são tratadas por análise estática. Elas são definidas como padrões e a análise estática tem como objetivo encontrar estes padrões dentro da classe ou conjunto de classes. A análise estática é composta de três fases:

1. Derivação de fatos a partir daqueles gerados pelo pré-processador.
2. Redução de hierarquias de classes complexas a grafos direcionados.
3. Indicação de falhas encontradas nos grafos baseados nos padrões de falhas.

Os demais problemas são tratados por testes. A partir da hierarquia de herança descrita nos fatos pode ser gerada uma lista de classes a serem testadas considerando a dependência entre elas. Tomando uma classe desta lista por vez, o testador gera os casos de teste para a classe escolhida e a ferramenta os executará depois. Tais casos de teste visam produzir defeitos decorrentes da execução de uma seqüência incorreta de chamadas de métodos. Para os problemas de alocação/liberação de memória, a estratégia de teste gera programas que fazem a criação e destruição de objetos.

A ferramenta permite que o usuário possa definir novas estratégias de teste e adicioná-las à base de conhecimento. Isto pode ser feito definindo-se novas regras ou alterando as existentes.

Algumas vantagens da abordagem baseada em sistemas especialista são:

- maior flexibilidade para modificar ou incorporar métodos de teste;
- facilidade de entendimento das estratégias de teste;
- capacidade para relatar o estado da execução dos testes e adicionar comentários as estratégias de forma a torná-las mais claras.

6.4 Circuitos Integrados de Software

Este trabalho [Hof89] apresenta uma abordagem para os testes de regressão de componentes de software baseada nos princípios de teste de circuitos integrados.

São utilizados os conceitos de controlabilidade e observabilidade como sendo a facilidade de aplicar uma entrada arbitrária a um módulo e a facilidade para se observar o comportamento deste módulo, respectivamente.

A abordagem segue três princípios básicos:

⇒ Testes de regressão sistemáticos
É importante planejar os testes antes da implementação e manter o código de teste juntamente com o código de produção. Em testes bem sucedidos, o ganho com as falhas detectadas supera os custos de gerar, manter, executar e avaliar os testes.

⇒ Projeto visando testabilidade
A fim de ter uma boa relação custo x benefício, a testabilidade deve ser um importante critério de projeto. A utilização de *drivers* e *stubs* no teste de componentes permite que este possa ser testado isoladamente. Deste modo, a controlabilidade e a observabilidade do mesmo aumentam e consequentemente, sua testabilidade.

⇒ Considerar a relação custo x benefício da automatização
A automação de tarefas que são produtivas feitas manualmente ou que não possuem um processo bem definido podem resultar em alto custo e não trazer os resultados esperados. Porém, tarefas que são repetitivas e realizadas freqüentemente são relativamente mais fáceis de automatizar e os resultados são muito mais evidentes. Assim, considerando os testes de regressão, o desenvolvimento dos casos de teste poderia ser manual, enquanto que sua execução e avaliação poderiam ser feitas automaticamente.

A fim de aumentar a controlabilidade e observabilidade do componente são adicionados procedimentos específicos para testes à interface deste, correspondendo aos

pinos de teste de um circuito integrado. São também adicionadas assertivas ao código do componente correspondendo aos circuitos de teste embutidos nos circuitos integrados.

Os casos de teste são definidos formalmente com a utilização de uma linguagem para representar o traço do componente. Traços são seqüências de chamadas aos programas de acesso ao componente, por exemplo, no caso de um objeto seriam os métodos. A sintaxe de um traço é a seguinte:

s_inicio( ).s_InsSimb("gato").g_RetornaId("gato")

onde, *s_inicio()*, *s_InsSimb(p)* e *g_RetornaId(p)* são programas de acesso ao componente.

Os casos de teste são descritos indicando um traço e associando a ele o comportamento requerido do componente para aquele traço. Eles são definidos através de uma quíntupla:

<traço, exceção_esperada, valor_real, valor_esperado, tipo>

onde:

traço - traço utilizado para exercitar o componente;

exceção_esperada - nome da exceção que o traço deveria gerar ou a palavra-chave *noexc* se nenhuma exceção é esperada;

valor_real - uma expressão que deverá ser avaliada depois do traço e cujo valor resultante será considerado o valor do traço;

valor_esperado - valor que *valor_real* deveria retornar, e

tipo - tipo do dado de *valor_real* e *valor_esperado*.

A palavra-chave *empty* pode ser utilizada em casos de teste onde não seja necessário fornecer o valor real ou esperado do traço, por exemplo casos de teste que verificam se as exceções esperadas foram sinalizadas.

Um script de teste é definido como uma lista de programas de acesso e declarações de exceções, uma lista de casos de teste e, opcionalmente, um código em C. Ele pode ser visto como uma especificação parcial do componente, expressando o comportamento requerido sob determinadas circunstâncias.

Foi construída uma ferramenta automatizada, chamada PGMGEN, para gerar programas que apliquem os casos de teste (*drivers*) e verifiquem se o resultado foi igual ao esperado. Porém, não existe um oráculo para geração dos resultados esperados dos testes.

Para permitir a geração automática de *drivers* foi adotado um padrão para invocar os programas de acesso e sinalizar exceções. Cada programa de acesso deve ser implementado como uma função C. Para cada exceção deve ser implementada uma função em C com o mesmo nome que deverá agir como o tratador da exceção.

6.5 Assertivas

Neste trabalho [TR94] são dadas recomendações para manter uma alta testabilidade durante o projeto de um sistema orientado a objetos, entre elas o uso de assertivas. O conceito de testabilidade utilizado é o dado por Voas [VM95]: "uma previsão da probabilidade de um defeito de software ocorrer devido a existência de uma falha", dado que os testes são realizados utilizando uma determinada distribuição de entrada selecionada por uma técnica específica.

Para sistemas que exijam um alto nível de confiabilidade, testar o software somente talvez não seja suficiente. A análise de sensibilidade de software pode ser utilizada como um técnica de validação complementar para verificar se é possível alcançar o nível de confiabilidade desejado para um determinado sistema somente com testes.

A análise de sensibilidade permite determinar quando o nível de confiabilidade foi alcançado, permitindo que os testes possam ser interrompidos. Dessa forma pode-se também prever as prováveis regiões do código onde as falhas, caso existam, sejam mais difíceis de serem encontradas. A análise de sensibilidade para software procedimental foi implementada por uma ferramenta chamada PiSCES Software Analysis Toolkit. Através da análise de sensibilidade são indicados locais do código onde seria interessante inserir uma assertiva.

As assertivas devem ser derivadas da especificação. Neste trabalho é indicado que o uso de assertivas nunca diminui a testabilidade de um sistema.

Foi proposta uma ferramenta de auxílio a inserção de assertivas em sistemas orientados a objetos, especialmente nas áreas de código onde a testabilidade é menor. Deste modo, objetos reutilizáveis podem ser testados suficientemente e o sistema composto pode ser considerado de alta confiabilidade. Esta ferramenta poderá ser utilizada em conjunto com uma outra ferramenta de Análise de Testabilidade de Software C++.

6.6 Sumário

Neste capítulo apresentamos alguns trabalhos baseados em conceitos semelhantes aos utilizados nesta dissertação. A tabela 2 apresenta um resumo das características destes trabalhos, comparando-as ao trabalho apresentado aqui.

Trabalhos	Problema Tratado	Solução	Ferramenta	Permite geração de testes automática?	Oráculo
TOOP	Melhorar controlabilidade e observabilidade.	Uso de assertivas para cada estrutura de controle básica (BCS).	Não possui.	Sim.	Não aborda o problema.
PACT	Melhorar observabilidade de componentes de software; facilitar a manutenção e reutilização de testes.	Utilizar mecanismos próprios da linguagem em que o sistema está implementado; criar e manter uma arquitetura paralela para manter as informações de teste.	Não se aplica, pois define uma arquitetura.	Não define um método de teste específico.	Não aborda o problema.
Ferramenta baseada em Sistemas especialistas	Melhorar testabilidade através de suporte automatizado para execução dos testes	Fornecer uma base de conhecimento com estratégias de teste implementadas, um *parser* para retirar informações da classe e uma interface para controlar a aplicação dos testes.	Sim.	Sim.	Para análise estática define os padrões de falhas.
CI's de Software	Melhorar controlabilidade e observabilidade de componentes; fornecer suporte automatizado para facilitar os testes de regressão.	Incluir assertivas no código e procedimentos específicos de teste; implementar uma ferramenta que gera *drivers* de teste.	PGMGEN que implementa a geração de *drivers*.	Não.	Considera um oráculo manual.
Uso de Assertivas	Melhorar a testabilidade facilitando a propagação de falhas.	Incluir assertivas em pontos do código mais prováveis de esconder falhas.	PiSCES faz a análise de sensibilidade do código.	Não se aplica.	Não aborda o problema do oráculo.

CONCAT	Melhorar a controlabilidade e observabilidade de classes.	Incluir procedimentos específicos de teste na classe e assertivas no código dos métodos; definir um formato para especificação de teste.	CONCAT faz a geração e execução dos testes.	Sim.	Considera um oráculo manual, exceto para as assertivas

Tabela 2: Características dos trabalhos relacionados.

TOOP, PACT, CI's de Software e Utilização de Assertivas abordam o projeto para testabilidade. Com exceção de PACT, os demais enfocam as capacidades de controle e observação assim como ConCAT. Porém, ConCAT se mostra mais completa que estes trabalhos pois:

- além de assertivas, define também métodos padrões específicos para teste, e
- permite a geração automática de casos testes.

PACT enfatiza a separação entre as funcionalidades de teste e da aplicação e a facilidade de manter e reutilizar as informações de teste. Como ela define uma arquitetura de teste, sua aplicação não se restringe a sistemas escritos em uma determinada linguagem. Contudo, apesar de ser desenvolvida para sistemas escritos em C++, ConCAT também aborda o aspecto de controle, e fornece auxílio automatizado para a geração e execução dos testes.

O outro trabalho estudado, dado em [CCC99], apresentou uma ferramenta para teste de sistemas orientados a objetos que utiliza uma abordagem de sistemas especialistas. Assim como ConCAT ela utiliza uma base de conhecimento para gerar casos de testes e possui um conjunto de fatos sobre a classe sob teste. Porém, além da aplicação dos testes, ConCAT também utiliza assertivas para detectar erros durante a execução dos métodos.

Nenhum dos trabalhos apresenta uma solução para o problema do oráculo, ou é considerado que os resultados esperados são obtidos manualmente ou são verificados tipos padrões de falhas ou exceções esperadas.

Capítulo 7

Conclusões e Trabalhos Futuros

A utilização do paradigma orientado a objetos tem sido cada vez maior e a possibilidade de reutilizar componentes é um dos grandes atrativos deste paradigma.

A confiabilidade de um componente reutilizável, mais que em outro software, é um requisito muito importante. Para obter o nível de confiança desejado o componente deve ser bem validado. Uma das atividades de validação são os testes.

O processo de testar o componente implica em custos adicionais ao desenvolvimento e manutenção do sistema. Esses custos dependem da testabilidade do componente, ou seja, de quão fácil é testá-lo. A testabilidade ganha mais importância quando se trata de componentes reutilizáveis, pois tais componente precisam ser testados a cada vez que são reutilizados.

Neste trabalho foi estudada uma proposta para aumentar a testabilidade de um sistema orientado a objetos, dada em [Bin94a]. Ela faz uma comparação entre classes e circuitos integrados projetados visando testabilidade. Tais circuitos possuem componentes embutidos de autoteste, sendo chamados de autotestáveis. Da mesma forma, foi proposto que classes fossem construídas visando a testabilidade com mecanismos de teste embutidos e um *driver* genérico capaz de gerar e aplicar casos de teste sobre a classe. A classe resultante deste processo foi chamada "classe autotestável".

O objetivo principal foi verificar a viabilidade de implementar essa proposta. Isto foi comprovado através do desenvolvimento de uma metodologia para construir uma classe autotestável e de um protótipo que automatiza parte desta metodologia.

Neste capítulo são apresentados: o que foi realizado neste trabalho, as contribuições desta dissertação e sugestões de trabalhos futuros.

7.1 O que foi Realizado

Inicialmente foram realizados estudos sobre a técnica de projeto visando testabilidade utilizada na construção de circuitos autotestáveis para compreender melhor a proposta dada em [Bin94a]. Esta apresentava alguns pontos em aberto que precisavam de uma definição para que pudesse ser implementada.

O primeiro ponto definido foi o modelo de teste que seria utilizado. O modelo de fluxo de transação foi escolhido por ser um modelo de teste caixa-preta e testar os casos de uso da classe. Testes de caixa-preta são mais indicados para classes que serão reutilizadas pois a funcionalidades destas sofrerão menos variações que seu código fonte.

Foi decidido também a utilização da técnica incremental hierárquica para possibilitar a reutilização de casos de teste, desta forma aumentando ainda mais a testabilidade de classes derivadas.

O próximo passo foi definir uma especificação de teste que descrevesse o modelo de teste escolhido e informações da classe necessárias para os testes. Uma das considerações feitas para construção desta especificação é que ela deveria ser simples e fácil para que o usuário pudesse construí-la.

Outro ponto muito importante definido foi o formato de uma classe autotestável. Isto envolveu a decisão de quais métodos de teste seriam utilizados e qual mecanismo seria utilizado para que eles fossem "embutidos" na classe. Optou-se por utilizar um método para testar invariante de classe e outro para relatar o seu estado interno. O mecanismo escolhido para embutir estes métodos foi a herança.

Para descrever os passos necessários para construir e utilizar uma classe autotestável foi desenvolvida uma metodologia. Com ela, o usuário pode gerar o modelo de fluxo de transação para a classe, construir sua especificação de teste e preparar a classe para os testes.

Também foi construído um protótipo, chamado ConCAT, de uma ferramenta que implementa parte desta metodologia. Ele possui dois componentes principais: um *Driver* Genérico que gera objetos, casos e a seqüência de teste para a classe; e o Controlador, que possui a interface e faz a gerência do repositório de dados.

Foram realizados testes com ConCAT visando a aplicação da metodologia em vários tipos de classe. Foram utilizados exemplos de classes abstrata, derivada, base, composta e *template*.

7.2 Contribuições

Neste trabalho foi estudada uma forma de projetar classes com maior testabilidade. A construção de classes autotestáveis se mostrou viável a partir do momento em que pudemos implementá-la através de uma metodologia e uma ferramenta automatizada.

Algumas contribuições apresentadas por ele são:

- Um modo de adicionar a uma classe escrita em C++ métodos de teste que auxiliarão a realização da fase de testes.

- A gramática de uma especificação de teste que descreve o modelo de teste de fluxo de transação para uma classe.

- A metodologia que descreve os passos para gerar a especificação e adicionar os métodos de teste, preparando a classe para os testes.

- A utilização de predicados na forma de invariantes de classe, bem como pré e pós condições de métodos, para aumentar a testabilidade de classes.

- Uma ferramenta automatizada para gerar testes a partir desta especificação. Esta ferramenta é capaz de reutilizar casos de teste no teste de classes derivadas, utilizando a abordagem incremental hierárquica. Além disso, ela também aplica os testes sobre a classe e gera um arquivo com os resultados dos testes que auxiliam a verificação dos mesmos.

7.3 Trabalhos Futuros

Como sugestões de trabalhos futuros são propostas:

- Aplicar a metodologia e ferramenta desenvolvidas neste trabalho em classes de um sistema real, a fim de verificar a viabilidade de utilização.

- Aprimorar a ConCAT: adicionando mais técnicas de teste, o que implica em desenvolver ou modificar a especificação de teste existente; alterar o repositório de dados para um banco de dados; utilizar um oráculo automatizado para não precisar mais validar os testes manualmente; e, tornar a interface mais amigável.

- Realizar uma série de experimentos com ConCAT para verificar sua eficiência em relação a outras ferramentas semelhantes.

Apêndice A

Gramática da Especificação de Teste para o Modelo de Fluxo de Transação

A gramática dada a seguir representa a especificação de teste utilizada neste trabalho. Ela está descrita segundo uma notação BNF (*Backus Naur Form*) estendida. Nela, os símbolos terminais são escritos em negrito, símbolos não terminais são representados entre os símbolos "<" e ">". Símbolos terminais formados de apenas um caracter são representados entre aspas duplas.

<especificação> ::= <classe_def> "." [<param_template>] [<atrib_def>] <met_def> [<param_def>] <nós_def> <arcos_def>

<classe_def> ::= **classe(** <nome_classe> "," <abstrata> "," <template> "," <nome_super> "," <arq_include> ")"

<param_template> ::= **parametros_template([** [<valor> { "," <valor> }] **]).**

<atrib_def> ::= {<atributo> "."}

<met_def> ::= {<método> "."}

<param_def> ::= {<parâmetro> "."}

<nós_def> ::= {<nó> "."}

<arcos_def> ::= {<arco> "."}

<nome_classe> ::= <nome>

<abstrata> ::= "s" | "n"

<template> ::= "s" | "n"

<nome_super> ::= <nome> | <símbolo_especial>

<arq_include> ::= **[ ]** | "[" <nome_arq> [{"," <nome_arq>}] "]"

<valor> ::= <nome> | <número>

<atributo> ::= **atributo(** <nome_atributo> "," <tipo> ")"

<método> ::= **metodo(** <id_met> "," <nome_met> "," <tipo_met> "," <classificação> "," <num_param> ")"

<parâmetro> ::= **parametro(** <nome_parâmetro> "," <id_met> "," <tipo> ")"

<nó> ::= **no(** <id_no> "," <nó_inicial> "," <num_arcos> "," <lista_métodos> ")"

<arco> ::= **arco(** <nó_origem> "," <nó_destino> ")"

<nome> ::= " ' " letra {letra | dígito} " ' "

<símbolo_especial> ::= "_"

<nome_arq> ::= <nome>

<nome_atributo> ::= <nome>
<tipo> ::= <intervalo> | <string> | <conjunto> | <objeto> | <ponteiro>
<id_met> ::= **m**<número>
<nome_met> ::= <nome>
<tipo_met> ::= <nome> | <símbolo_especial>
<classificação> ::= **novo** | **redefinido** | **recursivo** | **construtor** | **destrutor**
<num_param> ::= <número>
<nome_parâmetro> ::= <nome>
<id_no> ::= **no**<número>
<nó_inicial> ::= **sim** | **nao**
<num_arcos> ::= <número>
<lista_métodos> ::= [ ] | "[" <id_met> [{"," <id_met>}] "]"
<nó_origem> ::= <id_no>
<nó_destino> ::= <id_no>
<intervalo> ::= **intervalo_inteiro** "," <valor_max> "," <valor_min>
<string> ::= **string** "," <lista_string> "," <simbolo_especial>
<conjunto> ::= **conjunto** "," <lista_valores> "," <símbolo_especial>
<objeto> ::= **objeto** "," <nome_classe> "," <símbolo_especial>
<ponteiro> ::= **ponteiro** "," <nome_classe> "," <símbolo_especial>
<valor_max> ::= <número>
<valor_min> ::= <número>
<lista_string> ::= "[" <nome> [{"," <nome>}] "]"
<lista_valores> ::= "[" <número> [{"," <número>}] "]"

Apêndice B

Componentes de ConCAT

Neste apêndice são apresentados detalhes de implementação sobre os principais componentes de ConCAT: o Controlador e o *Driver* Genérico. Sobre o primeiro é apresentado seu modelo de objetos na seção B.1, e sobre o segundo são apresentadas as regras que o compõem na seção B.2.

B.1 Controlador

O Controlador foi codificado em C++ e possui o modelo de objetos descrito na figura B.1. A seguir é apresentado resumidamente cada classe e os métodos mais importantes.

- Classe Controlador: responsável por inicializar os demais objetos e controlar as ações sobre a classe sob teste.
  - *Inicializar*: inicia o menu principal da ConCAT e gerencia as opções escolhidas pelo usuário chamando o método adequado do objeto CST.

- Classe CST: representa as configurações de teste para a classe sob teste.
  - *Associar_modelo*: responsável por criar um relacionamento persistente entre a classe e sua especificação de teste. Este relacionamento é armazenado no arquivo "modelos.tst".
  - *Verificar_modelos*: verifica os modelos de teste existentes para a classe sob teste, armazenados no arquivo "modelos.tst" e instancia objetos da classe Modelo_teste.
  - *Gerar_sequencia*: recebe como parâmetros as estratégias de teste de dados e teste de fluxo de transação e depois chama o método *Gerar_sequencia* do objeto Modelo_teste.
  - *Executar_sequencia*: apenas faz a chamada do método Executar_sequencia do objeto Modelo_teste.

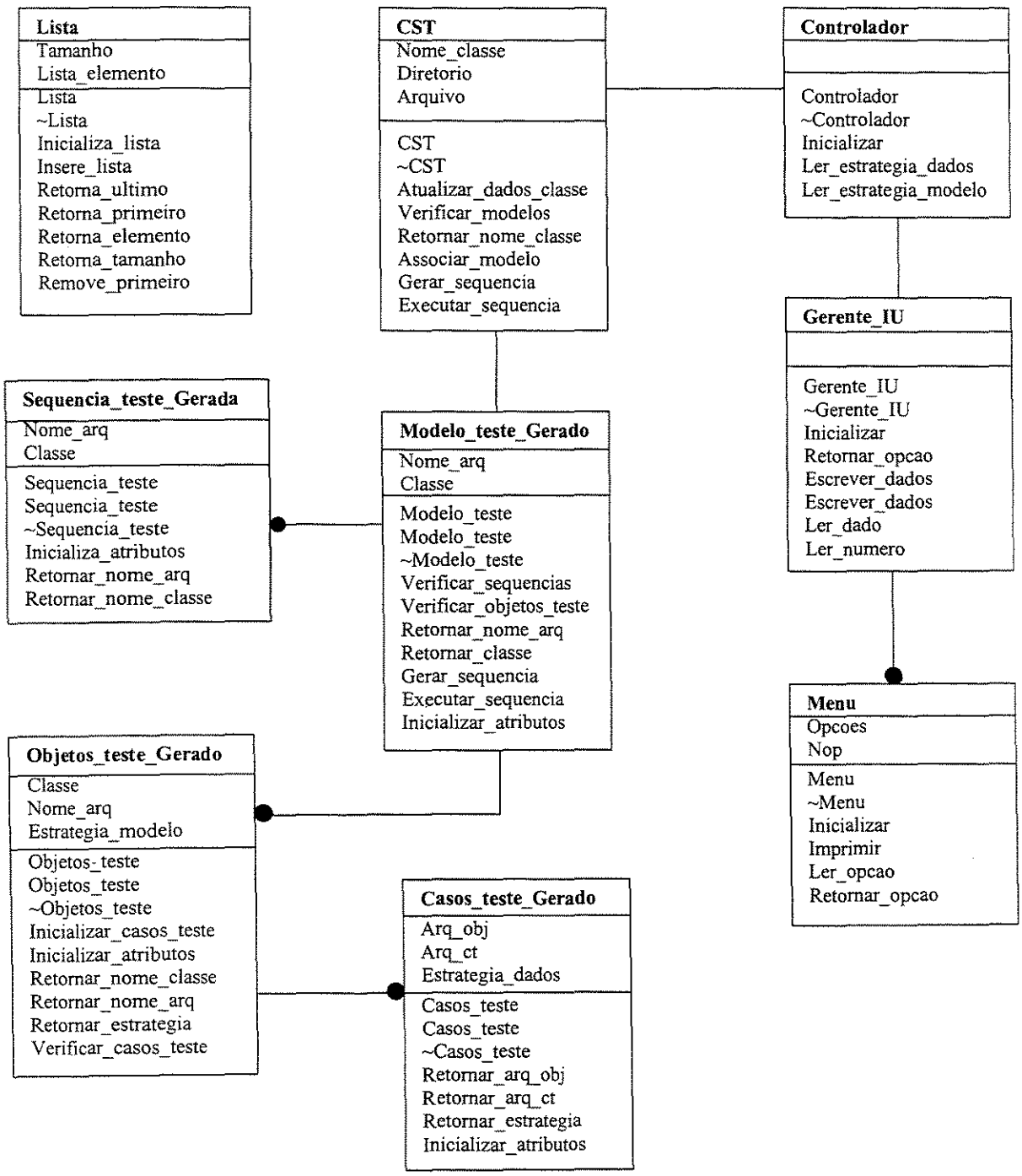


Figura B.1: Modelo de objetos do Controlador.

- Classe Modelo_teste_Gerado: representa a associação entre a especificação de teste e a classe sob teste.
  - *Verificar_sequencias*: verifica quais seqüências de teste a classe possui armazenadas no arquivo "sequencias.tst" e instancia objetos da classe Sequencia_teste.
  - *Verificar_objetos_teste*: verifica quais arquivos de objetos de teste a classe possui armazenados no arquivo "objetos.tst" e instancia objetos da classe Objeto_teste.
  - *Gerar_sequencia*: verifica se existe uma seqüência de teste gerada com a mesma estratégia de teste de dados e teste de fluxo de transação. Depois, inicia o *Driver* Genérico com os parâmetros fornecidos pelo usuário para que sejam gerados os casos de teste, objetos de teste e a seqüência de teste para a classe.
  - *Executar_sequencia*: verifica as seqüências de teste que a classe possui e executa aquela que o usuário escolher.

- Classe Objetos_teste_Gerado: representa a geração de objetos de teste já feita para a classe sob teste indicando a estratégia utilizada para realizá-la.
  - *Inicializar_casos_teste*: verifica os arquivos de casos de teste derivados para o arquivo de objeto de teste, armazenados no arquivo "casos.tst" e instancia objetos da classe Casos_teste.
  - *Verificar_casos_teste*: verifica se algum arquivo de casos de teste foi derivado com uma determinada estratégia de teste de dados.

- Classe Casos_teste_Gerado: representa a geração de casos de teste já feita para a classe, indicando quais objetos de teste estão sendo testados e qual a estratégia utilizada para geração de dados.

- Classe Sequencia_teste_Gerada: representa as seqüências de teste já geradas para a classe sob teste.

- Classe Lista: é uma classe auxiliar para representar o relacionamento de agregação.

- Classes Gerente_IU e Menu: são responsáveis pela interface com usuário. A interface implementada não é gráfica, mas feita através de linha de comando.

B.2 O Driver Genérico

O *driver* genérico foi implementado na linguagem Prolog. Nesta seção são apresentadas as regras que o compõem. Inicialmente são gerados os objetos de teste, depois os casos de teste e finalmente, a seqüência de teste.

```
gerador_sequencia_tst(Sistema,Espec,Gera_val,Exec_met) :-
        consult(Espec),
```

```
        classe(_,Abstrata,_,_,_),
        gerador(Abstrata,Sistema,Gera_val,Exec_met).

gerador(n,Sistema,Gera_val,Exec_met) :-
        gerar_objetos_tst(Exec_met,Gera_val),
        gerar_sequencia_tst(Sistema,Gera_val).

gerador(s,_,Gera_val,Exec_met) :-
       gerar_objetos_tst(Exec_met,Gera_val).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Regras para geracao dos objetos de teste                                    %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
gerar_objetos_tst(Exec_met,Gera_val) :-
        classe(X,_,_,_,_),   %%Procura o numero correto para nomear o arquivo de objetos
        listar_arq_objetos(X,Lista_arq_obj),
        length(Lista_arq_obj,N),
        concat('.ot',N,Extensao),
        concat(X,Extensao,Arquivo),
        /*Adiciona ao arquivo o nome do novo arquivo de objetos de teste*/
        append('objetos.tst'),
        write('objeto_teste('''),
        write(X),
        write(''','''),
        write(Arquivo),
        write(''','''),
        write(Exec_met),
        write(''').\n'),
        told,
        /*Abre o novo arquivo para gerar os objetos de teste*/
        inicializar_geracao(Exec_met),
        /*Gera os casos de teste para os objetos de teste criados*/
        abolish(classificacao,1),
        gerar_casos(X,Lista_arq_obj,Arquivo,Gera_val,Exec_met).

listar_arq_objetos(Classe,Lista_arq_obj) :-
        open('objetos.tst',read,Arq_obj),
       read(Arq_obj,X),
        ((X == 'end_of_file')->(Lista_arq_obj = []);
          (consult('objetos.tst'),
           findall(Y,objeto_teste(Classe,Y,_),Lista_arq_obj))),
        close(Arq_obj).

inicializar_geracao(Exec_met) :-
        findall(Id_no,no(Id_no,sim,_,_),Lista_construtores),
        assert(num_sequencia(1)),
        testar_no_inicial(Lista_construtores,Exec_met,inicial),
        abolish(num_sequencia,1).

testar_no_inicial([],_,_).

testar_no_inicial([X|Y],Exec_met,C) :-
        testar_no(X,[],Exec_met,C),
        testar_no_inicial(Y,Exec_met,C).

testar_no(Id_no,L,Exec_met,C) :-
        no(Id_no,_,_,Lista_met),
        testar_metodo(Id_no,Lista_met,Exec_met,L,C).

testar_metodo(_,[],_,_,_).
```

```
testar_metodo(Id_no,[X|_],'Primeiro',L,C) :-
       metodo(X,_,_,Classificacao,_),
       verificar_classificacao(C,Classificacao,Cnova),
       no(Id_no,_,Num_arcos,_),
       append(L,[X],L1),
       ((Num_arcos > 0)->
       (setof(No,arco(Id_no,No),Lista_Prox_Nos));
       (Lista_Prox_Nos = [])),
       proximo_no(Lista_Prox_Nos,L1,'Primeiro',Cnova).

testar_metodo(Id_no,[X|Y],'Todos',L,C) :-
       metodo(X,_,_,Classificacao,_),
       verificar_classificacao(C,Classificacao,Cnova),
       no(Id_no,_,Num_arcos,_),
       append(L,[X],L1),
       ((Num_arcos > 0)->
       (setof(No,arco(Id_no,No),Lista_Prox_Nos));
       (Lista_Prox_Nos = [])),
       proximo_no(Lista_Prox_Nos,L1,'Todos',Cnova),
       testar_metodo(Id_no,Y,'Todos',L,C).


%% Estas regras irao fazer a classificacao de todo o objeto de teste
verificar_classificacao(inicial,_,construtor).
verificar_classificacao(construtor,C_met,C_met).
verificar_classificacao(C,destrutor,C).
verificar_classificacao(novo,_,novo).
verificar_classificacao(recursivo,recursivo,recursivo).
verificar_classificacao(recursivo,C_met,C_met).
verificar_classificacao(redefinido,novo,novo).
verificar_classificacao(redefinido,_, redefinido).

/* se os metodos que compoe um objeto de teste sao todos recursivos, */
/* este objeto nao eh gerado. */
 proximo_no([],L,_,C) :-
      ((C \= 'recursivo')->
      (imprimir(L,C));
       true).

proximo_no([X|Y],L,Exec_met,C) :-
      no(X,_,_,_),
      testar_no(X,L,Exec_met,C),
      ((Y \= [])->
       (proximo_no(Y,L,Exec_met,C));
       true).

imprimir(Lista,C) :-
       num_sequencia(N),
       abolish(num_sequencia,1),
       assert(objeto_num(N,Lista)),
       ((C == 'redefinido')->
       (assert(redefinido(N)));
       true),
       S is N + 1,
       assert(num_sequencia(S)).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Regras para geracao dos casos de teste                                      %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% Significa que este é o primeiro arquivo de objetos de teste para a
%% classe, entao deve ser gerado um caso de teste para cada objeto
%% derivado anteriormente.
gerar_casos(_,[],Arq_obj,Gera_val,_) :-
       /*Abre o arquivo de objetos de teste para ser atualizado com os */
       /* novos casos de teste gerados ou reutilizados*/
       open(Arq_obj,append,Objetos),
       findall(Num,objeto_num(Num,_),L_obj),
       gerar_casos_tst(Gera_val,L_obj,Objetos,[],Arq_obj,[]),
       close(Objetos).

%% Significa que pode existir um arquivo de objetos derivado com a
%% estrategia 'Primeiro' e com a mesma estrategia para geracao dos dados

gerar_casos(_,La_obj,Arq_obj,Gera_val,'Todos') :-
       /*Abre o arquivo de objetos de teste para ser atualizado com os */
       /* novos casos de teste gerados ou reutilizados*/
       open(Arq_obj,append,Objetos),
       /* Verifica quais obj. teste ja possuem casos de teste definidos*/
       consult('casos.tst'),
       findall(Num,objeto_num(Num,_),L_obj_total),
       verificar_ct('Todos',Gera_val,La_obj,[],La_ct,[],L_obj_ct,Objetos),
       subtract(L_obj_total,L_obj_ct,L_obj),
       ((L_obj \= [])->
        (
         /*ainda existem obj. teste que nao possuem casos de teste*/
         gerar_casos_tst(Gera_val,L_obj,Objetos,La_ct,Arq_obj,La_obj));
        (
         /*todos os objetos ja possuem casos de teste definidos */
         append('casos.tst'),
         imprimir_casos_tst(La_ct,Arq_obj,Gera_val),
         told)),
       close(Objetos).

%% Significa que pode existir um arquivo de objetos derivado com a
%% estrategia 'Todos' e com a mesma estr. para geracao dos dados.
gerar_casos(Classe,La_obj,Arq_obj,Gera_val,'Primeiro') :-
       /*Abre o arquivo de objetos de teste para ser atualizado com os */
       /* novos casos de teste gerados ou reutilizados*/
       open(Arq_obj,append,Objetos),

       /* Verifica se todos obj. teste ja tem casos de teste definidos*/
       consult('casos.tst'),
       findall(Y,objeto_teste(Classe,Y,'Todos'),L_todos),
       ((L_todos == [])->
        (/*nao existe arquivo de ot derivado com a estrategia "Todos" */
         verificar_ct('Primeiro',Gera_val,La_obj,[],La_ct,[],L_obj_ct,Objetos)
        );
        (/*existe arquivo de ot derivado com estr. "Todos", verificar */
         /*se existem arquivos de ct derivados para ele com Gera_val */
         procurar_arq_ct(L_todos,L_todos1,Gera_val),
         ((L_todos1 == [])->
          (/* nao existe*/
           subtract(La_obj,L_todos,La_obj1),

verificar_ct('Primeiro',Gera_val,La_obj1,[],La_ct,[],L_obj_ct,Objetos));
           /* existe */
           verificar_ct('Primeiro',Gera_val,L_todos1,[],La_ct,[],L_obj_ct,Objetos)
          ))),
       findall(Num,objeto_num(Num,_),L_obj_total),
       subtract(L_obj_total,L_obj_ct,L_obj),
```

```
        ((L_obj \= [])->
         (
          gerar_casos_tst(Gera_val,L_obj,Objetos,La_ct,Arq_obj,La_obj));
         (append('casos.tst'),
          imprimir_casos_tst(La_ct,Arq_obj,Gera_val),
          told)),
        close(Objetos).

verificar_ct(_,_,[],La_ct,La_ct,L_ini,L_ini,_).

verificar_ct(Exec_met,Gera_val,[Arq_obj|Y],La_ct,La_ct1,L_ini,L_fin,Objetos) :-
        findall(Arq_ct,caso_teste(Arq_obj,Arq_ct,Gera_val),La_ct2),
        ((La_ct2 \= [])->
        (classe(Classe,_,_,_,_),
          objeto_teste(Classe,Arq_obj,Est_Met),
         verificar_estr(Exec_met,Est_Met,Arq_obj,L_fin1,La_ct,La_ctAux,
                        La_ct2,Objetos),
          verificar_ct(Exec_met,Est_Met,Y,La_ctAux,La_ct1,L_fin1,L_fin,Objetos));
        (verificar_ct(Exec_met,Gera_val,Y,La_ct,La_ct1,L_ini,L_fin,Objetos))).


verificar_estr('Primeiro','Todos',Arq_obj,L_fin,La_ct,La_ct1,[Arq_ct|_],Objetos)
:-
        /* Significa que nao eh necessario gerar nenhum caso de teste */
        append(La_ct,[Arq_ct],La_ct1),
        findall(Met,objeto_num(_,Met),L_obj_tst_aux),
        list_to_set(L_obj_tst_aux,L_obj_tst),
        consult(Arq_obj),
        declarar_obj(L_obj_tst,Arq_ct,[],L_fin,Objetos).

verificar_estr('Todos','Primeiro',Arq_obj,L_fin,La_ct,La_ct1,[Arq_ct|_],Objetos)
:-
        append(La_ct,[Arq_ct],La_ct1),
        /* Significa que alguns casos de teste precisam ser criados,*/
        /* mas nao todos */
        findall(Met,objeto_num(_,Met),L_obj_tst),
        consult(Arq_obj),
        findall(Met,objeto_tst(_,Met,_,Arq_ct),L_obj_ct),
        intersection(L_obj_tst,L_obj_ct,L_obj_com_ct),
        list_to_set(L_obj_com_ct,Lobj_com_ct),
        declarar_obj(Lobj_com_ct,Arq_ct,[],L_fin,Objetos).

verificar_estr('Todos','Todos',_,L_fin,La_ct,La_ct1,_,_) :-
         classe(_,_,Template,_,_),
         verifica_template(Template,L_fin,La_ct,La_ct1).

verificar_estr('Primeiro','Todos',_,L_fin,La_ct,La_ct1,_,_) :-
         classe(_,_,Template,_,_),
         verifica_template(Template,L_fin,La_ct,La_ct1).

verifica_template(s,[],[],[]).

verifica_template(n,_,_,_) :- halt.

%% Estas regras declaram fatos na base de dados que permitem saber quais
%% objetos de teste ja possuem casos de teste associados.
declarar_obj([],_,L_ini,L_ini,_).

declarar_obj([X|Y],Arq_ct,L_ini,L_fin,Objetos) :-
        objeto_num(N,X),
        append(L_ini,[N],L_fin1),
```

```
        objeto_tst(_,X,CT,Arq_ct),
        write(Objetos,'objeto_tst('),
        write(Objetos,N),
        write(Objetos,',['),
        imprimir_objetos(Objetos,X),
        write(Objetos,''']'),
        write(Objetos,CT),
        write(Objetos,''','''),
        write(Objetos,Arq_ct),
        write(Objetos,''')'),
        write(Objetos,'.\n'),
        assert(caso_teste(N,X,CT,Arq_ct)),
        declarar_obj(Y,Arq_ct,L_fin1,L_fin,Objetos).

%% Procura arquivos de ct derivados para arquivos de ot, que por sua vez
%% foram derivados com a estrategia de selecao de metodos "Todos",
%% com a mesma estrategia de geracao de valores.
procurar_arq_ct([],[],_).

procurar_arq_ct([Arq_obj|Y],L_todos,Gera_val) :-
      findall(Arq_ct,caso_teste(Arq_obj,Arq_ct,Gera_val),L_aux),
      ((L_aux == [])->
       (procurar_arq_ct(Y,L_todos,Gera_val));
       (L_todos = [Arq_obj])).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Regras para gerar o codigo para casos de teste                          %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
gerar_casos_tst(Gera_val,L_obj,Objetos,La_ct,Arq_obj,La_obj) :-
/*Procura o numero correto para nomear o arquivo de casos de teste*/
        classe(Classe,_,_,_,_),
        verificar_nome(La_obj,[],La_casos),
        length(La_casos,N),
        concat('_ct',N,Nome_aux),
        concat(Nome_aux,'.cc',Extensao),
        concat(Classe,Extensao,Arq_ct),
        /*Acha os objetos de teste gerados e redefinidos, e inicializa*/
        /* a geracao dos casos de teste */
        findall(N1,redefinido(N1),Lista_redefinidos),
        inicializar_geracao(Lista_redefinidos,L_obj,Gera_val,N,Arq_ct,Objetos,
                            La_ct,Arq_obj).

inicializar_geracao([],Lista_obj,Gera_val,N,Arq_ct,Objetos,La_ct,Arq_obj) :-
        /*Nao ha metodos redefinidos,entao casos de teste sao gerados*/
        /* para todos os objetos de teste criados anteriormente*/
        tell(Arq_ct),
        gerar_caso(Lista_obj,Gera_val,Objetos,N,Arq_ct),
        told,
        /*Adiciona o nome dos arquivos de casos de teste*/
        append('casos.tst'),
        imprimir_casos_tst(La_ct,Arq_obj,Gera_val),
        imprimir_casos_tst([Arq_ct],Arq_obj,Gera_val),
        told.

inicializar_geracao(Lista_red,Lista_obj,Gera_val,N,Arq_ct,Objetos,La_ct,Arq_obj)
:-
        tell(user),
        see(user),
        write('Existem objetos de teste redefinidos!\n'),
        write('Deseja (g)erar casos de teste ou (r)eutiliza-los?'),
        get(Resp),
        told,
```

```
        seen,
        /* 103 = g  114 = r*/
        ((Resp == 103)->
        (inicializar_geracao([],Lista_obj,Gera_val,N,Arq_ct,Objetos),
          /*Adiciona o nome dos arquivos de casos de teste*/
          append('casos.tst'),
          imprimir_casos_tst(La_ct,Arq_obj,Gera_val),
          imprimir_casos_tst(Arq_ct,Arq_obj,Gera_val),
          told);
        (
         subtract(Lista_obj,Lista_red,Lista_objetos),
          ((Lista_objetos \= [])->
          (tell(Arq_ct),
            gerar_caso(Lista_objetos,Gera_val,Objetos,N,Arq_ct),
           reutilizar_caso(Lista_red,Objetos,La_ct,La_ct1),
           told,
            /*Adiciona o nome dos arquivos de casos de teste*/
            append('casos.tst'),
            imprimir_casos_tst(La_ct1,Arq_obj,Gera_val),
            imprimir_casos_tst([Arq_ct],Arq_obj,Gera_val),
            told);
           (reutilizar_caso(Lista_red,Objetos,La_ct,La_ct1),
            /*Adiciona o nome dos arquivos de casos de teste*/
            append('casos.tst'),
            imprimir_casos_tst(La_ct1,Arq_obj,Gera_val),
            told)
          )
         )).

%% O parametro Gera_val representa a estrategia para derivacao
%% de valores para os parametros e atributos.
gerar_caso([],_,_,_,_).

gerar_caso([Num|Y],Gera_val,Objetos,N,Arq_casos) :-
   /*Escreve o caso de teste relacionando-o ao objeto de teste*/
        objeto_num(Num,Lista_metodos),
        concat('Caso_teste',Num,Parte_nome),
        concat(Parte_nome,'_',Parte_nome1),
        concat(Parte_nome1,N,Nome),
        write(Objetos,'objeto_tst('),
        write(Objetos,Num),
        write(Objetos,',['),
        imprimir_objetos(Objetos,Lista_metodos),
        write(Objetos,''']'),
        write(Objetos,Nome),
        write(Objetos,''','''),
        write(Objetos,Arq_casos),
        write(Objetos,''')'),
        write(Objetos,'.\n'),
   /*Escreve na base de dados um fato com nome do caso de teste*/
   /*gerado, para ser utilizado na geracao da sequencia de teste.*/
        assert(caso_teste(Num,Lista_metodos,Nome,Arq_casos)),
   /*Gera caso de teste */
        gerar_codigo_inicial(Lista_metodos,Gera_val,Nome),
        gerar_caso(Y,Gera_val,Objetos,N,Arq_casos).

imprimir_objetos(Objetos,[X]) :-
        write(Objetos,''''),
        write(Objetos,X).

imprimir_objetos(Objetos,[X|Y]) :-
```

```
        write(Objetos,''''),
        write(Objetos,X),
        write(Objetos,''',''),
        imprimir_objetos(Objetos,Y).

%% Esta regra gera o codigo que inicializa todos os casos de teste
gerar_codigo_inicial([X|Y],Gera_val,Nome) :-
        write('template <class Tipo>\n'),
        write('void '),
        write(Nome),
        write('(Tipo* cst) { \n'),
        tab(3),
        write('char* met= new char[30];\n'), tab(3),
        write('ofstream arq("Result.txt",ios::app);\n'),tab(3),
        write('if (! arq)\n'),tab(6),
        write('cout << "Erro abrindo arquivo! \\n";\n'),tab(3),
        write('else\n'),tab(6),
        write('cout << "Arquivo criado! \\n"; \n'),
        tab(3),
        write('try {\n'),tab(6),
        write('cst->Testa_Invariante();\n'),
        gerar_codigo(Y,Gera_val,Nome).


gerar_codigo([X],_,Nome) :-
        tab(6),
        write('arq << "'),
        write(Nome),
        write(' OK!\\n";\n'),tab(6),
        write('arq.flush();\n'), tab(6),
        write('cst->Relator("Result.txt");\n'),tab(6),
        write('arq << "\\n";\n'), tab(6),
        write('arq.close();\n'), tab(6),
        write('delete cst;\n'),
        tab(3),
        write('}\n\n'),tab(3),
        write('catch(Erro& er) {\n' ),tab(6),
        write('arq << "'),
        write(Nome),
        write(' \\n";\n'),tab(6),
        write('arq.flush();\n'), tab(6),
        write('cout << "...";\n'), tab(6),
        write('arq << er.msg << "\\n";\n'), tab(6),
        write('arq << "Metodo chamado: " << met << "\\n";\n'),tab(6),
        write('arq.flush();\n'), tab(6),
        write('cst->Relator("Result.txt");\n'),tab(6),
        write('arq << "\\n";\n'), tab(6),
        write('arq.close();\n'),tab(3),
        write('}\n\n'),tab(3),
        write('catch(...) {\n'), tab(6),
        write('arq.close();\n'),tab(6),
        write('}\n'),
        write('}\n\n\n').

 gerar_codigo([X|Y],Gera_val,Nome) :-
        metodo(X,NomeX,_,_,Num_param),
        tab(6),
        write('met = "'),
        write(NomeX), write('('),
        ((Num_param > 0)->
        (findall(P,parametro(P,X,_,_,_),Lista_parametros),
```

```
        buscar_parametros(X,Lista_parametros,[],Lista_p,[],Lista_imp,
        Gera_val),
        imprimir_parametros(Lista_imp),
        write(')";\n'),tab(6),
        write('cst->'),
        write(NomeX),write('('),
        imprimir_parametros(Lista_p),
        write(');\n'));
       (write(')";\n'),tab(6),
        write('cst->'),
        write(NomeX),write('('),
        write(');\n'))),
       tab(6),
       write('cst->Testa_Invariante();\n'),
       gerar_codigo(Y,Gera_val,Nome).

%% Esta regra faz a busca de valores para os parametros do metodo.
%% Ela retorna em L1 a lista dos parametros e em L1_imp a lista dos
%% parametros mas no formato correto para impressao.
buscar_parametros(_,[],L,L,L_Imp,L_Imp,_).

buscar_parametros(M,[X|Y],L,L1,L_imp,L1_imp,Gera_val) :-
                    parametro(X,M,Tipo,_,_),
                    atribui_valor(M,X,Tipo,Valor,Valor_imp,Gera_val),
                    append(L,[Valor],L2),
                    append(L_imp,[Valor_imp],L3),
                    buscar_parametros(M,Y,L2,L1,L3,L1_imp,Gera_val).

atribui_valor(M,P,'intervalo_int',Valor,Valor,'Aleatorio') :-
                    parametro(P,M,'intervalo_int',Vmax,Vmin),
                    Valor is (random(Vmax) - Vmin).

atribui_valor(M,P,'string',Valor,Valor_imp,'Aleatorio') :-
                    parametro(P,M,'string',Lista_val,_),
                    length(Lista_val,N),
                  S is random(N) + 1,
                    nth1(S,Lista_val,V1),
                    concat('"',V1,V2),
                    concat(V2,'"',Valor),
                    concat('\\"',V1,V3),
                    concat(V3,'\\"',Valor_imp).

atribui_valor(M,P,'objeto',Valor,Valor,'Aleatorio') :-
                    parametro(P,M,'objeto',Tipo_objeto,_),
                    concat('Parametro deve ser um objeto do tipo',
                         Tipo_objeto,Msg),
                    Valor = Msg.

atribui_valor(M,P,'ponteiro',Valor,Valor,'Aleatorio') :-
                    parametro(P,M,'ponteiro',Tipo_ptr,_),
                    concat('Parametro deve ser um ponteiro para ',
                         Tipo_ptr,Msg),
                    Valor = Msg.

atribui_valor(M,P,'conjunto',Valor,Valor,'Aleatorio') :-
                    parametro(P,M,'conjunto',Lista_val,_),
                    length(Lista_val,N),
                  S is random(N) + 1,
                    nth1(S,Lista_val,Valor).

imprimir_parametros([X]) :- write(X).
```

```
imprimir_parametros([X|Y]) :-
        write(X), write(','),
        imprimir_parametros(Y).

%% Regras para reutilizacao de casos de teste
reutilizar_caso(Lista_redefinidos,Objetos,La_ct,La_ct1) :-
        classe(_,_,_,Super,_),
        findall(Arq,objeto_teste(Super,Arq,_),Lista_arq),
        buscar_caso(Lista_redefinidos,Lista_arq,Objetos,La_ct,La_ct1).

buscar_caso([],_,_,La_ct,La_ct).

buscar_caso([O1|O2],Lista_arq,Objetos,La_ct,La_ct1) :-
        objeto_num(O1,Metodos),
        pesquisar_arq(O1,Lista_arq,Metodos,Objetos,La_ct,La_ct2),
        buscar_caso(O2,Lista_arq,Objetos,La_ct2,La_ct1).

pesquisar_arq(_,[],_,_,_,_).

pesquisar_arq(Num,[A1|A2],Metodos,Objetos,La_ct,La_ct2) :-
        consult(A1),
        findall(N,objeto_tst(N,Metodos,_,_),Lista),
        ((Lista \=[])->
        (objeto_tst(N,Metodos,CT,Arq_ct),
          write(Objetos,objeto_tst(Num,Metodos,CT,Arq_ct)),
         write(Objetos,'.\n'),
         assert(caso_teste(Num,Metodos,CT,Arq_ct)),
          append(La_ct,[Arq_ct],La_ct1),
          list_to_set(La_ct1,La_ct2),
          abolish(objeto_tst,4),
         pesquisar_arq(_,[],_,_,_,La_ct2));
        (abolish(objeto_tst,4),
          pesquisar_arq(Num,A2,Metodos,Objetos,La_ct,La_ct2))).

verificar_nome([],L1,Lista_arq) :-
   /*Remove possiveis duplicatas */
      list_to_set(L1,Lista_arq).

verificar_nome([Arq_obj|Y],L,L1) :-
   /*Procura todos os arq. de casos de teste para um determinado arq*/
   /* de objetos de teste, sem duplicatas */
       setof(Nome_arq,caso_teste(Arq_obj,Nome_arq,_),Lista_arq),
       append(L,Lista_arq,L2),
       verificar_nome(Y,L2,L1).

%% Regras para adicionar ao arquivo 'casos.tst' os arquivos de casos de
%% teste que testam determinado arquivo de objetos de teste
imprimir_casos_tst([],_,_).

imprimir_casos_tst([Arq_ct|Y],Arq_obj,Gera_val) :-
      write('caso_teste('''),
      write(Arq_obj),
      write(''','''),
      write(Arq_ct),
      write(''','''),
      write(Gera_val),
      write(''').\n'),
      imprimir_casos_tst(Y,Arq_obj,Gera_val).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%% Regras para geracao da sequencia de teste                              %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
gerar_sequencia_tst(Sistema,Gera_val) :-
         classe(X,_,Template,_,_),
         findall(Nome,caso_teste(_,_,Nome,_),Lista_casos_aux),
         list_to_set(Lista_casos_aux,Lista_casos),
         findall(Nome_arq,caso_teste(_,_,_,Nome_arq),Lista_arq_aux),
         list_to_set(Lista_arq_aux,Lista_arq),
    /*Procura o numero correto para nomear o arquivo de objetos*/
         listar_arq_sequencias(Lista_arq_seq,X),
         length(Lista_arq_seq,N),
         concat('_st',N,Nome_aux),
         concat(Nome_aux,'.cc',Extensao),
         concat(X,Extensao,Arq_seq),
    /*Adiciona ao arquivo o nome do novo arquivo de sequencias de teste*/
         append('sequencias.tst'),
         write('sequencia_teste('''),
         write(X),
         write(''','''),
         write(Arq_seq),
         write(''').\n'),
         told,
    /*Abre o novo arquivo para gerar os objetos de teste*/
         tell(Arq_seq),
         inicializa_seq(Sistema,Lista_arq),
         assert(sequencia(1)),
         gerar_seq(Template,Lista_casos,X,Gera_val),
         finaliza_seq,
         told.

listar_arq_sequencias(Lista_arq_seq,Nome_classe) :-
         open('sequencias.tst',read,Arq_seq),
         read(Arq_seq,X),
         ((X == 'end_of_file')->
          (Lista_arq_seq = []);
         (consult('sequencias.tst'),
          findall(Y,sequencia_teste(Nome_classe,Y),Lista_arq_seq))),
         close(Arq_seq).

inicializa_seq(Sistema,Lista_arq) :-
         write('#include <fstream.h>\n'),
         write('#include <iostream.h>\n'),
         write('#define TESTE\n'),
         write('#include "'),
         write(Sistema),
         write('"'),write('\n'),
         classe(_,_,_,_,Arq),
         adiciona_arq(Arq),nl,
         adiciona_arq_ct(Lista_arq),nl,
         nl,
         write('void main() {\n').

adiciona_arq([]).
adiciona_arq([X|Y]) :-
         write('#include<'),
         write(X),
         write('>\n\n'),
         adiciona_arq(Y).

adiciona_arq_ct([]).
```

```
adiciona_arq_ct([X|Y]) :-
         write('#include "'),
         write(X),
         write('"\n\n'),
         adiciona_arq_ct(Y).

gerar_seq(_,[],_,_).

%% Geracao do codigo da sequencia para classes comuns
gerar_seq(n,[X|Y],Nome_classe,Gera_val) :-
         tab(3),
         caso_teste(_,[Met1|_],X,_),
         metodo(Met1,Nome_metodo,_,_,Num_param),
        sequencia(N),
         write(Nome_classe),
         write('* obj'),
         write(N),
         write(' = new '),
         write(Nome_metodo),
         ((Num_param > 0)->
          (write('('),
           findall(P,parametro(P,Met1,_,_,_),Lista_parametros),
           buscar_parametros(Met1,Lista_parametros,[],Lista_p,[],_,Gera_val),
           imprimir_parametros(Lista_p),
           write(')')
          );
          true),
         write(';\n'),
         tab(3),
         write(X),
         write('(obj'),
         write(N),
         write(');\n'),
         abolish(sequencia,1),
         S is N + 1,
         assert(sequencia(S)),
         gerar_seq(n,Y,Nome_classe,Gera_val).

%% Geracao do codigo da sequencia para classes templates
gerar_seq(s,[X|Y],Nome_classe,Gera_val) :-
        parametros_template([Z|W]),
         caso_teste(_,[Met1|_],X,_),
         metodo(Met1,Nome_metodo,_,_,Num_param),
         tab(3),
         sequencia(N),
         write(Nome_classe),
         write('<'),
         write(Z),
         escreve_parametros(W),
         write('>* obj'),
         write(N),
         write(' = new '),
         write(Nome_metodo),
         write('<'),
         write(Z),
         escreve_parametros(W),
         write('>'),
         ((Num_param > 0)->
          (write('('),
           findall(P,parametro(P,Met1,_,_,_),Lista_parametros),
           buscar_parametros(Met1,Lista_parametros,[],Lista_p,[],_,Gera_val),
```

```
            imprimir_parametros(Lista_p),
            write(')')
           );
           true),
          write(';\n'),
          tab(3),
          write(X),
          write('(obj'),
          write(N),
          write(');\n'),
          abolish(sequencia,1),
          S is N + 1,
          assert(sequencia(S)),
          gerar_seq(s,Y,Nome_classe,Gera_val).

escreve_parametros([]).

escreve_parametros([X|Y]) :-
        write(','),
         write(X),
         escreve_parametros(Y).

finaliza_seq :- write('}').
```

Apêndice C

Exemplos de Uso

Neste apêndice são apresentadas as definições das classes utilizadas como exemplos de uso da metodologia proposta neste trabalho.

C.1 Classes Template e Aninhadas

Uma classe *template* não pode ser instanciada diretamente, ela necessita de algum parâmetro. Considere a classe *Queue* ilustrada abaixo, ela é uma classe *template* e possui uma classe aninhada.

```
template <class Type>
class Queue {
 public:
   Queue() {front = back = 0;}
   ~Queue();
   Type remove();
   void add(const Type&);
   int is_empty() {
      return front == 0 ? 1 : 0; }
 private:
   class QueueItem {
      Type item;
      QueueItem* next;
    public:
      QueueItem(Type t) : item(t), next(0) {}
      ~QueueItem() {}
      void set_item(Type i) {item = i;}
      void set_next(QueueItem* q) {next = q;}
      Type retorna_item() {return item;}
      QueueItem* retorna_next() {return next;}
```

```
    };
   QueueItem* front;
   QueueItem* back;
};
```

Para testar classes aninhadas o procedimento foi primeiro testar a classe mais interna e depois a mais externa. Primeiramente, a classe *QueueItem* foi instrumentada com os mecanismos de teste embutidos e se tornou autotestável. Foi construída uma especificação de teste para ela e gerados casos de teste. Após a aplicação dos testes, os resultados foram analisados e verificou-se que nenhum erro havia sido detectado durante os testes.

O passo seguinte foi instrumentar a classe *Queue* e torná-la autotestável. A classe *Queue* instrumentada está ilustrada a seguir.

```
template <class Type>
class Queue: public Autoteste {
public:
   Queue() {front = back = 0;}
   ~Queue();
   Type remove();
   void add(const Type&);
   int is_empty() {
      return front == 0 ? 1 : 0; }
   void Testa_Invariante() ;
   #ifdef TESTE
   void Relator(char* arquivo);
   #endif
private:
   class QueueItem: public Autoteste {
      Type item;
      QueueItem* next;
   public:
      QueueItem(Type t) : item(t), next(0) {}
      ~QueueItem() {}
      void set_item(Type i) {item = i;}
      void set_next(QueueItem* q) {next = q;}
      Type retorna_item() {return item;}
      QueueItem* retorna_next() {return next;}
      #ifdef TESTE
        void Relator(char* arquivo) {
           ofstream resultados(arquivo,ios::app);
           if (!resultados) {
              cout << "Erro abrindo arquivo!" << endl;
              exit;
           };
```

```
            resultados << "item = "<< item << "\n";
            resultados << "next = " << next << "\n";
        }
      #endif
   };
   QueueItem* front;
   QueueItem* back;
};
```

Foi construída a especificação de teste para ela. Note, porém, que os métodos da classe *QueueItem* não foram considerados no modelo de fluxo de transação da classe *Queue*, pois ela é definida como um atributo privado à classe. Assim, o modelo de fluxo de transação gerado para a classe *Queue* não é influenciado por ela possuir uma classe aninhada. A especificação de teste gerada para a classe *Queue* está ilustrada na figura C.1.

```
classe('Queue',n,s,_,[]).

parametros_template(['float']).

atributo('front',ponteiro,'QueueItem',_).
atributo('back',ponteiro,'QueueItem',_).

metodo(m1,'Queue',_,construtor,0).
metodo(m2,'~Queue',_,destrutor,0).
metodo(m3,'remove','float',novo,0).
metodo(m4,'add','void',novo,1).
metodo(m5,'is_empty','int',novo,0).

parametro('val',m4,conjunto,[1.2,1.3,1.4,1.5],_).

no(n1,sim,1,[m1]).
no(n2,nao,0,[m2]).
no(n3,nao,2,[m4]).
no(n4,nao,1,[m3]).

arco(n1,n3).
arco(n3,n2).
arco(n3,n4).
arco(n4,n2).
```

Figura C.1: Especificação de teste da classe *Queue*.

Na especificação, a cláusula "parametros_template" descreve o parâmetro necessário para instanciar um objeto da classe *Queue*. Assim, na geração da seqüência de testes quando são instanciados objetos da classe sob teste, esses parâmetros são utilizados pelo *driver* genérico. A figura C.2 ilustra a seqüência de teste gerada para a classe *Queue*. Os casos de teste gerados para uma classe *template* não se diferenciam daqueles gerados

paras outras classes. Note também que para testar a classe *Queue* é necessário que a classe *QueueItem* já seja uma classe autotestável para que caso um erro seja detectado os estados internos de ambas as classes possam ser reportados.

```
#include <fstream.h>
#include <iostream.h>
#define TESTE
#include "Queue.cc"

#include "Queue_ct0.cc"

void main() {
   Queue<float>* obj1 = new Queue<float>;
   Caso_teste1_0(obj1);
   Queue<float>* obj2 = new Queue<float>;
   Caso_teste2_0(obj2);
}
```

Figura C.2: Seqüência de teste gerada para a classe *Queue*.

C.2 Classe Abstrata

As classes abstratas não podem ser instanciadas diretamente e, portanto, não podem ser testadas diretamente mas somente através de suas classes derivadas concretas. A classe *Abstrata*, apresentada a seguir, foi utilizada para exemplificar a preparação dos testes.

```
class Abstrata{
public:
  Abstrata() {}
  ~Abstrata() {}
  virtual void Inverta() = 0;
  virtual void Duplique() = 0;
};
```

Na especificação de teste da classe deve ser indicado que a mesma é abstrata. A especificação da classe *Abstrata* está ilustrada na figura C.3. Para ela o *Driver* Genérico gera somente os objetos e casos de teste que poderão ser reutilizados no teste de classes derivadas concretas de *Abstrata*. Como os casos de teste não poderão ser aplicados sobre a classe abstrata, o *driver* Genérico não gera a seqüência de teste para ela.

```
classe('Abstrata',s,n,_,[]).

parametros_template([]).

metodo(m1,'Abstrata',_,construtor,0).
metodo(m2,'~Abstrata',_,destrutor,0).
metodo(m3,'Inverta','void',novo,0).
metodo(m4,'Duplique','void',novo,0).

no(n1,sim,2,[m1]).
no(n2,nao,0,[m2]).
no(n3,nao,2,[m3,m4]).

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

Figura C.3: Especificação de teste para a classe *Abstrata.*

C.3 Classes Base

Como exemplos de classe base foram utilizadas as classes *cartao* e *list*, ilustradas nas figuras C.4 e C.5.

```
class cartao {
protected:
      int banco_emissor;
      int identificador;
      char* proprietario;
      float temp;
public:
      cartao() {
            banco_emissor = identificador = 0;
            temp = 0.0;
            proprietario = new char[50];}
      cartao(int banco,int id,const char* nome);
      ~cartao() {delete proprietario;}

      void Atualiza_dados(int banco, int id,const char* nome);
      void Consulta_dados();
      void Operacoes_disponiveis(float n);
      void Consulta_limite();
}
```

Figura C.4: Classe *cartao.*

```
class list {
 protected:
      listMember* _head;
      listMember* _tail;
 public:
      list()
            { _head = _tail = 0; }
      virtual ~list() {};
      listMember *Head()
            { return _head; }
      listMember *Tail()
            { return _tail; }
      list& Append(listMember *);
      list& Prepend(listMember *);
      list& Remove(listMember *);
      list& CleanUpMarked();
      int   Length();
      listMember *GetNthMember(int n);
      list& Push(listMember *l)
            { return Append(l); }
      listMember *Pop()
            { listMember *l = _tail;
              Remove(_tail);
              return l;
            }
      listMember *Top()
            { cout << "Metodo Top-list" << endl;
              return _tail; }
};
```

Figura C.5: Classe *list*.

Estas classes não possuem características especiais, não são *templates* ou abstratas. As especificações de teste construídas para elas estão ilustradas nas figuras C.6 e C.7 respectivamente.

A classe *cartao* possui métodos com parâmetros que são do tipo *string* e *float*. Na especificação foram definidos para esses tipos um conjunto dos valores que tais parâmetros podem assumir durante os testes. Por exemplo, o parâmetro *nome* pode ter os valores '*Joao*', '*Maria*' ou '*Jose*'.

A classe *list* possui parâmetros e atributos que são ponteiros para a classe *listMember*. Os casos de teste gerados para a classe *list* conterão mensagens que o usuário deverá substituir por ponteiros para a classe correta, como ilustrados na figura C.8.

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

parametros_template([]).

atributo('banco_emissor',intervalo_int,1000,1).
atributo('identificador',intervalo_int,100000,1).
atributo('proprietario',string,['Joao','Maria','Jose'],_).
atributo('temp',conjunto,[1.9,1.3,1.1],_).

metodo(m1,'cartao',_,construtor,0).
metodo(m2,'cartao',_,construtor,3).
metodo(m3,'~cartao',_,destrutor,0).
metodo(m4,'Atualiza_dados','void',novo,3).
metodo(m5,'Consulta_dados','void',novo,0).
metodo(m6,'Operacoes_disponiveis','void',novo,1).
metodo(m7,'Consulta_limite','void',novo,0).

parametro('banco',m2,intervalo_int,1000,1).
parametro('id',m2,intervalo_int,10000,1).
parametro('proprietario',m2,string,['Joao','Maria','Jose'],_).
parametro('banco',m4,intervalo_int,1000,1).
parametro('id',m4,intervalo_int,10000,1).
parametro('proprietario',m4,string,['Joao','Maria','Jose'],_).
parametro('n',m6,conjunto,[1.9,1.3,1.1],_).

no(n1,sim,1,[m1]).
no(n2,sim,1,[m2]).
no(n3,nao,2,[m4]).
no(n4,nao,1,[m5]).
no(n5,nao,2,[m6,m7]).
no(n6,nao,0,[m3]).

arco(n1,n3).
arco(n2,n5).
arco(n3,n4).
arco(n3,n5).
arco(n4,n6).
arco(n5,n6).
arco(n5,n4).
```

Figura C.6: Especificação de teste da classe *cartao*.

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

parametros_template([]).

atributo('_head',ponteiro,'listMember*',_).
atributo('_tail',ponteiro,'listMember*',_).

metodo(m1,'list',_,construtor,0).
metodo(m2,'~list',_,destrutor,0).
metodo(m3,'Head','listMember*',novo,0).
metodo(m4,'Tail','listMember*',novo,0).
metodo(m5,'Append','list&',novo,1).
metodo(m6,'Prepend','list&',novo,1).
metodo(m7,'Remove','list&',novo,1).
metodo(m8,'CleanUpMarked','list&',novo,0).
metodo(m9,'Length','int',novo,0).
metodo(m10,'GetNthMember','listMember*',novo,1).
metodo(m11,'Push','list&',novo,1).
metodo(m12,'Pop','listMember*',novo,0).
metodo(m13,'Top','listMember*',novo,0).

parametro('l',m5,ponteiro,'listMember',_).
parametro('l',m6,ponteiro,'listMember',_).
parametro('l',m7,ponteiro,'listMember',_).
parametro('n',m10,intervalo_int,100,1).
parametro('l',m11,ponteiro,'listMember',_).

no(n1,sim,6,[m1]).
no(n2,nao,0,[m2]).
no(n3,nao,3,[m5]).
no(n4,nao,3,[m6]).
no(n5,nao,3,[m11]).
no(n6,nao,2,[m7]).
no(n7,nao,2,[m8]).
no(n8,nao,2,[m12]).
no(n9,nao,1,[m3,m4,m10,m9]).
no(n10,nao,1,[m9,m13]).
no(n11,nao,1,[m13]).

arco(n1,n2).
arco(n1,n3).
arco(n1,n4).
arco(n1,n5).
arco(n1,n9).
arco(n1,n11).
arco(n3,n6).
arco(n3,n7).
arco(n3,n9).
arco(n4,n6).
arco(n4,n7).
arco(n4,n9).
arco(n5,n7).
arco(n5,n8).
arco(n5,n10).
```

```
arco(n6,n2).
arco(n6,n9).
arco(n7,n2).
arco(n7,n9).
arco(n8,n2).
arco(n8,n10).
arco(n9,n2).
arco(n10,n2).
arco(n11,n2).
```

Figura C.7: Especificação de teste da classe *list*.

```
template <class Tipo>
void Caso_teste3_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 = "Append(Parametro deve ser um ponteiro para listMember)";
      cst->Append(Parametro deve ser um ponteiro para listMember);
      cst->Testa_Invariante();
      met = "Remove(Parametro deve ser um ponteiro para listMember)";
      cst->Remove(Parametro deve ser um ponteiro para listMember);
      cst->Testa_Invariante();
      arq.close();
      delete cst;
   }

   catch(char* c) {
      arq << "Caso_teste3_0 \n";
      arq.flush();
      cout << "...";
      arq << c << "\n";
      arq << "Metodo chamado: " << met << "\n";
      arq.flush();
      cst->Relator("Result.txt");
      arq.close();
   }
   catch(...) {
      arq.close();
      }
}
```

Figura C.8: Exemplo de caso de teste para classe *list*.

C.4 Classes Derivadas

As classes derivadas devem ser testadas após os testes de suas classes base. Como exemplo de classes derivadas foram utilizadas as classes *Cartao_Mul* e *EventHandlerList*, classes derivadas das classes *Cartao* e *List* respectivamente.

A classe *Cartao_Mul* está ilustrada na figura C.9. O modelo de fluxo de transação da classe derivada pode ser construído a partir do modelo de sua classe base, como indicado no capítulo 5, mas no caso desta classe derivada não houve alteração do modelo. Note que ela possui um atributo a mais que a classe *cartao* e redefine os métodos *Consulta_dados* e *Operacoes_disponiveis*. A especificação de teste construída para a classe está ilustrada na figura C.10. Na primeira cláusula da especificação está indicado o nome da classe base e na definição dos métodos, a classificação de acordo com as modificações que eles sofreram em relação a classe base.

```
class cartao_mul: public cartao{
       int codigo;
 public:
      cartao_mul():cartao() {}
      cartao_mul(int banco,int id,const char* nome,int cod);
      ~cartao_mul() {}
      void Consulta_dados();
      void Operacoes_disponiveis(float n);
};
```

Figura C.9: Classe *Cartao_Mul*.

Estão ilustrados nas figuras C.11 e C.12 um caso de teste e a seqüência de testes gerada para a classe respectivamente. Não foram definidos novos casos de teste para a classe, todos foram reutilizados da classe base *Cartao*. Na seqüência de teste são instanciados objetos da classe *Cartao_Mul* utilizando os dois construtores definidos, com e sem parâmetros.

```
classe('cartao_mul',n,n,'cartao',[]).

parametros_template([]).

atributo('banco_emissor',intervalo_int,1000,1).
atributo('identificador',intervalo_int,100000,1).
atributo('proprietario',string,['Joao','Maria','Jose'],_).
atributo('temp',conjunto,[1.9,1.3,1.1],_).
atributo('codigo',intervalo_int,9999,1).

metodo(m1,'cartao_mul',_,construtor,0).
metodo(m2,'cartao_mul',_,construtor,4).
metodo(m3,'~cartao_mul',_,destrutor,0).
metodo(m4,'Atualiza_dados','void',recursivo,3).
metodo(m5,'Consulta_dados','void',redefinido,0).
metodo(m6,'Operacoes_disponiveis','void',redefinido,1).
metodo(m7,'Consulta_limite','void',recursivo,0).

parametro('banco',m2,intervalo_int,1000,1).
parametro('id',m2,intervalo_int,10000,1).
parametro('nome',m2,string,['Joao','Maria','Jose'],_).
parametro('cod',m2,intervalo_int,9999,1).
parametro('banco',m4,intervalo_int,1000,1).
parametro('id',m4,intervalo_int,10000,1).
parametro('proprietario',m4,string,['Joao','Maria','Jose'],_).
parametro('n',m6,conjunto,[1.9,1.3,1.1],_).

no(n1,sim,1,[m1]).
no(n2,sim,1,[m2]).
no(n3,nao,2,[m4]).
no(n4,nao,1,[m5]).
no(n5,nao,2,[m6,m7]).
no(n6,nao,0,[m3]).

arco(n1,n3).
arco(n2,n5).
arco(n3,n4).
arco(n3,n5).
arco(n4,n6).
arco(n5,n6).
arco(n5,n4).
```

Figura C.10: Especificação de teste da classe *Cartao_Mul*.

```
template <class Tipo>
void Caso_teste4_1(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 = "Atualiza_dados(411,790,\"Jose\")";
      cst->Atualiza_dados(411,790,"Jose");
      cst->Testa_Invariante();
      met = "Consulta_limite()";
      cst->Consulta_limite();
      cst->Testa_Invariante();
      met = "Consulta_dados()";
      cst->Consulta_dados();
      cst->Testa_Invariante();
      arq << "Caso_teste4_1 OK!\n";
      arq.flush();
      cst->Relator("Result.txt");
      arq << "\n";
      arq.close();
      delete cst;
   }

   catch(Erro& er) {
      arq << "Caso_teste4_1 \n";
      arq.flush();
      cout << "...";
      arq << er.msg << "\n";
      arq << "Metodo chamado: " << met << "\n";
      arq.flush();
      cst->Relator("Result.txt");
      arq << "\n";
      arq.close();
   }

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

Figura C.11: Caso de teste reutilizado pela classe *Cartao_Mul*.

```
#include <fstream.h>
#include <iostream.h>
#define TESTE
#include "cartao.cc"
#include "cartao_mul.cc"
#include "cartao_ct0.cc"
#include "cartao_ct1.cc"

void main() {
   cartao_mul* obj1 = new cartao_mul;
   Caso_teste1_0(obj1);
   cartao_mul* obj2 = new cartao_mul;
   Caso_teste2_0(obj2);
   cartao_mul* obj3 = new cartao_mul;
   Caso_teste3_0(obj3);
   cartao_mul* obj4 = new cartao_mul;
   Caso_teste4_1(obj4);
   cartao_mul* obj5 = new cartao_mul(848,1600,"Jose",7876);
   Caso_teste4_0(obj5);
   cartao_mul* obj6 = new cartao_mul(847,6626,"Maria",5358);
   Caso_teste5_0(obj6);
   cartao_mul* obj7 = new cartao_mul(393,9325,"Joao",3866);
   Caso_teste8_1(obj7);
}
```

Figura C.12: Seqüência de teste para a classe *Cartao_Mul*.

A próxima classe exemplo é a *EventHandlerList* classe derivada da classe *list*. Ela está ilustrada na figura C.13. Esta classe ao ser derivada teve os tipos de retorno de quase todos os seus métodos alterados para um outro objeto. Assim, de acordo com a abordagem incremental hierárquica, tais métodos são considerados novos. Apenas dois não foram alterados e nenhum método novo foi criado. A especificação de teste construída para a classe está ilustrada na figura C.14 e a seqüência de teste, na figura C.15.

```
class eventHandlerList: public list {
 public:
  eventHandlerList() : list() {}
  ~eventHandlerList() {}

  eventHandler* Tail()
    {return (eventHandler*) _tail; }

  eventHandler* Head()
    {return (eventHandler*) _head; }

  eventHandler* Pop()
    {return (eventHandler*) list::Pop();}

  eventHandler* Top()
    {   cout << "metodo Top-EventHandler" << endl;
      return (eventHandler*) _tail;}

  eventHandler* GetNthMember(int n)
    {
        return (eventHandler*)list::GetNthMember(n);}

  eventHandlerList& Append(eventHandlerMember* l)
    {
      list::Append(l);
      return *this;
    }
  eventHandlerList& Prepend(eventHandlerMember* l)
    {
      list::Prepend(l);
      return *this;
    }
  eventHandlerList& Remove(eventHandlerMember* l)
    {
      list::Remove(l);
      return* this;
    }
  eventHandlerList& Push(eventHandlerMember* l)
    {
      list::Push(l);
      return *this;
    }
 };
```

Figura C.13: Classe *EventHandlerList.*

```
classe('eventHandlerList',n,n,'list',[]).

parametros_template([]).

atributo('_head',ponteiro,'listMember*',_).
atributo('_tail',ponteiro,'listMember*',_).

metodo(m1,'eventHandlerList',_,construtor,0).
metodo(m2,'~eventHandlerList',_,destrutor,0).
metodo(m3,'Head','eventHandler*',novo,0).
metodo(m4,'Tail','eventHandler*',novo,0).
metodo(m5,'Append','eventHandlerList&',novo,1).
metodo(m6,'Prepend','eventHandlerList&',novo,1).
metodo(m7,'Remove','eventHandlerList&',novo,1).
metodo(m8,'CleanUpMarked','list&',recursivo,0).
metodo(m9,'Length','int',recursivo,0).
metodo(m10,'GetNthMember','eventHandler*',novo,1).
metodo(m11,'Push','eventHandlerList&',novo,1).
metodo(m12,'Pop','eventHandler*',novo,0).
metodo(m13,'Top','eventHandler*',novo,0).

parametro('l',m5,ponteiro,'eventHandlerMember*',_).
parametro('l',m6,ponteiro,'eventHandlerMember*',_).
parametro('l',m7,ponteiro,'eventHandlerMember*',_).
parametro('n',m10,intervalo_int,100,1).
parametro('l',m11,ponteiro,'eventHandlerMember*',_).

no(n1,sim,6,[m1]).
no(n2,nao,0,[m2]).
no(n3,nao,3,[m5]).
no(n4,nao,3,[m6]).
no(n5,nao,3,[m11]).
no(n6,nao,2,[m7]).
no(n7,nao,2,[m8]).
no(n8,nao,2,[m12]).
no(n9,nao,1,[m3,m4,m10,m9]).
no(n10,nao,1,[m9,m13]).
no(n11,nao,1,[m13]).

arco(n1,n2).
arco(n1,n3).
arco(n1,n4).
arco(n1,n5).
arco(n1,n9).
arco(n1,n11).
. . .
arco(n10,n2).
arco(n11,n2).
```

Figura C.14: Especificação de teste da classe *EventHandlerList*.

```
// Arquivo instrumentado
#include <fstream.h>
#include <iostream.h>
#include "stub1.cc"
#include "ehlist.cc"
#include "eventHandlerList_ct0.cc"

#define TESTE

void main() {
   eventHandlerList* obj1 = new eventHandlerList;
   Caso_teste1_0(obj1);
   eventHandlerList* obj2 = new eventHandlerList;
   Caso_teste2_0(obj2);
   eventHandlerList* obj3 = new eventHandlerList;

   . . .

   eventHandlerList* obj39 = new eventHandlerList;
   Caso_teste39_0(obj39);
   eventHandlerList* obj40 = new eventHandlerList;
   Caso_teste40_0(obj40);
   eventHandlerList* obj41 = new eventHandlerList;
   Caso_teste41_0(obj41);
   eventHandlerList* obj42 = new eventHandlerList;
   Caso_teste42_0(obj42);
   eventHandlerList* obj43 = new eventHandlerList;
   Caso_teste43_0(obj43);
}
```

Figura C.15: Seqüência de teste para classe *EventHandlerList*.

C.5 Sumário dos Exemplos

Os resultados dos experimentos realizados estão resumidos na tabela 3. Nela estão indicados quantos casos de teste foram aplicados (incluindo os casos de teste reutilizados) e a quantidade de casos de teste que foram reutilizados para cada tipo de estratégia de teste.

Classe	Tipo de Classe	Estratégia Teste	Casos de Teste	Testes reutilizados
Abstrata	classe abstrata	Primeiro	1	0
Abstrata	classe abstrata	Todos	1	0
Cartão	classe base	Primeiro	5	0
Cartão	classe base	Todos	9	5
Cartão_Mul	classe derivada	Primeiro	5	5
Cartão_Mul	classe derivada	Todos	7	7
Queue	Aninhada/template	Primeiro	2	0
Queue	Aninhada/template	Todos	2	2
List	classe base	Primeiro	18	0
List	classe base	Todos	44	18
EventHandlerList	classe derivada de list	Primeiro	17	0
EventHandlerList	classe derivada de list	Todos	43	17

Tabela 3: Exemplos de uso de classes autotestáveis.

Apêndice D

Prolog

Prolog é uma linguagem de programação lógica utilizada para resolver problemas que envolvam objetos e relacionamentos entre estes objetos. Por exemplo, a frase "João é pai de José" estabelece o relacionamento "ser pai de" entre os objetos "João" e "José". Um programa em Prolog é um modo de representar o conhecimento, ele consiste em:

- declarar fatos sobre os objetos e seus relacionamentos;
- declarar regras sobre os objetos e seus relacionamentos, e
- questionar sobre os objetos e seus relacionamentos.

Este anexo descreve os conceitos básicos da linguagem Prolog. As definições utilizadas foram apresentadas em [CM87] e [Rog87].

D.1 Fatos

Um fato descreve um relacionamento envolvendo um ou mais objetos. Por exemplo, "Maria ama Pedro" é um fato envolvendo os objetos "Maria" e "Pedro" através do relacionamento "ama". Os objetos de um fato são chamados argumentos e o relacionamento entre eles é chamado predicado. Em Prolog este fato é descrito com a seguinte sintaxe:

```
ama(maria,pedro).
```

Em Prolog "maria" e "pedro" são escritos em letras minúsculas pois representam constantes. É importante que a ordem dos objetos na declaração dos fatos seja mantida, pois o significado do que está sendo declarado pode ser alterado. Por exemplo, se o fato anterior fosse alterado para "ama(pedro,maria)", o significado seria que Pedro ama Maria, ao invés de Maria ama Pedro.

Em Prolog, um conjunto de fatos é chamado uma base de dados.

D.2 Regras

É comum descrever relacionamentos entre objetos utilizando regras. Regras possuem uma cabeça e um corpo conectados pelo símbolo ":-". Em Prolog, regras são utilizadas para dizer que um fato depende de outros fatos. A diferença entre um fato e uma regra é que um fato é sempre incondicionalmente verdade e uma regra é verdadeira se alguma condição é satisfeita. Por exemplo, João e Maria são irmãos se o pai de João é o pai de Maria. O exemplo acima, de uma forma genérica, seria descrito pela regra:

```
irmao(X,Y) :- pai(W,X), pai(W,Y).
```

Neste exemplo, X, Y e W são variáveis. Uma variável em Prolog é qualquer nome que se inicie com uma letra maiúscula ou um caracter de sublinha "_", não existe noção de tipo. A variável anônima, representada somente pelo caracter sublinha "_", é uma variável especial e indica que o argumento que ela representa não é interessante para o predicado. Por exemplo, considere a regra:

```
tem_filho (X) :- pai(X,Y).
```

Esta regra define o relacionamento "tem_filho", indicando que X tem um filho se X é pai de algum Y. Porém, para este predicado o nome do filho não importa. Nesse caso a regra poderia ser escrita como:

```
tem_filho(X) :- pai(X, _ ).
```

Deve-se notar que o escopo de uma variável é a própria regra ou questão onde ela é utilizada. A resolução de uma regra depende da resolução dos fatos e/ou outras regras que a compõem. Geralmente um predicado em Prolog será definido por regras e/ou fatos, que são chamados cláusulas para o predicado.

D.3 Questões

Questões podem ser feitas sobre os fatos ou regras de uma base de dados. Uma questão em Prolog é parecida com um fato, mas possui um símbolo "?" a frente dele. Por exemplo, considere a questão:

```
?- ama(maria,pedro).
```

A interpretação é "Maria ama Pedro?". Para responder a esta questão Prolog utiliza um processo chamado casamento de padrões (do inglês, *pattern matching*). Ele pesquisa sua base de dados procurando fatos que casem com o fato em questão, ou seja, tenham o mesmo predicado e os mesmos argumentos correspondentes. Caso encontre um fato que corresponda uma resposta positiva é dada, caso contrário, ele fornece uma resposta negativa.

Variáveis podem ser utilizadas para responder questões do tipo "quem Maria ama?", que ficaria:

```
?-ama(maria,X).
```

Como resposta Prolog pesquisa sua base de dados até encontrar um fato ou regra cujo predicado e primeiro argumento se casem com os fornecidos pela questão, então X é instanciado com o valor do segundo argumento do fato ou regra encontrado. Para que todas as respostas possíveis para X sejam fornecidas é necessário utilizar o comando ";".

Cada fato que Prolog deve procurar em sua base de dados para responder uma questão é um objetivo a ser satisfeito. Uma questão pode representar relacionamentos complexos, que possuam mais de um objetivo. Para descrever tais questões é utilizada a conjunção "," que representa "e". Assim, para saber se Pedro e Maria amam as mesmas pessoas pode-se formular a questão:

```
?-ama(maria,X), ama(pedro,X).
```

Para responder a essa questão, além do processo de casamento de padrões, Prolog também utiliza um mecanismo de busca chamado *backtracking*. Primeiramente, ele tenta satisfazer o primeiro objetivo:

```
ama(maria,X).
```

Se ele não consegue satisfazê-lo, ou seja, não encontra um fato que case com o fato dado, uma resposta negativa é fornecida ao usuário. Caso ele consiga satisfazer o objetivo, a base de dados é marcada indicando onde a pesquisa do primeiro objetivo parou e X é instanciado com um valor. Suponha que na base de dados existam os fatos:

```
ama(maria, jose).
ama(maria,antonio).
ama(pedro,antonio).
```

A variável X é instanciada com "jose". Prolog então tenta satisfazer o segundo objetivo com a variável X já instanciada:

```
ama(pedro,jose).
```

Prolog inicia sua pesquisa do início da base de dados buscando por este fato. Como ele não encontra um fato que se case, Prolog retorna ao objetivo anterior

ama(maria,X).

e procura outra resposta para ele iniciando sua pesquisa do marcador de lugar deste objetivo e deixando X como uma variável não instanciada. Então ele encontra o fato:

ama(maria,antonio).

X é instanciada com "antonio" e Prolog tenta satisfazer então o segundo objetivo que será:

ama(pedro,antonio).

Prolog inicia sua busca novamente do início da base de dados, pois este objetivo é diferente do anterior ("ama(pedro,jose)"). Ele encontra um fato que case com o fato procurado e fornece uma resposta ao usuário:

X=antonio.

O *backtracking* é um mecanismo poderoso de Prolog, mas sua utilização indevida pode levar a um baixo desempenho do programa.

D.4 Listas

Uma lista é uma estrutura de dados que representa uma seqüência ordenada de elementos. Elas são utilizadas para agrupar objetos e podem ser manipuladas como um objeto único. Cada elemento pode ser um termo qualquer: constantes, variáveis ou estruturas de dados. A notação para descrever uma lista consiste de colocar os elementos entre colchetes separados por vírgulas. Os seguintes exemplos representam uma lista vazia e uma lista com os elementos "a", "b" e "c", respectivamente:

[ ]
[a,b,c]

Cada elemento de uma lista pode ser acessado através de sua posição na lista ou através da divisão da lista em cabeça e cauda. A cabeça de uma lista é o seu primeiro elemento. A cauda é uma outra lista com todos os outros elementos exceto o primeiro. A notação em Prolog para representar uma lista com cabeça X e cauda Y é "[X|Y]".

A recursão provê a base para o processamento de listas. Por exemplo, a busca por um determinado elemento em uma lista é recursiva. O elemento procurado é inicialmente comparado com a cabeça da lista. Se ele não é este elemento, a busca é feita novamente na cauda da lista. Como dito anteriormente, a cauda de uma lista também é uma lista, de modo que a busca é feita comparando-se o elemento procurado com a cabeça da cauda. Se o elemento não é encontrado, a busca continua sucessivamente até que a cauda seja uma lista vazia ou até que o elemento seja encontrado.

Em Prolog para que uma busca recursiva seja feita é necessário definir um predicado recursivo. Por exemplo, o relacionamento X pertence a uma lista Y é recursivo e descrito como:

```
pertence( X, [ X | _ ] ).
pertence( X, [ W| Y ] ) :- pertence( X, Y).
```

Isto significa que, X pertence a uma lista dada se X é a cabeça desta lista ou se ele está na cauda da lista.

Bibliografia

[Ana95] Anil Ananthaswamy. Data Communications Using Object-Oriented Design and C++. McGraw-Hill, 1995.

[Bei90] Boris Beizer. "*Software Testing Techniques*". 2a edição, Thomson Comp. Press, 1990.

[Bei95] Boris Beizer. "*Black-Box Testing:Techniques for Functional Testing of Software and Systems*". John Wiley & Sons, 1995.

[Bin94a] Robert V. Binder. "Design for Testability with Object-Oriented Systems". *Communications of the ACM*, v. 37, no. 9, set./1994, p. 87-101.

[Bin94b] Robert V. Binder. "The FREE Approach to Testing Object-Oriented Software: An Overview". RBSC-94-003, Robert Binder Systems Consulting , Chicago, 1994.

[BS94] Stéphane Barbey e Alfred Strohmeier. "The Problematics of Testing Object-Oriented Software". *Second Conference on Software Quality Management*, vol. 2, Edinburgh, Scotland, UK, jul./1994, p. 411-426.

[BR98] Luiz E. Buzato e Cecília M. F. Rubira. "*Construção de Sistemas Orientados a Objetos Confiáveis*". 11a Escola de Computação, Rio de Janeiro, jul./1998.

[Chi93] Shigueru Chiba. "Open C++ Release 1.2 - Programmer's Guide". Technical Report 93-3, Department of Information Science, University of Tokyo, 1993.

[CM87] W. F. Clocksin e C. S. Mellish. "*Programming in Prolog*". Springer-Verlag, 3a edição, 1987.

[CCC99] Wu-Chi Chen, Deng-Jyi Chen e Chyan-Goei Chung. "An Expert-System Technique for Object-Oriented Program Testing". In: *Journal of Object-Oriented Programming*, fev./1999, p. 25-35.

[Den91] Richard Denney. "Test-Case Generation from Prolog-Based Specifications". In: *IEEE Software*, mar./1991, p. 49-57.

[Fre91] Roy S. Freedman. "Testability of Software Components". In: *IEEE Transactions on Software Engineering*, vol. 17, n° 6, jun./1991, p. 553-564.

[HM92] Mary Jean Harrold, John D. McGregor. "Incremental Testing of Object-Oriented Class Structures". In: *Proceedings of 14th International Conference on Software Engineering*, 1992.

[HP91] Daniel M. Hoffman e Paul Strooper. "Automated Module Testing in Prolog". IEEE Transactions on Software Engineering, v. 17, n. 9, set./1991, p. 934-943.

[Hof89] Daniel Hoffman. "Hardware Testing and Software ICs". In: *Proceedings of Pacific NW Software Quality Conference*. Oregon, set./1989.

[How86] William E. Howden. "A Functional Approach to Program Testing and Analysis". *IEEE Transactions on Software Engineering*, vol. 12, no. 10, out./1986.

[HS91] Daniel M. Hoffman e Paul Strooper. "Automated Module Testing in Prolog". In: *IEEE Transactions on Software Engineering*, v. 17, n. 9, set./1991, p. 934-943.

[HS95] Daniel Hoffman e Paul Strooper. "The testgraph methodology: Automated testing of collection classes". In: *Journal of Object-Oriented Programming*, Nov.-Dez./1995, p. 35-41.

[HS97] Daniel Hoffman e Paul Strooper. "ClassBench: a Framework for Automated Class Testing". In: *Software - Practice and Experience*, vol. 27(5), mai./1997, p. 573-597.

[Jac92] Ivar Jacobson. "*Object-Oriented Software Engineering - A Use Case Driven Approach*". Addison-Wesley, 1992.

[KGH+95] David Kung, Jerry Gao, Pei Hsia, et. al.. "Developing an Object-Oriented Software Testing and Maintenance Environment". In: *Communications of the ACM*, vol. 38, no 10, out./1995.

[KGH97] David Kung, Jerry Gao e Pei Hsia. "A Test Strategy for Object-Oriented Programs". Computer Science Engineering Dept., University of Texas at Arlington, Mar./1997.

[Lis97] Maria Lucia Blanch Lisboa. "Reflexão Computacional no Modelo de Objetos". Mini-curso apresentado no II Simpósio Brasileiro de Linguagens de Programação, Campinas, São Paulo, ago./1997.

[LL87] J. Leite, O. G. Loques. "Software". *II Simpósio de Computação Tolerante a Falhas*, cap. 4 do mini-curso intitulado "Introdução à Tolerância a Falhas", Campinas, SP, jul./87.

[LRP+95] Ted Lewis, Larry Rosenstein, Wolfgan Pree, et. al. "Object Oriented Application Frameworks". Manniing Publications Co., Greenwich, 1995.

[Mar98] Eliane Martins. "Verificação e Validação". Instituto de Computação, UNICAMP. Jun./1998.

[McG96] John D. McGregor. "A Parallel Architecture for Component Testing of Object-Oriented Software". In: *http://www.cs.clemson.edu/~johnmc/new/pact*, Ago./1996.

[Mey00] Bertrand Meyer. "Building bug-free O-O software: An introduction to Design by Contract". In: http://eiffel.com/doc/manuals/technology/contract, ISE Inc., Santa Barbara, California, 2000.

[MT99] Eliane Martins e Cristina M. Toyota. "Construção de Classes Autotestáveis". In: *Anais do VIII Simpósio de Computação Tolerante a Falhas*, Campinas/SP, jul./1999, p.106-209.

[PK90] Dewayne E. Perry e Gail E. Kaiser. "Adequate Testing and Object-Oriented Programming". In: *Journal of Object-Oriented Programming*. Jan.-Fev./1990, p. 13-19.

[Pos96] R.M. Poston. "Specification-based Software Testing". IEEE Computer Society Press, 1996.

[Pre95] Roger S. Pressman. "Engenharia de Software". Makron Books, São Paulo, 1995.

[PSS+85] Herbert Pesch, Hans Schaller, Peter Schnupp, et. al.. "Test Case Generation Using Prolog". In: *8th International Conference on Software Engineering*, London, UK, ago./1995, p. 252-258.

[Rog87] Jean B. Rogers. "*A Prolog Primer*". Addison-Wesley, 1987.

[RBP+94] James Rumbaugh, Michael Blaha, W. Premerlani, et. al.. "*Modelagem e Projetos Baseados em Objetos*". Editora Campus, Rio de Janeiro, 1994.

[Sie96] Shel Siegel. "*Object Oriented Software Testing: a Hierarchical Approach*". John Wiley & Sons, 1a edição, New York, 1996.

[SN94] Ernst Siepmann e A. Richard Newton. "TOBAC: A Test Case Browser for Testing Object-Oriented Software". *ISSTA'94*, Seattle, Washington, USA, 1994, p. 154-168.

[SR92] M.D. Smith e D. J. Robson. "A framework for testing object-oriented programs". In: *Journal of Object-Oriented Programming*. Jun./1992, p. 45-53.

[TM98] C. M. Toyota e E. Martins. "Reutilização em Teste de Software Orientado a Objetos: Metodologia para Construção de Classes Autotestáveis". In: *IX Conferência Internacional de Tecnologia de Software - Qualidade de Software*, Curitiba, jun./1998.

[TR94] *Technical Report 95-01*. "Testability of Object-Oriented Systems". Reliable Software Technologies Corporation, Loudon Tech Center, Dez./1994.

[VM95] Jeffrey M. Voas e Keith W. Miller. "Software Testability: The New Verification". In: *IEEE Software*, 12(3), mai./1995, p. 17-28.

[UP83] Hasan Ural e Robert L. Probert. "User-Guided Test Sequence Generation". In: *Protocol Specification, Testing and Verification III*. North-Holland, 1983, p. 421-436.

[Wey98] E. Weyuker. "Testing component-based software: a cautionary table. In: *IEEE Software*, set.-out./1998, p.54-59.

[WKC+97] Y. Wang, G. King, I.Court, et al.. "On Testable Object-Oriented Programming". In: *Software Engineering Notes*, vol.22, nº 4, Jul./1997, p. 84-90.