Avaliação da Construção e Uso de Classes Autotestáveis

Rosileny Lie Yanagawa

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Avaliação da Construção e Uso de Classes Autotestáveis

Rosileny Lie Yanagawa[1]

Dezembro de 2000

Banca Examinadora:

- Profa. Dra. Eliane Martins (Orientadora)

Instituto de Computação - UNICAMP

- Prof. Dr. Mario Jino

Faculdade de Engenharia Elétrica e Computação -UNICAMP

- Prof. Dr. Luiz Eduardo Buzato

Instituto de Computação - UNICAMP

- Profa. Dra. Maria Beatriz Felgar de Toledo (Suplente)

Instituto de Computação – UNICAMP

[1] Auxílios concedidos pelo Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Yanagawa, Rosileny Lie

Y15a Avaliação da construção e uso de classes autotestáveis /Rosileny Yanagawa Lie -- Campinas, [S.P. :s.n.], 2000.

Orientador : Eliane Martins

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

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

Avaliação da Construção e Uso de Classes Autotestáveis

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rosileny Lie Yanagawa aprovada pela Banca Examinadora.

Campinas, 22 de dezembro de 2000.

Eliane Martins

Profa. Dra. Eliane Martins

(Orientadora)

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

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 22 de dezembro de 2000, pela Banca Examinadora composta pelos Professores Doutores:

Prof. Dr. Mário Jino
FEEC - UNICAMP

Prof. Dr. Luiz Eduardo Buzato
IC – UNICAMP

Profa. Dra. Eliane Martins
IC – UNICAMP

Para meus pais com amor.

Agradecimentos

Aos meus pais pela ajuda incondicional, pelo apoio e estímulo constantes em todos os momentos de minha vida.

A minha orientadora e amiga Eliane Martins, pela compreensão com que conduziu a realização deste trabalho e a forma com que me ajudou a lidar com as dificuldades que surgiram durante o percurso.

A Ana Maria, Paulinho e Luciana do Instituto Nacional de Pesquisas Espaciais – INPE, pela paciência com que sempre atenderam as minhas dúvidas e questionamentos durante os testes do sistema TMTC.

Aos amigos do mestrado pela troca de experiências, pelas madrugadas inesquecíveis de estudo, pelo companheirismo de todas as horas e momentos de alegria. Em especial, aos queridos amigos: Andréia, Simone, Márcia Renata, Gaúcho, Alessandro, Gandhi, Nathan, Hilson, Sandro e Ivan.

A minha prima Akemi e amiga Patricia pelo incentivo e ajuda imprescindíveis, pela grande amizade que me mostraram e por tudo que fizeram para que eu conseguisse finalizar esta tese.

Ao amigo Sandro que me ajudou na fase de revisão do texto, pacientemente auxiliando a rever os textos em inglês.

Agradeço também ao Conselho Nacional de Desenvolvimento Científico e Tecnológico – CNPq, e a meus pais pelo auxílio financeiro, essenciais para a concretização deste trabalho.

Por fim, as pessoas que fizeram parte da minha vida durante a fase de mestranda e que de alguma forma estiveram ao meu lado me incentivando a enfrentar as inúmeras dificuldades, e compartilhando não só os problemas, mas também os momentos de alegria e realizações. Em particular, agradeço o Marcelo que esteve ao meu lado, me apoiando e ajudando durante grande parte deste percurso.

Resumo

Esta dissertação descreve uma metodologia para a construção e uso de classes autotestáveis. A metodologia busca melhorar a testabilidade de Sistemas Orientados a Objetos, através da adaptação de conceitos consolidados em hardware, como o Projeto para Testabilidade (DFT, do inglês *Design for Testability*) e o conceito de autoteste, que significa a adição de estruturas especiais a componentes para permitir que testes sejam gerados e avaliados internamente. Além do conceito do autoteste, nosso trabalho faz uso da hierarquia de herança existente para permitir também a reutilização dos testes. Neste sentido, a Técnica Incremental Hierárquica (HIT) [Har92] é usada com o objetivo de permitir que os testes de uma superclasse possam ser reusados para uma subclasse.

As vantagens e desvantagens do uso da metodologia são mostradas na dissertação de duas formas: a primeira através de uma apresentação dos passos para a construção e uso de classes autotestáveis baseada nos testes de classes da biblioteca MFC *(Microsoft Foundation Classes);* e a segunda através do uso da metodologia nos testes de um conjunto de classes de uma aplicação real.

Esta dissertação também apresenta uma avaliação empírica feita para determinar se os testes gerados neste trabalho têm um bom potencial para encontrar falhas. Os resultados mostraram que os testes gerados possuem um bom potencial na detecção de falhas. Nossos resultados ainda não oferecem uma evidência definitiva sobre a eficácia do conjunto de requisitos de teste gerado, porém mostram que a estratégia de teste adotada pode ser útil nos testes de classes de sistemas OO.

Abstract

This thesis describes a methodology for building and using self-testing classes. The methodology aims to improve the testability of Object Oriented (OO) Systems, through the adaptation of concepts consolidated in hardware, as the Design for Testability (DFT) and the self-testing concepts, which means the addition of special structures to the components to allow tests to be generated and evaluated internally. In addition to the self-testing concept, our work makes use of the existing inheritance hierarchy allowing also the tests reuse. In this direction, the *Hierarchical Incremental Testing* (HIT) approach, proposed in [Har92], allows a test sequence for a parent class to be reused, whenever possible, when testing one of its subclasses.

The advantages and disadvantages of the methodology are shown in the thesis in two ways: (1) a presentation of steps that should be carried out in building and using a self-testing class, based on classes from the *Microsoft Foundation Class (MFC);* and (2) the methodology use in testing a set of classes from a real application.

This thesis also presents an empirical evaluation to determine if the tests generated in this work have a good potential for finding faults. The results have shown that the generated tests have a good potential in fault detention. Our results still do not offer a definitive evidence on the effectiveness of a set of generated test cases; however, they show that the adopted strategy of test can be useful in the OO systems classes tests.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1 Contexto e Motivação

A verificação de software é normalmente utilizada quando se quer garantir um software de qualidade. A atenção aumenta quando vidas humanas ou fortunas dependem da confiabilidade deste software. Pode-se citar duas maneiras mais usadas de verificação: a primeira através de uma *verificação formal,* o que envolve um conjunto de teoremas e provas matemáticas, sendo este um processo rigoroso, que muitas vezes se torna mais difícil que a própria escrita do programa. A segunda, mais utilizada, a realização de *teste do software,* que tem como objetivo maior encontrar falhas, através da execução repetitiva do software, dando-lhe dados de entrada, obtendo dados de saída e os comparando com os resultados esperados. Desta forma, pode-se dizer que este processo engloba as seguintes atividades: geração, execução e avaliação dos testes. Pela ausência de consenso quanto à terminologia a ser usada em português, o termo falha usado aqui será definido como: uma falha ("*fault*") é a causa direta de um erro; um erro é a parte do estado do sistema que pode levar a um defeito ("*failure*").

Na fase de geração dos testes, dados de entrada devem ser escolhidos para exercitar o software. Entretanto, na maioria das vezes, é impossível realizar testes completos, já que o conjunto de entradas possíveis pode ser infinito. Neste sentido, deve-se escolher um subconjunto de entrada adequado capaz de encontrar falhas existentes. Para isto, existem técnicas de teste que possibilitam a criação de conjuntos de testes capazes de revelar o maior número possível de falhas no software. Duas categorias principais de técnicas são usadas: os testes funcionais e os testes estruturais. Testes estruturais ou de caixa branca são baseados na implementação, utilizando a estrutura interna do programa para gerar os dados de teste. Já os testes funcionais ou de caixa preta são baseados na especificação,

sendo projetados para validar os requisitos funcionais sem ter o conhecimento do código ou da estrutura interna do software.

Pode-se citar alguns exemplos de testes estruturais, como [Mye79, Bei90]: *testes de fluxo de controle, e testes de fluxo de dados.* Os testes de fluxo de controle, baseados no grafo de controle do programa, selecionam um conjunto de caminhos para serem executados através do conjunto de requisitos de testes gerado. A escolha dos caminhos a serem exercitados deve ser feita de forma a ser eficiente em achar falhas. Para isto, condições são estabelecidas e estas devem ser satisfeitas durante os testes, sendo chamadas de *critérios de adequabilidade* ou simplesmente *critérios de testes*. No requisito de testes de fluxo de controle, um conjunto de critérios foram definidos, tais como: *critério baseado nas instruções,* que requer que cada instrução seja executada ao menos uma vez para os dados de entrada selecionados; ou *critério baseado nos ramos,* que requer que cada ramo de uma decisão do programa seja executado pelo menos uma vez. Já os testes de fluxo de dados, definem um conjunto de critérios de testes baseados na análise do grafo de fluxo de dados, de forma a explorar os relacionamentos de *definição-uso* de variáveis no programa, ou seja, caminhos entre os pontos onde uma variável é definida e os pontos onde o uso da variável é afetado. Neste teste é feita uma distinção entre os tipos de referências a variáveis: usada em um cálculo (c-uso), e usada em um predicado (p-uso). Baseados nestes conceitos, os testes de fluxo de dados procuram verificar se o programa satisfaz a um determinado critério de teste. Exemplos de critérios de testes são: critério *todas as definições,* requer que cada definição seja exercitada pelo menos por um uso (p-uso ou c-uso) dessa variável; critério *todos os usos,* requer que todas as associações entre uma definição de uma variável e todos os seus usos (c-uso ou p-uso) sejam exercitadas; critério *todos os DU-caminhos,* requer a execução de todo caminho livre de laços entre uma definição de uma variável e todos os seus usos (c-uso ou p-uso).

Exemplos de testes funcionais são [Bei90, Mye79, Bei95]: *testes de dados; testes de transição de estados.* Os testes de dados têm por objetivo selecionar dados de acordo com caraterísticas dos domínios de entrada e saída do programa, com o objetivo de encontrar falhas com a manipulação de dados. Alguns testes que se baseiam em dados na geração dos seus requisitos de teste podem ser citados: *testes de partições de equivalência,* que dividem o domínio de entrada em classes de equivalências (entradas válidas e entradas

inválidas), com o intuito de gerar requisitos de testes que exercitem valores pertencentes a cada classe; e *testes de valores limites,* que testam valores limites de cada classe de equivalência. Os testes de transição de estado são baseados em máquinas finitas de estados, que mostram a variação de estados de um sistema ao longo do tempo. Os requisitos de testes gerados devem exercitar estados e transições do grafo.

Já na fase de execução dos testes, os dados gerados (requisitos de testes) devem ser aplicados e os testes feitos, obtendo um conjunto de saídas. No requisito de estar-se testando um componente de forma isolada, dois elementos de teste são necessários: *stubs e drivers. Stubs* são utilizados para substituir outros componentes que interagem com o componente em teste. Um *driver* serve para ativar e controlar os testes, fornecendo entradas e recebendo saídas.

Na fase de avaliação dos testes é verificado se os dados de saída estão corretos conforme a especificação do software. Nesta etapa, existe a necessidade de um mecanismo que possibilite verificar a ocorrência de defeitos. Este mecanismo é denominado de *oráculo,* tendo este um papel fundamental na determinação de testes bem sucedidos, pois a existência de um *oráculo* não confiável acarreta conclusões incorretas.

Entretanto, apesar de existirem bons critérios e técnicas de teste, esta é considerada uma atividade consumidora de tempo e recursos. Garantir um nível de confiabilidade de um software capaz de evitar que desastres aconteçam não é um trabalho trivial. Desta maneira, a testabilidade de um software mostra ser importante na redução dos esforços necessários, uma vez que determina qual será o grau de dificuldade em testar este software. Assim, projetar o software visando torná-lo mais testável desde o início de sua construção significaria a redução na dificuldade de encontrar falhas e um menor esforço para produzir testes que garantam a qualidade esperada.

Particularmente, ao se falar de sistemas Orientados a Objetos (OO), a garantia de uma boa testabilidade se mostra fundamental, pois a possibilidade de se construir componentes de software reutilizáveis, que por sua vez necessitam de um alto nível de confiabilidade, exige a realização de mais testes a cada vez que o componente é reutilizado. Além disso, muitos dos problemas já existentes de testes são agravados em sistemas OO, devido principalmente ao conjunto de características introduzido pela abordagem, tais como: o encapsulamento, a herança, a ligação dinâmica, e o polimorfismo.

Portanto, faz-se necessário o uso de soluções que permitam que componentes OO sejam fáceis de serem testados, garantindo assim que as vantagens oferecidas pelo reuso não sejam perdidas na fase de testes.

Neste trabalho, busca-se melhorar a testabilidade de sistemas OO, através da utilização do conceito de Projeto para Testabilidade (DFT, do inglês *Design for Testability*), usado em hardware para melhorar a testabilidade de circuitos integrados (CI), e adaptado por Binder [Bin94] para sistemas OO. O conceito de **autoteste,** denominado em inglês *"Built-in-Self Test"* ou BIST, significa a adição de estruturas especiais ao componente para permitir que testes sejam gerados e avaliados internamente. Em [Bin94] é proposta a utilização do autoteste em sistemas orientados a objetos, através da introdução da capacidade de geração, execução e análise automática dos testes, tendo-se assim a criação de **classes autotestáveis**.

Além do conceito de autoteste embutido que permite reduzir o esforço na realização dos testes, uma outra proposta para reduzir ainda mais esse esforço é a utilização da hierarquia de herança para permitir também a reutilização dos testes.

A **Técnica Incremental Hierárquica** (HIT), proposta em [Har92] aplica testes de forma incremental, explorando a natureza hierárquica da relação de herança para testar grupos de classes, com isso permitindo que os testes de uma superclasse possam ser usados para orientar os testes de uma subclasse.

Um protótipo, chamado *Concat* (*Construção de Classes AutoTestáveis*) que foi desenvolvido e implementado por Toyota [Toy98], auxiliou os testes realizados neste trabalho. Este protótipo utiliza os conceitos acima descritos para auxiliar a construção de classes autotestáveis, através da geração de requisitos de testes automáticos, bem como a execução destes e a avaliação dos resultados.

1.2 Objetivos

Esta dissertação apresenta um conjunto de estudos, utilizando os conceitos de autoteste, em conjunto com a técnica HIT nos testes de classes, tendo como objetivo principal responder as seguintes perguntas:

- *Quais as dificuldades encontradas para se aplicar a metodologia proposta?*

- *A estratégia de teste utilizada é eficiente para teste de classes?*

Neste sentido, serão apresentados os passos, as soluções e as dificuldades encontradas na aplicação da metodologia no teste de um conjunto de classes. Além disso, as etapas realizadas para a construção de classes autotestáveis serão mostradas, assim como os resultados obtidos dos estudos feitos sobre o modelo de teste.

1.3 Organização do Texto

Esta dissertação é composta de uma coleção de artigos e capítulos escritos em português e inglês. Por este motivo, alguns dos conceitos serão apresentados de forma repetida nos Capítulos escritos em forma de artigos, já que são necessários para o embasamento dos mesmos.

O Capítulo 2 apresenta conceitos importantes de testes e testabilidade de software.

Capítulo 2 – Teste e Testabilidade de Software OO: Este Capítulo apresenta uma visão geral dos conceitos básicos relacionados a teste e testabilidade de sistemas orientados a objetos. Assim como apresenta trabalhos correlatos sobre testabilidade de software OO, fazendo uma comparação com a metodologia usada nesta dissertação.

Os Capítulos 3 e 4 contêm dois artigos sobre a construção e uso de classes autotestáveis e uma avaliação da aplicabilidade da proposta.

Capítulo 3 – Aplicando os Conceitos de Autoteste em Sistemas OO: Este artigo, escrito em inglês, descreve os passos realizados para a construção de uma classe autotestável. Para isto, duas classes foram escolhidas para os testes: uma classe da biblioteca MFC *(Microsoft Foundation Classes)*; e uma derivada da classe escolhida.

Capítulo 4 - Avaliação Empírica da Eficácia dos Testes baseados no Modelo de Fluxo de Transação em Sistemas Orientados a Objetos: Este artigo apresenta uma avaliação empírica feita para determinar se os testes baseados no modelo utilizado neste trabalho (modelo de fluxo de transação) têm um bom potencial para encontrar falhas. O artigo apresenta de forma detalhada a estratégia utilizada para gerar os testes, o procedimento usado para criar os mutantes e os resultados obtidos.

Os Capítulos 5 e 6 descrevem a utilização de classes autotestáveis em uma aplicação real.

Capítulo 5 - O Uso de Métricas de Software para Analisar a Testabilidade e a Reusabilidade de um Sistema Orientado a Objetos : Este artigo apresenta uma análise feita para a escolha das classes a se tornarem autotestáveis, pois devido ao fator tempo que é normalmente restrito na atividade de testes em geral, não foi possível a realização dos testes de todas as classes da aplicação. Esta análise foi baseada nos seguintes critérios: criticalidade, reusabilidade e testabilidade. Com exceção do primeiro critério, que foi avaliado empiricamente através de consultas aos desenvolvedores, os outros foram obtidos através de métricas de software.

Capítulo 6 – O Uso de Classes Autotestáveis em uma Aplicação Aeroespacial: Este Capítulo apresenta os passos realizados nos testes das classes escolhidas, as experiências e os resultados obtidos com os testes do sistema alvo.

Capítulo 7 – Conclusão : Este Capítulo descreve as contribuições dessa dissertação e as extensões para continuidade do trabalho.

Capítulo 2

Teste e Testabilidade de Software Orientados a Objetos

Este Capítulo apresenta uma visão geral dos conceitos de teste e testabilidade de software OO, descrevendo também trabalhos que enfocam o Projeto para Testabilidade nestes sistemas. O objetivo é fornecer uma base para o entendimento do que será discutido nos Capítulos seguintes, bem como mostrar uma comparação dos trabalhos apresentados com a metodologia desta dissertação. As referências [Bin94,Voa95b,Bei90] podem ser consultadas para uma apresentação mais detalhada destes conceitos.

2.1 Teste de Software OO

O paradigma de orientação a objetos é baseado no encapsulamento de código e dados em uma única unidade chamada *objeto*. Um objeto pode ser definido como uma entidade instanciada que possui [Buz98]: (1) um estado encapsulado; (2) a habilidade para enviar mensagens para outros objetos como forma de operações destes outros objetos; (3) um comportamento bem definido e uma identidade única.

Objetos são instâncias de uma *classe*. Uma classe pode ser definida como um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares [Buz98]. Toda classe tem um nome e um corpo que define o conjunto de atributos e operações possuídas pelas instâncias.

Além destes conceitos, o paradigma de orientação a objetos possui algumas características poderosas, como: o *encapsulamento*, a *herança*, o *polimorfismo* e a *ligação dinâmica*. A seguir será dada uma visão geral destes conceitos; a referência [Buz98] pode ser consultada para informações mais detalhadas.

O encapsulamento possibilita que um objeto esconda seu estado do mundo externo, encapsulando atributos e operações. Assim, um objeto pode possuir dois tipos de dados:

públicos e privados. Os dados públicos permitem que outros objetos utilizem, chamem ou manipulem as características (atributos e métodos) contidas no objeto. Os dados privados, podem ser utilizados, manipulados e chamados apenas pelos objetos que os contêm. Além disso, algumas metodologias e linguagens oferecem diferentes opções de visibilidade destas características. Em C++, por exemplo, têm-se características *públicas* (visíveis por todos os objetos), *privadas* (visíveis internamente ao objeto, somente) e *protegidas* (visíveis internamente ao objeto ou aos objetos de classes derivadas).

A herança é um mecanismo que permite que uma classe (classe *derivada* ou *subclasse*) seja criada a partir de extensões ou modificações, herdando métodos ou atributos de outra classe já existente (classe *base* ou *superclasse*). A *herança múltipla* acontece quando uma classe derivada herda característica de mais de uma classe base (estas constituem então suas classes *ancestrais*).

O polimorfismo ("várias formas para o mesmo nome") é utilizado na orientação a objetos para possibilitar que diferentes tipos de objetos respondam à mesma mensagem de formas diferentes. Alguns conceitos relacionados ao polimorfismo são importantes para os testes: redefinição, sobrecarga e parametrização. A redefinição permite que os métodos ou atributos da classe base sejam redefinidas na subclasse. A sobrecarga possibilita associar um método a um objeto em tempo de execução. Já a parametrização possibilita utilizar classes genéricas ou classes parametrizadas, as quais servem de moldes para outras classes. A ligação dinâmica (do inglês,"*dynamic*" ou "*late binding*") pode ser definida como sendo a associação, em tempo de execução, entre uma operação e a sua implementação.

Por um lado, estas características descritas são importantes para introduzir a maior vantagem deste paradigma: o reuso. Por outro, são responsáveis por acarretar maiores dificuldades em se testar os componentes, pois intensificam problemas já existentes na realização dos testes. Smith e Robson [Smi92] citam um conjunto de problemas associados aos testes de programas orientados a objetos. Uma razão apresentada para os testes em sistemas OO serem mais difíceis é que os programas não possuem uma execução sequencial e os métodos da classe podem ser arbitrariamente combinados. Portanto, o teste se torna um problema de busca, ou seja, uma procura de sequências de execuções de métodos capazes de produzir falhas. Outra razão descrita é que as classes

são elementos que não podem ser testados dinamicamente, apenas os objetos que são instâncias das classes, são testáveis. Neste sentido, classes parametrizadas e classes abstratas, que não podem ser instanciadas por falta de uma definição completa, têm que ser testadas apenas de forma indireta.

Portanto, tem-se a necessidade de soluções que permitam melhorar a testabilidade do sistema, bem como a adaptação dos métodos de testes de software existentes a fim de tratar estes novos aspectos. Assim, existem algumas modificações na realização dos testes em sistemas OO, tais como: nas fases dos testes, nos modelos de testes, técnicas de testes e na testabilidade do software.

As características dos testes OO são apresentados nos subitens a seguir.

2.1.1 Fases de Teste

As fases de testes são utilizadas com o objetivo de melhor organizar a realização de testes, sendo que cada fase engloba etapas ao longo do desenvolvimento do software. Testes OO podem ser vistos segundo diferentes níveis de abstração. Estes podem ser divididos de forma semelhante aos testes de sistemas convencionais, contendo entretanto, algumas modificações que podem ser percebidas nas fases existentes nos testes OO, tais como [Bin94, Pre95]:

- **teste de classe:** em sistemas OO, a unidade mínima é a classe, e os testes enfocam os objetos, seus métodos e o seu comportamento;
- **teste de integração:** diferentemente dos sistemas convencionais, que utilizam estratégias incrementais (*top-down* ou *bottom-up*), em testes em sistemas OO são utilizados duas diferentes abordagens: testes de *'threads'* e testes de uso. Os testes de *'threads'* têm como objetivo integrar classes necessárias para responder a uma determinada entrada ou a um determinado evento do sistema. Já os testes de uso, visam integrar as classes de acordo com o relacionamento existente entre elas, iniciando pelas classes independentes, que não se relacionam com outras classes, depois classes que se relacionam diretamente, prosseguindo até que todas as classes estejam integradas;
- **teste de sistemas:** os testes são realizados da mesma forma que em testes convencionais, visando determinar se o sistema satisfaz à especificação.

2.1.2 Abordagens para testes de subclasses

Existem técnicas que foram desenvolvidas para testes OO que levam em conta a hierarquia de herança existente entre as classes, sendo elas: *técnica de nivelamento da hierarquia, e a técnica incremental hierárquica* (HIT).

A ***Técnica Incremental Hierárquica*** (HIT) [Har92] possui como maior benefício a redução do esforço adicional para testar novamente cada subclasse. Nesta técnica, as características de uma classe são classificadas para determinar como serão testadas. Características **novas,** que são aquelas definidas apenas na subclasse, devem ser testadas completamente. Características **herdadas,** que são aquelas herdadas sem modificações pela subclasse, devem ser testadas somente se interagem com métodos novos ou redefinidos. Por fim, características **redefinidas**, que são aquelas definidas na superclasse porém modificadas na subclasse, podem reutilizar os modelos de teste já existentes.

Assim, apenas os atributos novos, substituídos ou alterados são testados. A técnica é hierárquica porque se baseia na hierarquia entre classes introduzida pelo mecanismo de herança, e é incremental, pois visa reutilizar os testes de uma classe em um nível hierárquico para testar classes em níveis subsequentes.

Apesar das vantagens oferecidas através da herança, a técnica HIT impõe algumas restrições de uso, tais como: (1) a herança existente deve ser de comportamento; e (2) não é aplicável na existência de herança múltipla. Informações mais detalhadas sobre a abordagem HIT podem ser obtidas em [Har92].

Diferente da abordagem HIT, a *Abordagem baseada no nivelamento da Hierarquia de Classe* [Bin96] leva em consideração o nivelamento da hierarquia de herança nos testes de subclasses. Neste sentido, todas as características da classe, tanto novas quanto herdadas ou redefinidas, são utilizadas no desenvolvimento do modelo de teste.

O nivelamento da hierarquia de classe possui uma implementação mais direta. Entretanto, exige o reteste completo das subclasses, sem o aproveitamento (reuso) de testes gerados anteriormente para as superclasses.

Nas próximas seções os conceitos de testabilidade de software serão apresentados, bem como trabalhos que propõem projetar o software visando a testabilidade.

2.2 Testabilidade de Software

A qualidade do processo de desenvolvimento é essencial quando se quer construir um software confiável. Para se obter esta qualidade, o software deve ser bem especificado, bem projetado, bem desenvolvido, e bem testado [Sha00]. Ignorar uma destas etapas e esperar que os testes consigam detectar os problemas deixados é problemático, pois testes são processos consumidores de recursos (ou seja, tempo e dinheiro) que nem sempre são disponíveis e requerem esforços para realizá-los.

Construir um software visando ser mais testável ou visando a sua *testabilidade,* pode ser uma boa alternativa para se diminuir os esforços com os testes e é um passo significativo para um produto de qualidade.

A *testabilidade* de um software pode ser definida através de quão fácil é satisfazer uma meta particular de teste através de critérios de testes estabelecidos, como por exemplo, exercitar todas as instruções de um programa. O grau de testabilidade de um componente ou sistema tem influência direta nos esforços gastos na fase de testes. Assim, quanto maior a testabilidade de um componente ou sistema, mais fácil será alcançar os objetivos estabelecidos.

Uma definição mais formal de *testabilidade* de software, é a definida pelo padrão IEEE 610.12 como sendo:

1. em que medida um sistema (ou componente) facilita o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos;
2. em que medida um requisito é representado de forma a permitir o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos.

Além da definição tradicional, Voas e Miller [Voa95b] apresentam uma outra definição de *testabilidade*, que difere por não se preocupar em verificar a facilidade com que algum critério de seleção de entrada pode ser satisfeito durante os testes, mas verificar a probabilidade que um programa possa produzir saídas incorretas requisito existam falhas. Desta forma, a *testabilidade* é definida como sendo a possibilidade de ocorrer um defeito requisito existam falhas, ou seja, um componente com baixa testabilidade tende a produzir saídas corretas para a maioria das entradas que executam uma falha.

Segundo Hoffman [Hof89], um software com boa testabilidade deve possuir duas características importantes: boa controlabilidade e boa observabilidade. A observabilidade refere-se à facilidade em se determinar o quanto as entradas fornecidas afetam as saídas obtidas. A controlabilidade refere-se à facilidade de se produzir a saída desejada a partir da entrada fornecida. Assim, para se testar um componente deve ser possível controlar (controlabilidade) suas entradas e estado interno, bem como observar (observabilidade) a sua saída. Em geral é difícil conseguir uma boa controlabilidade e observabilidade, em especial se o componente está embutido em outros componentes.

Neste sentido, existem trabalhos que abordam a testabilidade do software, de forma a construir o software visando a sua testabilidade. Pode-se citar os trabalhos de Voas e Miller [Voa95b] e Binder [Bin94], que usam DFT em sistemas OO. Os conceitos descritos a seguir foram utilizados na verificação das melhorias na testabilidade da classe com o uso da metodologia adotada neste trabalho, apresentados no final deste Capítulo.

2.2.1 Projeto Visando Testabilidade (Voas e Miller)

Como citado anteriormente, a testabilidade abordada por Voas e Miller difere da definição tradicional, pois verifica a probabilidade de um programa esconder falhas. Desta forma, os autores propõem a construção do software visando a testabilidade e estabelecem medidas matemáticas para estimar esta testabilidade, com base na capacidade do software de esconder falhas.

Para se construir um software visando a testabilidade definida por Voas e Miller, deve-se projetar o software de tal forma que um defeito tenha grande probabilidade de ocorrer requisito existam falhas. Desta forma, a preocupação está em sempre poder observar a ocorrência de um defeito. Esta ocorrência acontece quando o sistema produz uma saída diferente da saída esperada. Assim, um defeito ocorre e é observado, quando três eventos são realizados:

(a) uma falha deve ser executada;

(b) um erro é criado;

(a) o erro deve ser propagado para uma saída observável.

Portanto, é importante que após a execução de uma falha, o erro seja criado e principalmente, não seja mascarado antes de chegar a uma saída, pois este mascaramento pode encobrir a existência da falha, diminuindo assim a testabilidade do sistema.

Neste contexto, para se definir os fatores que influenciam a testabilidade do software, é apresentado o conceito de *perda de informação,* relativo às informações computadas durante a execução do programa que não são repassadas para as saídas do programa. Mais especificamente, existem dois tipos de perda de informações: *as implícitas e as explícitas.*

Perda de informação implícita ocorre quando dois ou mais valores de entrada são utilizados por uma função e esta produz o mesmo resultado. A métrica DRR (do inglês, "*domain-to-range*") foi definida para medir o grau de perda de informação implícita existente em um módulo ou unidade.

A *perda de informação explícita* ocorre quando não é possível verificar valores de variáveis durante a execução ou no final da execução dos testes (através de uma saída). Este tipo de perda ocorre frequentemente com a existência de encapsulamento, pois este não permite que um módulo tenha acesso a informações internas de outros módulos que lhe prestaram serviços. Esta técnica é aceita como uma boa prática de programação, porém esconde informações internas, o que não é bom para a testabilidade do sistema, pois não permite que valores de variáveis internas sejam observadas na busca de falhas; em outros termos, a observabilidade é reduzida.

Voas e Miller propõem algumas formas de minimizar a perda de informação, que consistem em:

- **decompor a especificação:** usando a métrica DRR, decompor a especificação da classe para identificar os módulos que possuem ou não perda de informação implícita. Desta forma pode-se saber ainda em fase de projeto, quais módulos necessitam de mais testes.
- **minimizar o reuso de variáveis**: evitar o reuso de variáveis, pois este reuso destrói o valor que lhes foi atribuído anteriormente, proporcionando assim perda de informação implícita;
- **aumentar o uso de parâmetros de saída**: aumentar o uso de parâmetros de saída para reduzir a perda de informação explícita, o que pode se dar das seguintes formas: (a) através da inserção de declarações *write,* para imprimir o estado interno

da informação; (b) tratamento de variáveis locais como parâmetros de saída durante os testes; (c) através do uso de assertivas, para checar a informação interna durante a execução.

Além disso, os autores também recomendam estimar a probabilidade de falhas ainda existirem e estarem escondidas, através da *análise de sensibilidade*. Esta análise estima a probabilidade de um defeito acontecer no programa através da execução de uma falha. Como descrito anteriormente (página 12), um defeito é observado quando três eventos são realizados. Assim, a técnica estima a probabilidade destes eventos ocorrerem [Voa92]: análise de execução, análise de infecção, e análise de propagação.

A ***análise de execução*** é um método baseado na estrutura do programa, que serve para estimar a probabilidade da execução de uma região no programa. Isto é feito através da análise do número de requisitos de testes que executam esta determinada região.

Após a região ser executada, é realizada uma ***análise de infecção***, que estima a probabilidade da execução desta região criar um estado de dado incorreto (erro). Esta análise é feita através da introdução de uma série de mutantes na região.

Por fim, depois da criação de um erro, a ***análise de propagação*** estima a probabilidade deste erro se propagar até alguma saída, sendo a infecção simulada através de uma perturbação no programa, mudando o estado de um dado durante a execução do programa e comparando o resultado obtido com o do programa original.

Na seção seguinte é apresentado o Projeto Visando Testabilidade proposto por Binder [Bin94] que enfoca a definição tradicional de testabilidade.

2.2.2 Projeto Visando Testabilidade (Binder)

Segundo Binder [Bin94], uma boa testabilidade é resultado de uma boa prática de engenharia e um processo de desenvolvimento bem definido. Na construção do software visando testabilidade, seis aspectos básicos devem ser observados, pois influenciam diretamente na melhoria da testabilidade, são eles:

- **características de especificação**

A especificação do sistema é essencial na realização dos testes. Sem uma boa especificação, os resultados não podem ser avaliados corretamente, pois não há como

decidir se os testes passaram ou falharam, sem a existência de uma descrição do resultado esperado.

- **características da implementação**

A estrutura do sistema é um aspecto importante para a testabilidade do software. Por exemplo, a utilização do encapsulamento possibilita que a propagação de mudanças seja reduzida entre classes e métodos, pois permite que um objeto esconda seu estado do mundo externo, possibilitando a modularização. Entretanto, a utilização em conjunto com a herança pode acarretar um aumento de testes, devido à necessidade de testes em métodos herdados de uma classe pai quando esta é modificada, ou ainda na ocorrência de herança múltipla. Neste contexto, a ocorrência de encapsulamento ocasiona uma baixa observabilidade, já que não se tem acesso, externamente, ao estado interno de um objeto, dificultando assim os testes e diminuindo a testabilidade.

Na fase de implementação pode-se avaliar o grau de testabilidade de um programa através do uso de métricas de software ou ainda de técnicas como a análise de sensibilidade.

Binder cita um conjunto de métricas para sistemas OO existentes na literatura, que pode ser usado para encontrar pontos do sistema mais críticos e complexos, onde os testes devem ser concentrados. Por exemplo, métricas que analisam a herança existente nas classes (NOC- número de classes derivadas, ou DIT – profundidade na árvore de herança); e métricas que analisam o encapsulamento (LCOM- falta de coesão dos métodos). Algumas destas métricas foram utilizadas no estudo realizado nesta dissertação [Chi94] (Capítulo 05).

- **capacidade de teste embutido**

A capacidade de teste embutido fornece uma separação explícita entre os testes e a funcionalidade da aplicação, através da inserção de mecanismos de testes no componente. Os mecanismos de testes que podem ser adicionados são: métodos *set/reset*, métodos relatores e assertivas. A adição sistemática de métodos *set/reset* possibilita colocar um objeto em um estado pré-definido, não importando seu estado corrente. Os métodos relatores observam e armazenam o estado interno de um objeto durante a execução dos testes. Assertivas são usadas para testar uma condição referente ao estado do objeto e são compostas por condições e por uma rotina de exceção associada. Podem ser utilizadas

como: pré-condições (que testam uma condição de entrada do objeto), pós-condições (que testam uma condição de saída do objeto), e invariantes de classes (que testam condições invariantes do objeto). Assim, estes mecanismos podem contribuir para um aumento de testabilidade, introduzindo a capacidade de observação embutida no componente em teste.

- **sequência de teste**

A existência de uma sequência de teste é essencial. Uma sequência é o conjunto de requisitos de teste para o componente e o plano de como executá-lo.

- **ambiente de apoio ao teste**

A realização de testes exige automação; com a ausência de ferramentas, menos testes serão realizados, e portanto, maiores serão os gastos para alcançar a confiabilidade requerida, diminuindo assim a testabilidade.

- **processo de software em que o teste é conduzido**

O processo em que a construção do software é conduzido tem influência significativa na sua testabilidade. Ou seja, os testes devem ser utilizados de forma balanceada com outras práticas para garantir a qualidade, tais como: prototipação, inspeção e revisões para todos os estágios do produto.

2.2.3 Autoteste em Sistemas Orientados a Objetos

Binder propõe ainda o uso de mecanismos DFT (definido na seção 1.1) consolidados em hardware, nos testes de sistemas OO. Em hardware, circuitos adicionais são acrescentados dentro dos CIs, ainda em fase de projeto, com o intuito de facilitar o acesso aos componentes internos de um circuito, de maneira a facilitar também a observabilidade dos valores lógicos resultantes nestes pontos do circuito. Para simplificar a geração de padrões de teste são também incorporadas estruturas especiais para permitir que os padrões de teste sejam gerados e avaliados internamente. Dessa maneira, pode-se eliminar a etapa de geração externa de padrões de teste e o mais importante, a necessidade de um testador externo, que representa também um fator determinante na redução do custo final do produto [Cor91]. Este é o conceito de **autoteste,** denominado em inglês *"Built-in-Self Test"* ou BIST.

Neste contexto, existe um paralelo entre objetos e circuitos integrados (CIs) [Hof89], de forma que um objeto, assim como um CI, pode ser visto como uma caixa preta, com partes internas ocultas ao mundo exterior. Os métodos do objeto que possibilitam acesso aos serviços podem ser comparados aos pinos ou conjuntos de pinos de um CI. A existência deste paralelo possibilita aplicar a capacidade de autoteste dos CIs (capacidade de embutir sequências de testes e gerar testes de forma automática) em objetos, com a vantagem em relação ao hardware, de se poder remover os testes do código final através de compilação condicional. Desta forma, com base nos conceitos utilizados em hardware, tem-se a criação do conceito de **classes autotestáveis**.

A Figura 2.1 [Bin94] descreve os componentes necessários para a inserção de mecanismos BIST, adotados neste trabalho:

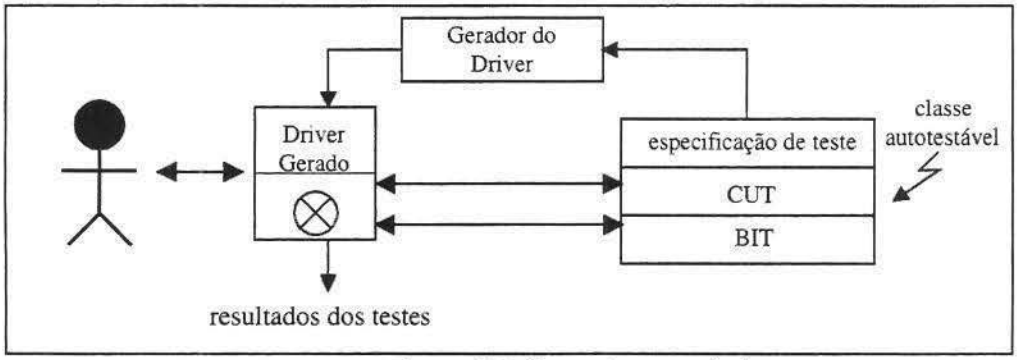

Figura 2.1 Classe Autotestável

- **a classe autotestável**

Este é um componente composto dos seguintes itens: (a) a *classe sob teste* (CUT, do inglês "*class under test*"); (b) da sua especificação de teste; (c) e das características BIT, que são compostas por métodos relatores (relatam o estado interno do objeto) e assertivas.

- **o gerador do driver**

Este componente é responsável pela geração do driver que ativará e controlará a CUT durante a execução dos testes.

- **driver gerado ou específico**

Este componente corresponde à sequência executável, gerada de acordo com o critério de teste implementado pelo gerador do *driver*, aplicado para um modelo de teste representado pela especificação de teste. Além da ativação e controle dos testes, este driver é

responsável pela avaliação dos testes. Esta avaliação é baseada no estado da CUT, dada pelas características BIT e pelas saídas obtidas.

2.3 Metodologia Usada nesta Dissertação

Muitos dos conceitos apresentados neste Capítulo são importantes e foram utilizados durante esta dissertação. A maioria deles serviu como base para o desenvolvimento da metodologia usada para a construção de classes autotestáveis, visando o teste de classes. A seguir serão apresentados os aspectos importantes que foram utilizados na metodologia:

- **uso do Conceito de Autoteste:**

Este conceito foi utilizado como base da metodologia desta dissertação. O objetivo foi usar os conceitos propostos por [Bin94] para a melhoria da testabilidade de sistemas orientados a objetos.

Desta forma foram empregados o uso de uma especificação de teste embutida na classe; o uso das características BIT, que são compostas por métodos relatores (relatam o estado interno do objeto) e assertivas; e a geração automática do *driver genérico,* que corresponde à sequência de teste executável.

- **uso de uma Ferramenta Automatizada:**

O protótipo *Concat* foi usado na geração da sequência de teste (driver específico). O objetivo do uso do protótipo foi reduzir o esforço para geração dos testes cada vez que uma classe é reusada. A *Concat* possibilita a automatização das seguintes tarefas:

A *Concat* é responsável pela geração automática de uma *parte do driver específico*, já que o usuário deve completá-lo com os parâmetros não escalares. Um *driver* pode ser definido como a sequência de teste executável para a CUT. A sequência de teste é o conjunto de requisitos de teste, cada requisito corresponde a um caminho no modelo de teste, e pode ser composto por um método ou um conjunto de métodos.

2.3.1 Melhoria da Testabilidade

Baseado nos conceitos propostos por Binder e Voas [Bin94,Voa95b] apresentados, observou-se que alguns fatores de testabilidade podem ser melhorados com o uso desta metodologia:

- **o entendimento da classe**, a possibilidade de se embutir na classe sua especificação de teste, contribui para resolver este problema normalmente existente em testes como os de regressão. Este recurso facilita o entendimento da classe a ser testada, pois descreve as diversas tarefas que um objeto desta pode realizar, otimizando e facilitando a realização dos testes;
- **o uso de métricas**, a utilização de métricas de software antes de se iniciar os testes, possibilita uma otimização no tempo gasto para se realizar testes, pois os resultados das métricas apontam para classes críticas ou mais propensas a conterem falhas, além de possibilitar a construção de uma ordem de teste parcial, que possibilita a redução do número de *stubs;*
- **a observabilidade das classes**, a utilização de assertivas utilizadas para checar a informação interna durante a execução, diminui a perda de informações, além de auxiliar na verificação dos resultados (oráculo parcial), possibilitando a melhoria da observabilidade;
- **os métodos relatores**, que juntamente com a especificação são embutidos na classe e podem ser reusados em testes futuros, possibilitam a construção de arquivos de resultados;
- **a existência de gerador de driver específico**, o que significa a geração automática e organizada de *uma sequência de teste* executável utilizada nos testes de cada classe.
- **a diminuição na perda da informações explícitas**: o uso dos mecanismos BIT (métodos relatores e assertivas) possibilita o aumento de parâmetros de saída, já que permite verificar informações internas durante a execução do programa; por exemplo, as assertivas produzem mensagens quando alguma restrição imposta é violada. Além disso, os métodos relatores armazenam valores de atributos em arquivos texto.

Na seção seguinte são apresentados alguns trabalhos correlatos que abordam problemas relacionados a melhoria da Testabilidade em Sistemas Orientados a Objetos. No final deste Capítulo, uma comparação é feita entre os trabalhos apresentados e a metodologia usada nesta dissertação.

2.4 Trabalhos Correlatos

2.4.1 Voas et. al.

A seguir são apresentados trabalhos feitos por Voas et. al. [Voa98, RSTC95] relacionados a testabilidade de sistemas OO. Os artigos usam os conceitos da *análise de sensibilidade* já apresentados na seção 2.2.1.

Neste artigo [RSTC95] é apresentado um trabalho feito sobre a testabilidade de sistemas orientados a objetos, abordando a testabilidade da seguinte forma:

1. verificou empiricamente as diferenças entre os paradigmas OO e procedimental em termos de testabilidade;
2. pesquisou como a testabilidade de Sistemas OO poderia ser melhorada, através do uso de assertivas;
3. definiu características importantes para o desenvolvimento de um protótipo para assegurar a testabilidade de sistemas OO.

O primeiro objetivo foi verificar empiricamente se a escolha do paradigma de programação afeta a testabilidade de um sistema. Para quantificar esta diferença, um mesmo programa foi desenvolvido em C e C++, a análise de sensibilidade foi feita usando o protótipo *PiSCES Software Analysis Toolkit*, para fazer a comparação entre a testabilidade das duas implementações. Como resultado, verificou-se que o encapsulamento e as informações escondidas possuíam impacto negativo sobre a testabilidade do programa OO.

No segundo estudo procurou-se melhorar a testabilidade de programas OO, introduzindo-se *assertivas*. As assertivas são mecanismos de validação que testam o estado de entrada ou saída de um programa em execução, com o objetivo de melhorar a sua observabilidade. Para isto, uma análise de propagação foi feita, utilizando a ferramenta *PiSCES* para determinar onde as assertivas seriam inseridas.

A partir dos dois primeiros estudos, notou-se que a inserção de assertivas de forma estratégica é um mecanismo poderoso para melhorar a capacidade de detecção de falhas. Como resultado foi proposta a construção de um protótipo com as seguintes características:

1. a habilidade para ler as saídas da ferramenta *PiSCES,* para determinar onde as assertivas seriam inseridas;
2. uma meta-linguagem interativa, para a especificação das pré e pós-condições para determinar quando um estado deve ser aceito ou não;
3. a habilidade para ler as condições escritas nesta meta-linguagem e converter em código executável, automaticamente inserido no código fonte antes dos testes.

Dando continuidade ao trabalho, em [Voa98] é apresentado a ferramenta ASSERT++, que utiliza os escores de testabilidade obtidos através da análise de sensibilidade do programa, para recomendar e auxiliar a inserção de assertivas. A ferramenta é composta de dois utilitários: (1) um utilitário que dá suporte à inserção das assertivas; (2) e um utilitário chamado *CBrowse*, que recomenda onde assertivas devem ser colocadas baseadas nos escores obtidos.

2.4.2 Wang et. al.

Wang et. al. [Wan99] apresentam o uso de mecanismos embutidos BIT (do inglês, *"Built-in Test"*) na manutenção de sistemas de software baseados em componentes. Os testes embutidos são incorporados ao componente sob teste (diretamente no código fonte) através de funções membros, como mostra a Figura 2.2.

```
Classe nome_de_classe {
                    // interface
                    declaração de dados
                    declaração do construtor
                    declaração das funções;
                    declaração de testes // mecanismos BIT
                    // implementação
                    construtor
                    destrutor
                    funções
                    casos de testes // casos de testes BIT
          }         BITobjeto
```

Figura 2.2 Objeto com mecanismos BIT

Os mecanismos BIT possuem dois tipos de execução: **normal e de manutenção/teste.** Na execução normal, o mecanismo BIT deve ter o mesmo comportamento do componente original, e isto acontece quando apenas funções membros

normais são chamadas. Na execução em modo de manutenção e teste, os mecanismos BIT são ativados através de chamadas das funções membros BIT incorporadas ao componente.

Além de possibilitar o uso destes mecanismos toda vez que houver a necessidade de reteste e manutenção, existe ainda a possibilidade de reuso e herança destes mecanismos pelas classes derivadas. Isto porque os mecanismos são embutidos nas classes como funções membros, assim se uma nova classe for desenvolvida, ela pode herdar diretamente os mecanismos BIT como funções membros normais. Existe apenas a necessidade de incorporar novos requisitos de testes BIT à classe derivada para testar as alterações feitas.

2.4.3 Traon et. al.

Traon et. al. [Tra00] apresentam uma abordagem para testes de integração e testes de regressão de sistema, utilizando o conceito de autoteste e de dependência entre os componentes (classes). Além disso, sugerem o uso de mutantes para a obtenção de métricas de qualidade para verificar a eficácia do conjunto de teste gerado.

O autoteste é introduzido no componente através de uma relação de herança feita entre este e uma classe *autoteste* criada, de forma a introduzir rotinas de testes que deverão exercitar todos os seus métodos. Estas rotinas têm o objetivo de testar se a implementação de cada método corresponde à sua especificação. Para a geração do oráculo, a solução encontrada foi determinar manualmente o resultado esperado para cada requisito de teste criado. Assim, o testador fica responsável por criar e verificar os resultados dos testes.

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

2.5 Comparação dos Trabalhos

A Tabela 2.1 sintetiza os problemas endereçados pelos trabalhos descritos neste Capítulo, além de mostrar uma comparação entre os trabalhos e a metodologia usada nesta dissertação.

Trabalho	Modelo para teste OO	Ferramenta ou *framework* utilizados	Contribuição para as questões de teste	Contribuição para o trabalho
Voas et. al	--	PiSCES e ASSERT++	Problema de testabilidade: estudo sobre a testabilidade de sistemas OO e inserção de assertivas	O trabalho discute problemas de testabilidade OO, e inserção de assertivas em programas OO
Wang et. al.	---	---	Inserção de mecanismos BIT em componentes de software	O trabalho apresenta uma metodologia para o uso de mecanismos BIT na manutenção de sistemas de software
Traon et. al	---	---	Uso de autoteste nos testes de integração e de regressão	O trabalho apresenta uma abordagem para o uso do autoteste, e o uso de mutante como métrica de qualidade
Este trabalho	MFT	**Concat**	testes de classes utilizando o conceito de autoteste e reuso de teste.	--X--

Tabela 2.1 Comparação entre os trabalhos

Capítulo 3

Aplicando os Conceitos de Autoteste em Sistemas OO

No artigo a seguir, descrevemos a metodologia e os passos para a construção de classes autotestáveis. Para tal, foi utilizado como exemplo classes da Biblioteca MFC (do inglês, *Microsoft Foundation Classes).*

Este artigo foi submetido ao ISSTA 2000 – International Symposium on Software Testing and Analysis 2000, evento realizado de 21 – 25 de agosto de 2000, em Portland, Oregon.

Applying the Self-Testing Concepts to OO Systems

Rosileny Lie Yanagawa
lie@dcc.unicamp.br

Eliane Martins
eliane@dcc.unicamp.br

Institute of Computing - IC
State University of Campinas-UNICAMP
CP 6176 - Cep 13081-970
Campinas-SP, Brazil
Fax (019) 788-5847

Abstract

This paper presents the use of *Design for Testability* techniques, used in VLSI (Very Large-Scale Integration) manufacturing, to increase the testability of Object Oriented (OO) Systems. The use of the *Built-in Self-Test* concept allows the building of self-testing classes, with the aim of reducing the effort of re-testing a class when it is reused or maintained. Moreover, the use of the *Hierarchical Incremental Technique* also allows the reuse to be extended to the tests. This text presents a methodology for development and use of self-testing classes. Preliminary results of the proposed methodology are also presented.

Keywords: Design for Testability, Self-testing Classes, BIST and BIT features

3.1 Introduction

The primary motivation for using OO paradigm is its potential for reuse. Reusability can reduce software development cost, but for this to occur, the reusable components must present high quality. Testing is one of the techniques that is used to guarantee a good quality level.

Tests for a component must be repeated many times in its lifecycle: during development and after each change (in the component or in the environment), since new faults could be introduced. This requirement for repeatability implies that costs and effort

for test application should be reduced; in other words, a reusable component should have a good testability.

Testability has been given considerable attention in the design and manufacturing of VLSI devices. *Design for Testability* (DFT) techniques have been used to improve *controllability* and *observability* of hardware components: to effectively test a component, one must be able to control its inputs (and internal state) and observe its outputs. *Built-in test* (BIT) capabilities can be used for that purpose. By extending these capabilities with the self-testing concept, a *Built-in Self Testing* (BIST) component is obtained, having the ability of generating test patterns for embedded chip components and capturing its outputs, thus reducing the need of external testers, which are expensive.

Design for testability and the self-testing concepts can also be used to reduce cost and time for software testing. In this text, we show the application of these techniques to improve class testability, as proposed in [Bin94]. To develop and use a self-testing class, the following tasks have to be performed:

1. Development of the class model that will be used for test generation;
2. Instrumentation of the class to include built-in test features;
3. Generation of a driver, which applies test cases to a class;
4. Development of an oracle, a reference to be used for test results analysis.

A prototype called Concat [Toy98] was used in support to tasks 2 and 3. By automating driver generation, test costs can be reduced [Kun95], making class reuse easier.

But reuse can not be limited to a class: test suites can also be reused whenever a class is instantiated or modifications are introduced in the class itself or in its ancestors. Use of the *Hierarchical Incremental Testing* (HIT) approach, proposed in [Har92], allows a test sequence for a parent class to be reused, whenever possible, when testing one of its subclasses.

The paper is organized as follows: Section 3.2 presents the concepts for OO testability; Section 3.3 presents some related works; Section 3.4 and 3.5 explain the methodology in detail, using as example a class from the *Microsoft Foundation Class (MFC)* library, CObList, and one derived class. Section 3.6 summarizes the results of applying the self-testing concepts to other classes of the aforementioned library. Section 3.7 shows the conclusion and future work.

3.2 Software Testing and Testability Concepts

Testing is the validation activity that consists of exercising software, providing inputs, in order to find faults. Basically, the testing process is composed of the following activities: the generation of test cases, their execution in the program under test, and the evaluation to check if the observed output is similar to the expected one.

In test case generation, the tester should select inputs that are well suited to reveal existing, but unknown, faults. Since it is not possible to submit the software to the entirety of its input domain, a criteria is required to select a subset of this domain. *Test criteria* are related to a model of the software under test. Most common *test models* are based on software functionality (functional or black-box testing) or on the structure of program source code (structural or white-box testing). How hard it is to establish the test criteria and to select inputs that satisfy the tests depend on how testable the software is.

The IEEE *Standard Glossary of Software Engineering Terminology* defines testability as: (1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met; and (2) the degree to which a requirement is represented so as to allow the establishment of test criteria and carrying out tests to determine whether those criteria have been met.

There are two important aspects for software with good testability [Hof89]: controllability and observability. Observability refers to the ease to determine how much data inputs affect the obtained outputs. Controllability refers to the ease to produce the expected output from data input. In component testing, one should be able to control the inputs (controllability) and internal behavior, as well as observe the outputs (observability). In general, it is difficult to obtain good controllability and observability, specially if the component is embedded in other components. In this aspect, the intrinsic features of OO systems present some obstacles for testability, as will be seen next.

3.3 Previous Work

Testing OO systems is a growing area. In this section, we will present only an overview of some previous work, mainly, those relative to class testing, which is our concern.

Functional testing approaches have been investigated by Doong and Frankl, which test classes based on algebraic specifications. The FREE methodology, proposed in [Bin96] uses a state based model for class testing. This model can be extended with dataflow information, giving the FREE flowgraph, thus allowing to mix functional and structural testing methods. Kung et al [Kun95] developed a test approach which involves 3 models: (a) *the object relation diagram* (ORD), which presents inheritance, aggregation, association, template class instantiation, use and nested-in relationships among classes; (b) *the block branch diagram* (BBD), presenting the control and data flow structure of a method; and (c) *the object state diagram* (OSD), which models the object state dependent behavior. The testgraph methodology [Hof95] proposes a graph corresponding also to a state transition diagram of a class under test.

Reuse of test cases is addressed by the *Hierarchical Incremental Testing* (HIT) approach, proposed in [Har92]. Reuse is based on the inheritance hierarchy: tests for a parent class can be used to reduce effort in testing each subclass. The approach is used in this study and will be further explained in the next section.

In relation to the design for testability of OO Systems, we can mention the work of [Voa98], which proposes the use of assertions to enhance controllability and observability of OO programs.

In this study, we use the DFT approach proposed in [Bin94], combining the use of embedded assertions and test access methods with test generation capabilities, leading to development and use of self-testing classes. These concepts are presented in the next section.

3.4 The Proposed Approach

The aim of our approach is to reduce efforts when retesting a class each time it is reused or modified. This goal can be achieved, following these principles: (a) use of Design for Testability (DFT) strategies; (b) reusability of test suites and (c) automation of test activities. A brief discussion of each of these points is given, followed by a presentation of the methodology for building and using self-testing classes.

3.4.1 DFT Strategies

In [Bin94], the use of DFT and self-testing concepts used in VLSI components are proposed for the class level. Different configurations are presented, and the one used here is shown in Figure 3.1.

According to Figure 3.1, the following components are required to have BIST in OO systems:

- a *self-testing class*, which is composed by the *class under test* (CUT) augmented by a test specification and BIT features. These are composed by reporter methods (used to report the internal state of an objet) and assertions (representing pre and post conditions for methods as well as class invariants). Differently from hardware these features need not to remain with the class when it is in normal operation: by carefully using compiler directives, embedded testing features could be included or excluded from the system. The assertions, however, could remain in the normal operation phase, to continue to provide self-checking capabilities;

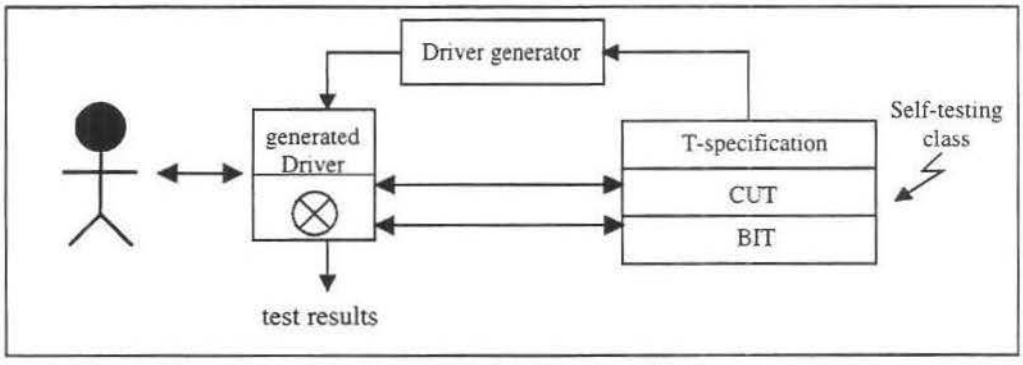

Figure 3.1 Self-Testing Class

- *a driver generator*, which uses the test specification to generate the driver that will activate and control the CUT during test execution;
- *a generated (or specific) driver*, which corresponds to an executable test suite, generated according to test criteria implemented by the driver generator, applied to a test model represented by the test specification. Besides activation and control of the CUT during tests, this driver also evaluates tests. Test evaluation is based on CUT state, given by BIT features, and CUT outputs.

3.4.2 Reusability of Generated Tests

As already mentioned, tests should be repeated each time a class is reused or modified, so it is important to maintain not only the class code, but also its tests. The use of a test specification in a self-testing class already provides this, thus facilitating class reuse and maintenance by reducing retesting effort.

To reduce test effort a bit more, it would be useful to reuse generated tests whenever possible. In fact, there are circumstances in which this possibility exists. For example, when testing a subclass one could reuse tests generated for those parent class features, which are not changed in the subclass.

For that purpose, the HIT approach proposed by *Harrold et.al.* [Har92] was considered in this study. This approach exploits the inheritance hierarchy by reusing test information for a parent class to guide the testing of a subclass. Each class has a *testing history,* which associates each test case with the elements it tests (a single method or interactions among methods).

The testing history is incrementally updated to reflect the modifications introduced by the subclass in its parent features. With this testing history, it is possible to identify which subclass features should be tested: new features (those defined in the subclass features but not in its parents), or those redefined (modified) by the subclass or yet those unmodified inherited features that interact with new or redefined ones. It is also possible to identify when new tests should be generated or when a parent class test case can be reused to validate the subclass. This approach imposes some design and implementation constraints, such as: (a) each class should have only one parent class (i.e., multiple inheritance is not allowed); (b) only strict inheritance model is accepted, in which a subclass takes all features from its parent (possible re-implementing some of them) as a subset of its own features. In other words, a subclass is a subtype compatible with its parents class such that it can take the place of such parent in polymorphs substitutions; (c) it is oriented for C++ implementations.

3.4.3 Test Automation

A prototyping tool, *Concat,* was developed to assist the user in building and testing self-testing classes. *Concat* gives support to the following tasks [Toy98]:

- *test case generation:* the driver generator creates test cases based on test specification describing a functional model of the CUT, called transaction flow model [Bei90,Sie96]. This model, which is explained in Section 3.6, represents the external behavior of a class. The test specification also describes the driver to generate test case values.

- *test case execution:* the generated driver consists of test cases, which are calls to the CUT methods, and also to BIT features. Arguments of non-scalar types, such as objects, structures, pointers or arrays, must be completed by the user. Instantiation of global data and stubs should also be completed manually before running the program, implementing the generated driver;
- *test case evaluation:* this step is based on the assertions that is part of BIT features, more precisely, methods post-conditions and class invariant. The driver can check the correctness of CUT state during testing. For the moment, comparing the obtained output with the expected ones should be done manually.

3.5 Testing Methodology

Figure 3.2 presents an overview of the steps that should be carried out in building and using a self-testing class, and they are explained in the next items. The activities inside a rounded rectangle should be carried out by the user; and the activities inside a rectangle are automatically performed by the *Concat* tool.

First, we present two classes, a parent and one derived class, used to illustrate the methodology.

3.5.1 CObList and CSortableObList Classes

The parent class, CObList, is one component of the MFC library (*Microsoft Foundation Classes*). This class represents a linked list whose data items are of CObject pointer type; Figure 3.3 presents its C++ declaration.

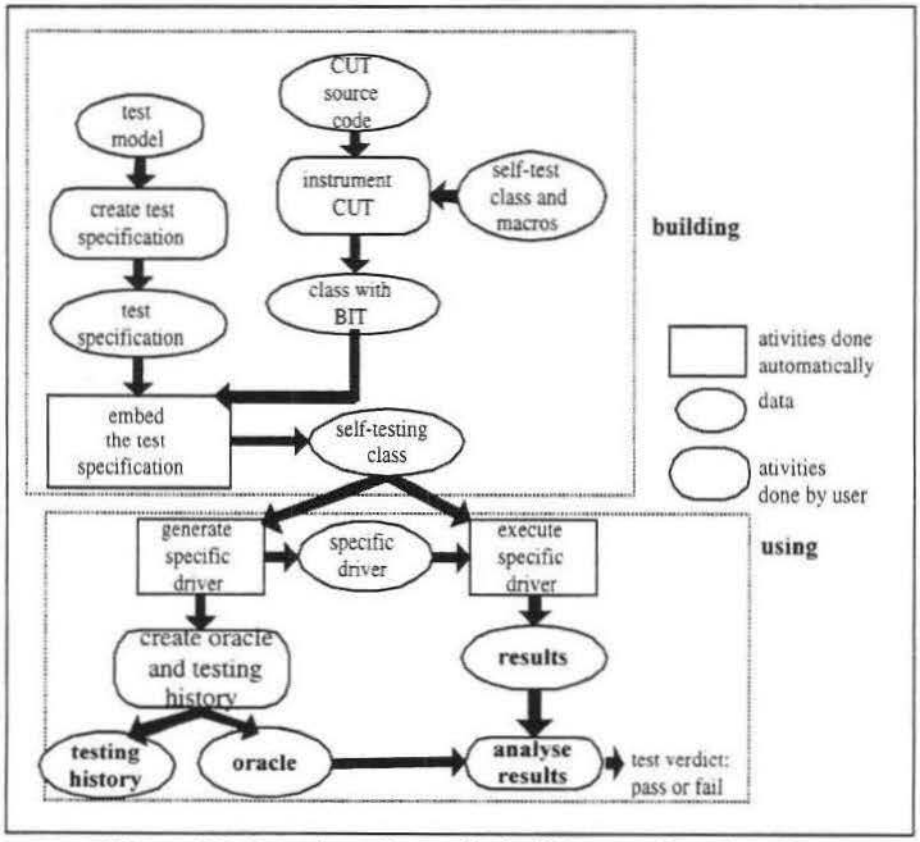

Figure 3.2 Overview of steps for building a self-testing class

The derived class CSortableObList[1] created to illustrate subclass testing, implements an ordered list of CObject type. Figure 3.4 shows its C++ declaration; new methods were developed and added to this class, besides those contained in the original one, to further illustrate the test approach. For example, to obtain the least element from a list, or to find an element in a given position, are added methods.

3.5.2 The Test Model

The test model used in this work was the *Transaction Flow Model* (TFM). This is a functional model, defined by Beizer [Bei90] to test concurrent systems, and adapted by Siegel [Sie96] to test objects. The model illustrates the different ways of creating an object, different tasks or processes that an object can do and different ways of destroying an object [Sie96].

[1] This class was obtained on the Internet (In: www.codeguru.com)

```
Class CObList : public Cobject
{
  CObList(int nBlockSize = 10);     // Constructs an empty list for CObject pointers.
  int GetCount() const;          // Returns the number of elements in this list.
  BOOL IsEmpty() const;      // Tests for the empty list condition (no elements).
  CObject*& GetHead();     // Returns the head element of the list (cannot be empty).
  CObject*& GetTail();     // Returns the tail element of the list (cannot be empty).
  CObject* RemoveHead();    // Removes the element from the head of the list.
  CObject* RemoveTail();   // Removes the element from the tail of the list.
  Void AddHead(CObList* pNewList);  // Adds an element (or an list) to the head of the list
  Void AddTail(CObList* pNewList);          // Adds an element (or an list) to the tail of the list
  Void RemoveAll();                         // Removes all the elements from this list.
  POSITION GetHeadPosition() const;         // Returns the position of the head element of the list.
  POSITION GetTailPosition() const;         // Returns the position of the tail element of the list.
  CObject* GetNext(POSITION& rPosition) const;          // Gets the next element for iterating.
  CObject* GetPrev(POSITION& rPosition) const;          // Gets the previous element for iterating.
  CObject* GetAt(POSITION position) const;              // Gets the element at a given position.
  void SetAt(POSITION pos, CObject* newElement);        // Sets the element at a given position.
  void RemoveAt(POSITION position);             // Removes an element from this list, specified by
                                                //position.
  POSITION InsertBefore(POSITION position, Cobject* newElement);      // Inserts a new element
                                                // before given position.
  POSITION InsertAfter(POSITION position, CObject* newElement);// Inserts a new element after
                                                                // a given position
  POSITION Find(Cobject* searchValue, POSITION startAfter = NULL) const;  // Gets the
                                                                // position of an element specified
                                                                // by pointer value.
  POSITION FindIndex(int nIndex) const;  // Gets the element position specified by a zero-based
                                                                // index.
  ~CobList();                                   // destructor
        }
```

Figure 3.3 Member functions of the CObList class

```
class CsortableObList : public CobList
{
public:
CsortableObList(int nBlockSize = 10);  // Constructs an empty list for CObject pointers.
 Virtual void Sort(int(*CompareFunc)(Cobject* pFirstObj, Cobject*pSecondObj));
                                        // Sort an list of the elements
 Virtual void Sort(POSITION posStart, int iElements, int (*CompareFunc)(CObject* pFirstObj,
Cobject* pSecondObj));          // This variation allows you to sort only a portion of the list
 Virtual void ShellSort(int (*CompareFunc)(CObject* pFirstObj, CObject* pSecondObj));
                                        // sort the list using Shell Sort algoritm
 virtual POSITION FindMax(int (*CompareFunc)(Cobject* pFirstObj, CObject*
pSecondObj));                   // Gets the position of biggest element in a list
 virtual POSITION FindMin(int (*CompareFunc)(Cobject* pFirstObj, CObject* pSecondObj));
                                        // Gets the position of smallest element in a list
CObject* FindElement(int position);     // Gets the element at a given number position
}
```

Figure 3.4 Member function of the CSortableObList Class

A *transaction* represents an objects life cycle, from the creation until its destruction. This way, each transaction starts from the creation of an object, proceeds with the object performing a set of tasks or processes, and finishes with the destruction of the object. The model is given by a transaction flow graph, where the *edges* represent valid sequence of methods and the *nodes* represent processes or tasks of interest, such as a method or a set of methods.

The tester builds the transaction flow model, based on *use-cases* or data-flow model (DFD) of the class. One example of a *use-case* for the CObList class is:

a) create a CObList list; b) insert a new object in the initial position on the list; c) retrieve the initial position on the list; d) insert a new object on the list head; e) find an element in the list; f) remove an element from a determined position g) and destroy the CObList list;

The transaction flow model, shown in Figure 3.5, represents several *use-cases*. A transaction is a path in this graph, beginning in the start node (where the object is created) and ending in an end node (where the object is destroyed). For example, a path in the

graph corresponding to the *use-case* mentioned above would be: (CObList, AddHead, GetHeadPosition, InsertBefore, Find, RemoveAt, ~CObList), shown in Figure 3.5.

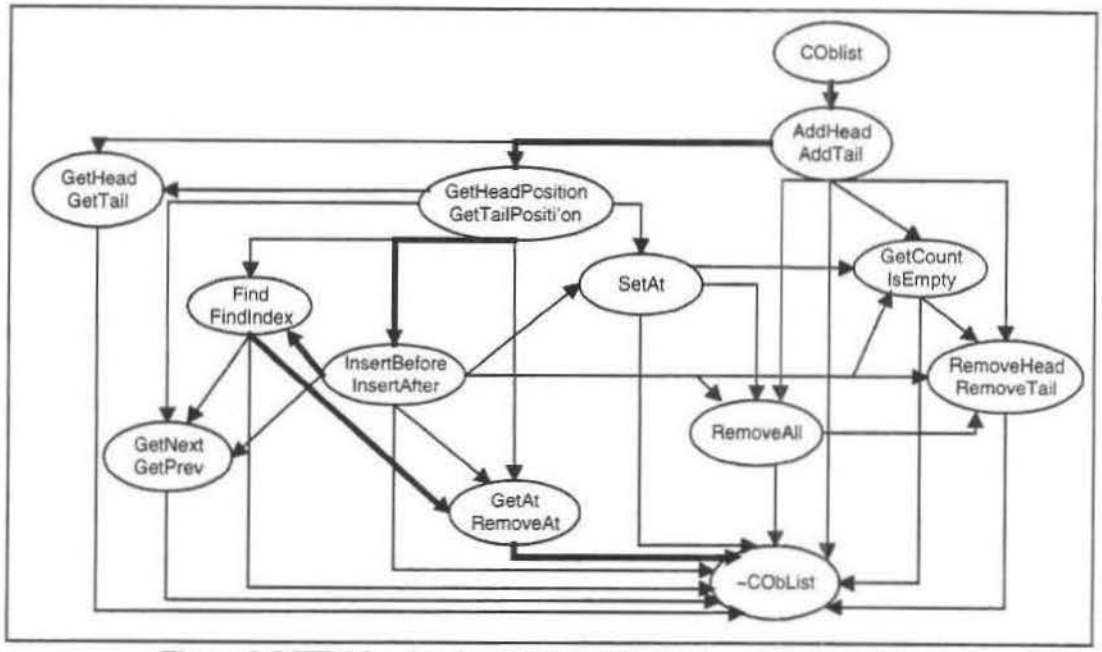

Figure 3.5 TFM for the class ObList; illustration of the test object example

The next phase consists in building the test model for the CSortableObList derived class, illustrated in Figure 3.6, using based on the test model of the CObList class. With the HIT approach, only strict inheritance is permitted, which means that the test model of the subclass should describe the same behavior as that of the base class. Therefore, the derived class neither omits inherited methods, nor changes the execution order of these methods. New methods can either be introduced in previously existing nodes, or in new nodes. New links can be created either for these new methods/nodes, or for redefined ones.

In this example, it can be observed that some methods of the chosen classes do not have a pre-determined execution sequence, which can cause *loop* occurrence. For example, after removing an object, another element can be added to the head of the list through the AddHead method. To reflect this situation, the model should have an edge from the RemoveHead/RemoveTail node to AddHead/AddTail node, thus forming a cycle. It is also possible to repeat certain operations several times, causing the generation

of *self-loops*. For example, several elements can be inserted in the list, which would be represented by a *self-loop* in the AddHead/AddTail node.

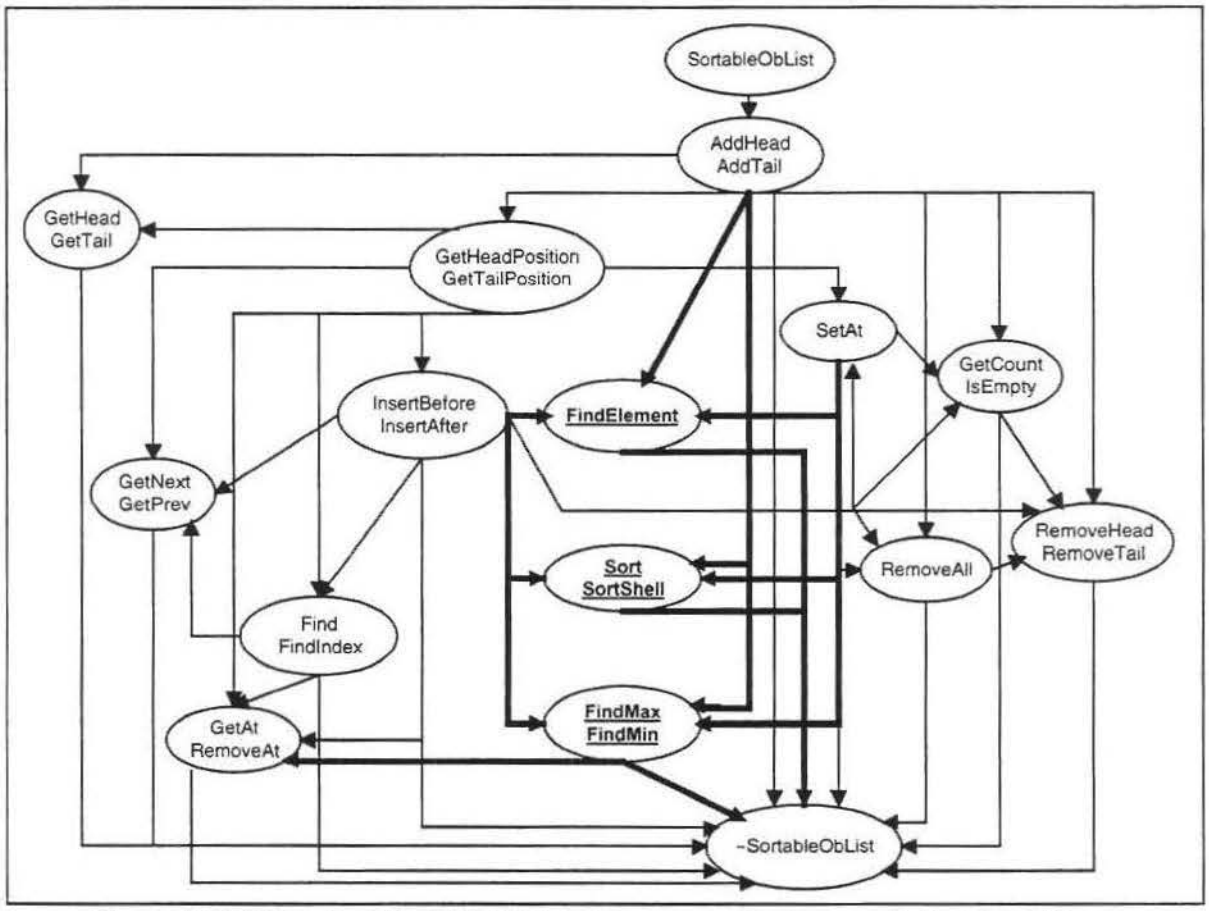

Figure 3.6 TFM to the CSortableObList class, new methods are underlined, and the new connections are highlighted

This kind of class is defined by Binder [Bin96] as a *non-modal class*, because it does not place any constraints on the acceptable sequence of method execution. According to [Bei90], if loops can not be avoided, *loop* testing should be used together with the transaction testing method. This was not done in this work, because the number of tests would be very large to be carried out manually, as the tool used does not give support to this testing method.

3.5.3 The Test Specification

Based on the model created in the previous phase, the test specification for each class chosen was created, according to the format indicated in Figure 3.7.

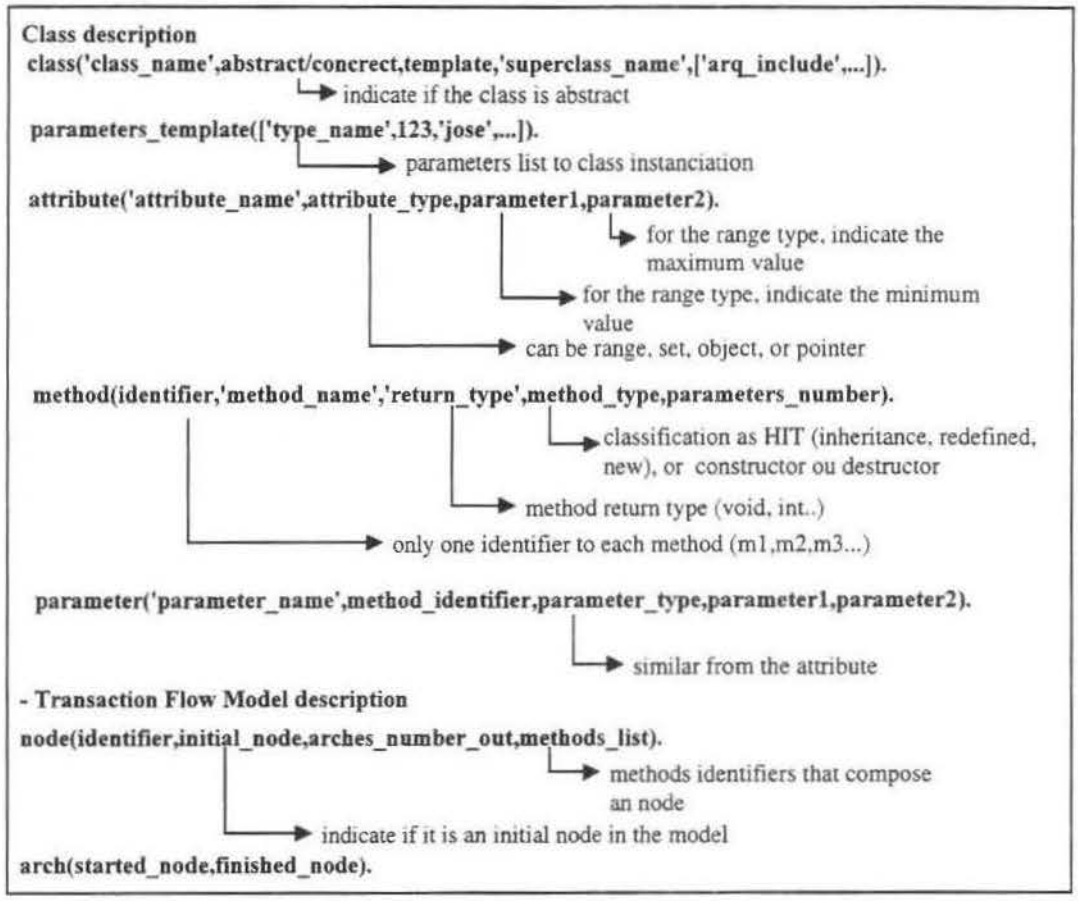

Figure 3.7 Test specification format

Besides methods and attributes names, types and input domain (range of possible values for method's parameters or attributes), the user should also indicate a classification (new, inherited or redefined) for each method, so that the HIT approach can be used in test generation. The test specification for the CObList base class is created and can be reused to create the test specification for the CSortableObList derived class. Part of the test specification for the CSortableObList class is shown in Figure 3.8.

The specification is stored in a file in the same directory as the CUT, this file has the same CUT name and ".mt" extension. In this way, it is possible for the *Concat* tool to use this specification for test case generation.

```
Class ('SortableObList',n,_,[]).
attribute('Cnode',object,'CNode',_).
attribute('m_pNodeHead',ponter,'CNode*',_)
attribute('m_ncount',integer_range,100,1).
....
method(m1,'SortableObList',_,constructor,1).
method(m2,'~SortableObList',_,destructor,0).
method(m3,'GetCount','int',inheriance,0).
method(m4,'IsEmpty','BOOL',inheritance,0).
...
method(m25,'FindMax','POSITION',new,1).
method(m26,'FindMin','POSITION',new,1).
....
```

```
parameter('nBlockSize',m1,intervalo_int,100,1).
parameter('newElement',m10,pointer,'Cobject*',_)
.
parameter('pNewList',m11,object,'CObList*',_).
parameter('newElement',m12,pointer,'Cobject*',_)
.
...
node(n1,yes,1,[m1]).
node(n2,yes,1,[m3,m4]).
node(n3,yes,9,[m10,m11,m12,m13]).
...
edge(n1,n3).
edge(n2,n13).
edge(n3,n2).
edge(n3,n5)
```

Figure 3.8 Part of test specification of CSortableObList class

3.5.4 Class Instrumentation

The class should be instrumented in order to introduce the *Built-in Test* features, described in 3.5.1. The Figure 3.9 shows CSortableObList class instrumented with such features, and Figure 3.10 presents the implementation of a reporter method for this same class.

As already noted, the CObList class from MFC library contains embedded assertions, so, only the assertions for the CSortableObList class were created.

```
POSITION CsortableObList::FindMax(int
(*CompareFunc) (Cobject* pFirstObj,
Cobject* pSecondObj)) {
  CObject* max;
  CObject* prox_element;
  int index;
  int Number_Total_Methods;
  POSITION pos;
 ASSERT_VALID(this);
  Assert (m_pNodeHead != NULL);
 // the list can not be empty
```

```
max=GetHead();
Number_Total_Methods=m_nCount;
for(index=1;index<Number_Total_Methods;index++)
{
   Pos=FindIndex(index);
   Assert (pos!=NULL);
   // the element position cannot be null
   prox_element=GetAt(pos);
   if (CompareFunc (prox_element,max) < 0)
max=prox_element;      };
 Assert(max!=NULL);// it must return a not null
value
 Return Find(max);      }
```

Figure 3.9 Example of assertions introduction in the tested methods of CSortableObList class

```
Void Partial_Report (CString method, char*
file, void* object) {
 POSITION position;
 CsortableObList *list=new CSortableObList;
 list=(CsortableObList*) object;
 CMyObject *contend=new CMyObject;
 // Open results file
 Ofstream results (file,ios::app);
 if (!results) {
  cout << "Error opening file!" << endl;
  exit(0); };
 results << "->Method executed::" <<
method<<"\n";
int n=list->GetCount();
 if (n==0) {
results << "-->list content is empty after the
method execution" << "\n"; } //show list content
 else {
  results << "-->Existent content in the list after
the method execution" << ":";
   position=list->GetHeadPosition( );
   while (pos!=NULL){
    content= (CmyObject*)list->GetNext(pos);
    results << content->name << " ";     }
    results << "\n";  }
 results.flush(); }
```

Figure 3.10 Example of reporter method implementation

3.5.5 Generating the specific driver

Part of this step is done automatically by the *Concat* tool. This sets up part of the *specific driver (*test sequence), as the user should complete this sequence with structured type parameters (*structs, objects, arrays types)*, as well as global data and stubs. The *driver generator* implemented by the Concat tool produces test cases by traversing the transaction flow model beginning from the start node. The criteria requires to cover all paths, corresponding to all possible transactions. As in structural testing of programs, path coverage is difficult (or even impossible) to achieve, especially if the model contain cycles: that is why they must be avoided. The test cases corresponding to the transaction showed in section 3.5.2 is illustrated in Figure 3.11.

3.5.6 Developing the oracle

In addition to the use of assertions, which are only partial oracles, another reference is needed as a complement, in order to improve fault detection capability. Analysis of the results was carried out manually, using the output generated by the reporter method. A partial representation of the output file is shown in Figure 3.12. For example, in test case 1, after the execution of the AddHead method, the list had one element: Lie.

```
Void Test_case12_1(CObList* cst) {
  char* met= new char[30];
.....
try {
    cst->Invariant_Test();
    met = "AddHead(newelement)";
    cst->AddHead(newelement);
    cst->Invariant_Test();
    met = "GetHeadPosition()";
    pos=cst->GetHeadPosition();
    cst->Invariant_Test();
    met = "InsertBefore(pos,newelement)";
    cst->InsertBefore(pos,newelement);
    cst->Invariant_Test();
    met = "Find(searchValue,pos)";
    cst->Find(searchValue,pos);
    cst->Invariant_Test();
    met = "RemoveAt(pos)";
   cst->RemoveAt(pos);
    cst->Invariant_Test();
are << "Case_Test12_1 OK!\n";
    arq.flush();
object=cst;
cst->Reporter("Result.txt",object);
    arq << "\n";
    arq.close();
    delete cst;  }
......
```

Figure 3.11 Example of a test case

```
Test_case1_0
 ->Method executed:AddHead(newelement)
 -->Existent content in the list after method
 execution:Lie
 ->Method
 executed:FindMax(CMyObject::CompBackward)
 -->Existent content in the list after method
 execution:Lie
 ---> returned value:Lie
 Test_case2_0
 ->Method executed:AddHead(newelement)
-->Existent content in the list after method
execution:Lie
 .....
 Test_case231_1
 ->Method executed:AddTail(newelement)
-->Existent content in the list after method
execution:Andre
 ->Method executed:GetTailPosition()
 -->Existent content in the list after method
 execution:Andre
 ->Method executed:SetAt(pos,newelement1)
 -->Existent content in the list after method
 execution:Ana
 ->Method executed:AddHead(newelement2)
 -->Existent content in the list after method
 execution:Lucia Ana
 ->Method executed:AddHead(newelement3)
 -->Existent content in the list after method
 execution:Alice Lucia Ana
```

Figure 3.12 Part of partial results file obtained from reporter method, showing internal state of an object of CSortableObList class

3.5.7 Building the history

As viewed in 3.4.2, a testing history should be created for each class when using the HIT approach. The testing history is created when tests are generated for parent class, and inherited by its children. In this study, a testing history contains test cases ready for execution, that is, after parameterization has taken place. This way, it is possible to reuse a test case without any further manual intervention. A utility program was specially created for that purpose, since testing history generated by Concat tool does not contain test cases ready for execution. Figure 3.13 shows a test case as generated by Concat and after the execution of the utility program.

3.6 Testing Results

Besides the classes presented, the methodology has been applied to other MFC classes, presented in Table 3.1. A summary of the obtained results is shown in Table 3.2. In the CSortableObList, only test cases with some new or redefined methods are generated.

```
Void Test_case2_1(Type* cst) {
.....
cst->Test_Invariant();
met = "AddHead(Parameter should be a pointer to
CObject*)";
cst->AddHead(Parameter should be a pointer to
CObject*);
cst->Invariant_Test();
met = "FindMax( Parameter should be an object of the
type CompBackward)";
cst->FindMax(Parameter should be an object of the
type CompBackward);
cst->Test_Invariant();
..... delete cst;     }
```

(a)

```
void Test_case2_1( Type* cst) {
.....
cst->Test_Invariant();
met = "AddHead(newelement)";
cst->AddHead(newelement);
cst->Invariant_Test();
met =
"FindMax(CMyObject::CompBackward)";
cst->FindMax(CmyObject::CompBackward);
cst->Test_Invariant();
....
delete cst;       }
```

(b)

Figure 3.13 Testing history; (a) a test case as generated by *Concat;* part (b) a test case after utility program

The low number of faults detected can be explained by the fact that the tested classes are indeed components of a previously tested library. The faults were found in loops of methods and they were detected through analysis of the output generated by reporter methods, since no assertion violation occurred. This reinforces the need of an oracle in complement to the embedded assertions.

Although this study has detected a low number of faults, another work [Yan00] shows empirically evidences that this strategy used is efficient.

Class	**Description**
CMapStringToPtr	Support *map* between **void** pointer and **CString** object
CPtrList	Implement an pointer list of the Type **void**
CObList	Support a pointer list of the Type **CObject**
Cfile	Implement the manipulation of data in files

Table 3.1 Classes tested from the MFC library.

Class	# of methods of the class	# of test cases generated	# failures found
CMapStringToPtr	11	165	0
CPtrList	22	329	0
CobList	22	329	0
CFile	28	435	0
CSortableObList	8	230	2

Table 3.2 Obtained results of the carried out testing

3.7 Conclusion and Future Works

This study shows the use of Design for Testability techniques and the self-testing concept, commonly employed in hardware testing, in the context of OO systems. Our goal is to ease the testing of reusable classes, contributing to improve quality of systems developed from these classes.

For achieving this goal, we propose a methodology for creating and using self-testing classes. These classes are composed by a class under test augmented with embedded assertions, state report methods and a test specification, allowing tests to be semi-automatically generated, executed and evaluated. The use of the hierarchical incremental testing approach in test generation, also contributes to reduce retesting efforts, since it allows to reuse testing for a parent class, when its subclasses are tested.

In this paper, we present the results of using this approach in the tests of a commercial class library. Our next step is to use that approach for testing a real application that reuses some classes of this library.

In the long term, we envisage the following works:

- extension of the prototyping tool already existent to support other tasks. One of the most crucial ones is automatic oracle generation, as results obtained already have shown their importance in complementing assertions for fault detection.
- a better understanding of the costs involved in testing a system developed from reusable classes to assess the impact of using self-testing class;
- extension of the self-testing concept to a group of classes (cluster or frameworks), thus improving integration testing of systems.

Capítulo 4

Avaliação Empírica da Eficácia dos Testes Baseados no Modelo de Fluxo de Transação em Sistemas Orientados a Objetos

Este artigo descreve uma avaliação empírica feita para determinar se os testes baseados no modelo de fluxo de transação utilizado neste trabalho, têm um bom potencial para encontrar falhas. Para isto, duas classes foram escolhidas para os testes: uma classe da biblioteca MFC (*Microsoft Foundation Classes);* e uma derivada da classe escolhida. O artigo apresenta de forma detalhada a estratégia utilizada para gerar os testes, o procedimento usado para criar os mutantes e os resultados obtidos.

Este artigo foi submetido e aceito ao XI CITS: QS – Conferência Internacional de Tecnologia de Software: Qualidade de Software, evento realizado nos dias 18 – 23 de junho de 2000 em Curitiba.

Avaliação Empírica da Eficácia dos Testes baseados no Modelo de Fluxo de Transação em Sistemas Orientados a Objetos[2]

Rosileny Lie Yanagawa
lie@dcc.unicamp.br

Eliane Martins
eliane@dcc.unicamp.br

Instituto de Computação - IC
Universidade Estadual de Campinas - UNICAMP
CP 6176 Cep 13081-970 Campinas-SP, Brazil
Fax (019) 788-5847

Abstract

One main advantage of the O-O paradigm is its reuse potential. In order to guarantee an adequate reliability level for the system, a reusable component should be thoroughly tested: during development and each time it is reused. Reducing test costs is thus very important. To achieve this cost reduction, the use of self-testing classes is proposed: classes would be delivered with their test specification, which would be a basis for test case generation. This specification represents thus the system model from which test cases are generated. So, one important aspect is the selection of this model. In this study we consider the use of the transaction flow model which represents the different use cases for an object. To assess the fault detection capability of test sets generated from this model an empirical evaluation, based on mutation analysis, was performed. Since the model represents different ways that methods from a class could interact, mutant were generated using interface mutation operators. Two classes were selected for the tests: one from the MFC library (*Microsoft Foundation Classes)* and a subclass of it. A fault was detected if the program terminates abnormally, or an assertion was violated or else, mutant outputs were different from those of the original program. The paper presents the strategy used for test case generation, the procedure used to create the mutants and the obtained results.

PALAVRAS-CHAVE: Teste, Modelo de Fluxo de Transação, Testes Orientado a Objetos, Mutação de Interface, Engenharia de Software.

4.1 Introdução

Uma das grandes vantagens do paradigma Orientado a Objetos é o seu potencial para o reuso, sendo a classe a unidade básica a ser reutilizada. Entretanto, para que a reutilização

---

[2] Este trabalho foi financiado pelo CNPq.

seja bem sucedida, é exigida a alta confiabilidade desses componentes. A realização de testes é uma das formas mais usadas para garantir esta qualidade.

Cada classe reutilizável deve ser devidamente testada: durante o desenvolvimento; ao ser reutilizada; ou ao ser modificada. Devido a grande repetição dos testes, é importante que o esforço para a sua realização seja reduzido. Desta forma, é importante que estes componentes sejam fáceis de serem testados, garantindo assim que as vantagens oferecidas pelo reuso não sejam perdidas na fase dos testes. Portanto, um primeiro passo consistiria em melhorar a testabilidade dos componentes para facilitar a realização dos testes.

O conceito de *testabilidade* pode ser definido através de quão fácil é satisfazer uma meta particular através de critérios de testes estabelecidos, como por exemplo, exercitar todos os laços de um programa. A testabilidade de um componente ou sistema tem influência direta nos esforços gastos na fase de testes. Assim, quanto maior a testabilidade de um componente ou sistema, mais fácil será alcançar os objetivos estabelecidos.

Neste sentido, Binder [Bin94] propõe o uso do projeto visando testabilidade (DFT, do inglês *"Design for Testability"*) e do conceito de autoteste usados em hardware, para a melhoria da testabilidade de sistemas orientados a objetos. Além da capacidade de teste embutido ou BIT (em inglês *"Built-in Test"*), que visa introduzir métodos relatores e assertivas na classe com o objetivo de facilitar o acesso a componentes internos de um objeto, Binder também propõe a introdução da capacidade de geração, execução e análise automática dos testes, tem-se assim o uso do conceito de autoteste nas classes (em inglês *"Built-in-Self Test"* ou BIST).

Para se criar uma classe autotestável é essencial a existência de uma especificação de teste que permita que os requisitos de testes sejam gerados, de acordo com um critério estabelecido. Nesse estudo, a geração dos testes é baseada em um modelo de fluxo de transação, que representa os diferentes cenários de uso de um objeto.

Para medir a eficácia em encontrar falhas dos testes baseados na estratégia de teste usada, uma avaliação empírica foi feita, com base nos operadores de mutação de interface proposta por [Del97]. Neste estudo, duas classes foram utilizadas: a primeira, uma classe da biblioteca MFC (*Microsoft Foundation Classes)*; e a segunda, uma classe derivada da primeira. Uma falha foi considerada detectada se o programa teve um término anormal; se

a saída obtida era diferente do programa original; ou ainda se uma assertiva não era satisfeita. O artigo apresenta de forma detalhada a estratégia utilizada para gerar os testes, o procedimento usado para criar os mutantes e os resultados obtidos.

As seções estão organizadas da seguinte forma; a Seção 4.2 apresenta uma visão geral da testabilidade de sistemas orientados a objetos; a Seção 4.3 apresenta a descrição das classes exemplos; a Seção 4.4 descreve a abordagem de teste utilizada e mostra os conceitos do modelo de fluxo de transação; Seção 4.5 apresenta uma descrição da estratégia utilizada para a avaliação empírica feita e os resultados obtidos; a Seção 4.6 apresenta as conclusões do trabalho.

4.2 Testabilidade de Software Orientados a Objetos

A *testabilidade* de software é a definida pelo padrão IEEE 610.12 como sendo: (1) em que medida um sistema (ou componente) facilita o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos; (2) em que medida um requisito é representado de forma a permitir o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos.

Um software com boa testabilidade deve possuir duas características importantes [Hof89]: boa controlabilidade e boa observabilidade. A observabilidade refere-se à facilidade em se determinar o quanto as entradas fornecidas afetam as saídas obtidas. A controlabilidade refere-se à facilidade de se produzir a saída desejada a partir da entrada fornecida. Assim, para se testar um componente deve ser possível controlar (controlabilidade) suas entradas e estado interno, bem como observar (observabilidade) a sua saída. Em geral é difícil conseguir uma boa controlabilidade e observabilidade, em especial se o componente está embutido em outros componentes.

Neste sentido, existem trabalhos que abordam a testabilidade do software, de forma a construir o software visando sua melhoria. Voas, Schimid e Schatz [Voa98] descrevem uma abordagem e uma ferramenta para encontrar regiões do código que se mostram com maior probabilidade de esconder falhas, com o objetivo de inserir assertivas para aumentar a observabilidade destes componentes. Neste trabalho, além do uso de assertivas sugerido por Binder e que será detalhado na próxima seção, o conceito de autoteste foi também usado para melhorar também a produtividade na geração de testes.

A descrição das classes usadas nos testes e a abordagem utilizada nesse trabalho é apresentada a seguir.

4.3 Classes CObList e CSortableObList

Duas classes foram utilizadas neste estudo. A classe base CObList, que é uma componente da biblioteca *MFC (Microsoft Foundation Classes)*, representa uma lista de ponteiros para objetos do tipo CObject. Esta classe contém métodos para a manipulação de uma lista, tais como: inserir, excluir e encontrar um elemento na lista; encontrar a posição de um elemento; ou contar o total de elementos na lista. A segunda classe CSortableObList, desenvolvida para ilustrar o teste de subclasse, como o nome indica mantém uma lista ordenada. Esta classe foi obtida na Internet[3], porém novos métodos foram adicionados à mesma, como por exemplo: recuperar o menor objeto da lista; ou encontrar um elemento em uma dada posição.

4.4 Abordagem de Teste Utilizada

A abordagem de teste utilizada levou em consideração os aspectos descritos nos itens a seguir.

4.4.1 Projeto para testabilidade e conceito de Autoteste em software OO

Binder [Bin94] aplica as técnicas de projeto visando testabilidade (DFT) usadas em hardware, para a melhoria da testabilidade de sistemas orientados a objetos. Além da capacidade de teste embutido ou BIT, do inglês *"Built-in Test"* para facilitar o acesso a componentes internos de um objeto, propõe também a introdução da capacidade de geração, execução e análise automática dos testes, tendo-se assim a aplicação do conceito de autoteste às classes.

Desta forma, para se ter BIST em sistemas OO, os seguintes componentes extras são necessários, conforme mostra a Figura 4.1 [Bin94]:

---

[3] In: http://www.codeguru.com/cpp_mfc/index.html

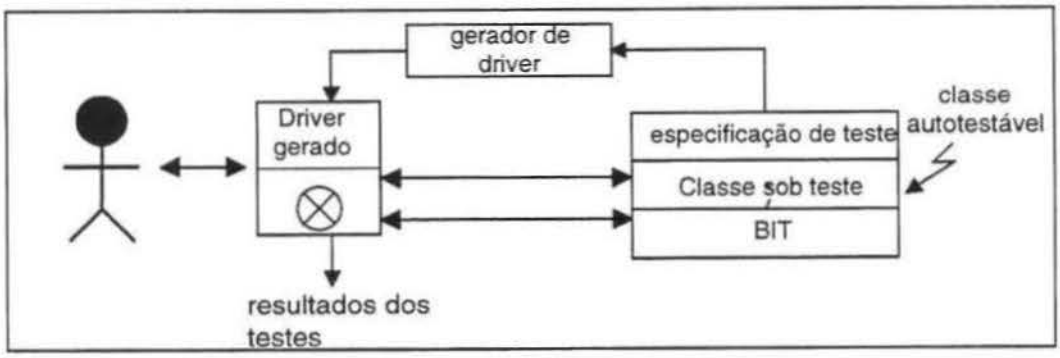

Figura 4.1 Classe Autotestável

- *classe autotestável,* que é composta pela *classe sob teste* (CUT, do inglês "*class under test*") acrescentada de sua *especificação de teste*, que é o modelo de base usado na geração dos testes e pelos *mecanismos embutidos de teste* ou BIT (métodos relatores e assertivas) usados no controle e observação do estado interno, bem como na análise dos resultados;
- *um gerador de driver,* que gera o *driver* que irá ativar e controlar a CUT durante a execução dos testes, a partir da especificação embutida na classe;
- *um driver específico,* que representa a seqüência executável de teste, gerado a partir do critério de teste implementado pelo *driver gerador.* A avaliação é baseada no estado da CUT, dado pelas características BIT, e pela saídas da CUT, de forma a verificar se as saídas da CUT estão corretas ou não.

4.4.2 Abordagem Incremental Hierárquica

Além do conceito de autoteste, a *Técnica Incremental Hierárquica* (HIT) [Har92] foi utilizada, com o objetivo de possibilitar que o reuso se estenda também aos testes. O maior benefício desta técnica é a redução do esforço adicional para testar novamente cada subclasse. Apenas características (atributos ou métodos) novas, substituídas ou alteradas são testadas. A técnica é hierárquica porque se baseia na hierarquia entre classes introduzida pelo mecanismo de herança, e é incremental, pois visa reutilizar os testes de uma classe em um nível hierárquico para testar classes em níveis subsequentes.

O reuso dos testes depende da classificação das características (métodos e atributos) da CUT, ou seja [Sie96]: *característica nova*, definida apenas na subclasse e necessita de testes completos; *característica herdada*, herdada sem modificação e necessita apenas de

testes quando interage com métodos novos ou redefinidos; e *características redefinidas*, definida na subclasse e modificada na subclasse, pode reutilizar os modelos de testes funcionais da superclasse. Entretanto, a técnica impõe algumas restrições de projeto e implementação, tais como: (1) a herança existente deve ser de comportamento; e (2) não é aplicável na existência de herança múltipla; e (3) o programas devem ser implementados em C++. Informações mais detalhadas sobre a abordagem HIT podem ser obtidas em [Har92].

4.4.3 Automatização dos testes

A aplicação efetiva de um critério de teste exige a utilização de uma ferramenta de teste automatizada. Desta forma, um protótipo, chamado *Concat* (Construção de Classes AutoTestáveis), foi desenvolvido para ajudar o testador na construção e uso das classes autotestáveis [Toy98]. O protótipo dá suporte as seguintes tarefas:

1. *geração do driver específico,* a ferramenta possui um driver gerador que gera a seqüência de teste executável (driver específico) para classe, com base na sua especificação construída a partir do modelo de teste usado. Esta seqüência representa o conjunto de requisitos de teste para a classe, cada requisito corresponde a um caminho no modelo, e pode ser composto por um método ou um conjunto de métodos;
2. *auxílio na instrumentação da classe,* a classe deve ser instrumentada para a introdução de mecanismos embutidos de teste ou BIT, usados no controle e observação do estado interno, bem como na análise dos resultados. Na instrumentação, métodos relatores e assertivas são introduzidos em cada classe sob teste. Os métodos relatores permitem verificar o estado interno da classe; e as assertivas são utilizadas para testar se um programa satisfaz certas restrições semânticas, como: uma pré-condição (que testa uma condição de entrada do objeto), uma pós-condição (que testa uma condição de saída do objeto), ou uma invariante de classe (que testa condições invariantes do objeto). Desta maneira, a ferramenta auxilia na instrumentação da classe, fornecendo a interface de métodos relatores que serão usados na análise dos resultados. Porém, a introdução das assertivas deve ser feita manualmente.

4.4.4 Testes Baseados no Modelo de Fluxo de Transação

Para a construção da especificação de teste foi utilizado neste trabalho o modelo de fluxo de transação. Este é um modelo funcional, definido por Beizer [Bei90] para testar sistemas concorrentes e adaptado por Siegel [Sie96] para testar objetos. O modelo ilustra as diferentes maneiras de se criar um objeto, as diferentes tarefas ou processos que um objeto pode realizar e as diferentes maneiras de se destruir um objeto [Sie96]. Uma *transação* representa então, o ciclo de vida de um objeto, desde a criação até sua destruição. Assim, cada transação se inicia com a construção de um objeto; realiza um conjunto de tarefas ou processos; e termina com a destruição do objeto. O modelo é dado por um grafo de fluxo de transação, de modo que cada transação seja um caminho através deste grafo. Os arcos representam chamadas de métodos e os nós representam processos ou tarefas de interesse, por exemplo a execução de um método ou um conjunto de métodos. O testador constrói o modelo de fluxo de transação para a classe sob teste, podendo ser feita a partir de cenários de uso (*use-case)* ou do modelo funcional da classe (DFD).

Nesse texto ilustramos a construção do modelo da classe CObList com base nos cenários de uso (*use-case)* da classe, onde um dos cenários possíveis é mostrado a seguir:

a) Crie uma lista CObList; b) insira um novo objeto na posição inicial da lista; c) recupere a posição inicial da lista; d) insira um novo objeto antes do primeiro objeto da lista; e) encontre um elemento na lista; f) remove um elemento em uma determinada posição g) e destrua a lista CObList;

Construindo-se diversos *use-cases* como o do exemplo, obtém-se o modelo de fluxo de transação, como ilustrado na Figura 4.2. A etapa seguinte consiste na construção do modelo de teste para a classe derivada CSortableObList ilustrado na Figura 4.3, utilizando como base o modelo da classe CObList. Na abordagem HIT, apenas a existência de herança de comportamento é permitida, ou seja, o modelo de teste da classe derivada deve possuir o mesmo comportamento de sua classe base.

Assim, na construção do modelo de teste para a classe derivada não foram retirados arcos ou nós já existentes, pois a subclasse não suprime métodos herdados e nem altera a ordem de execução desses métodos. Métodos novos podem ser introduzidos seja em nós já existentes, seja em novos nós. Novas ligações podem ser criadas, seja para métodos novos, seja para métodos redefinidos, sendo estas realçadas na Figura 4.3.

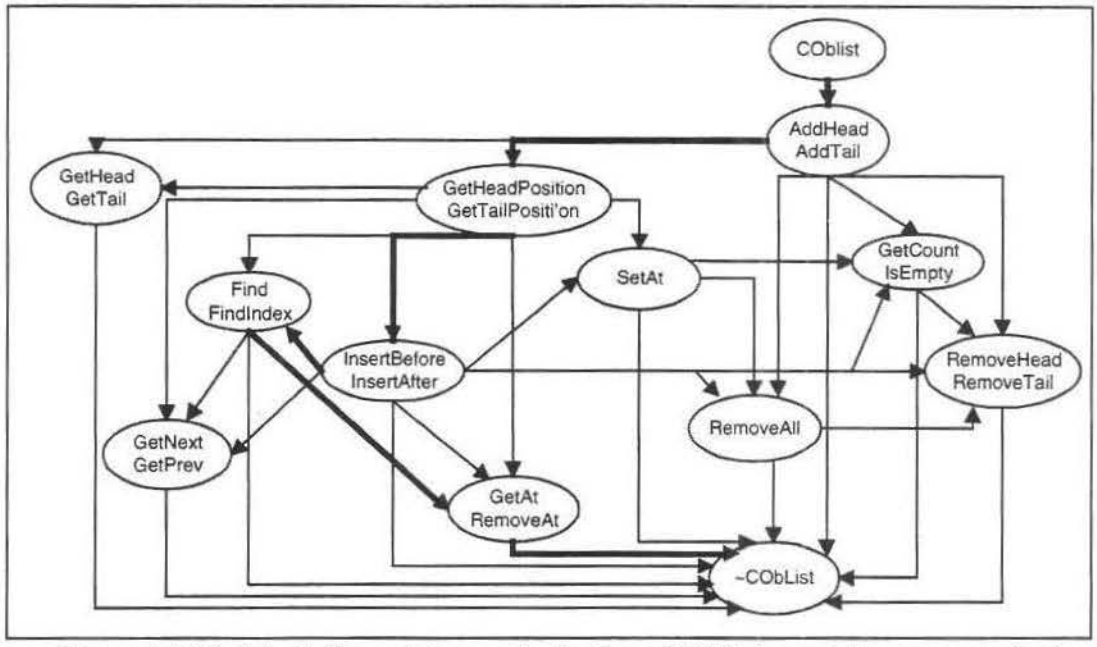

Figura 4.2 Modelo de fluxo de transação da classe CObList; caminho correspondente ao cenário mostrado

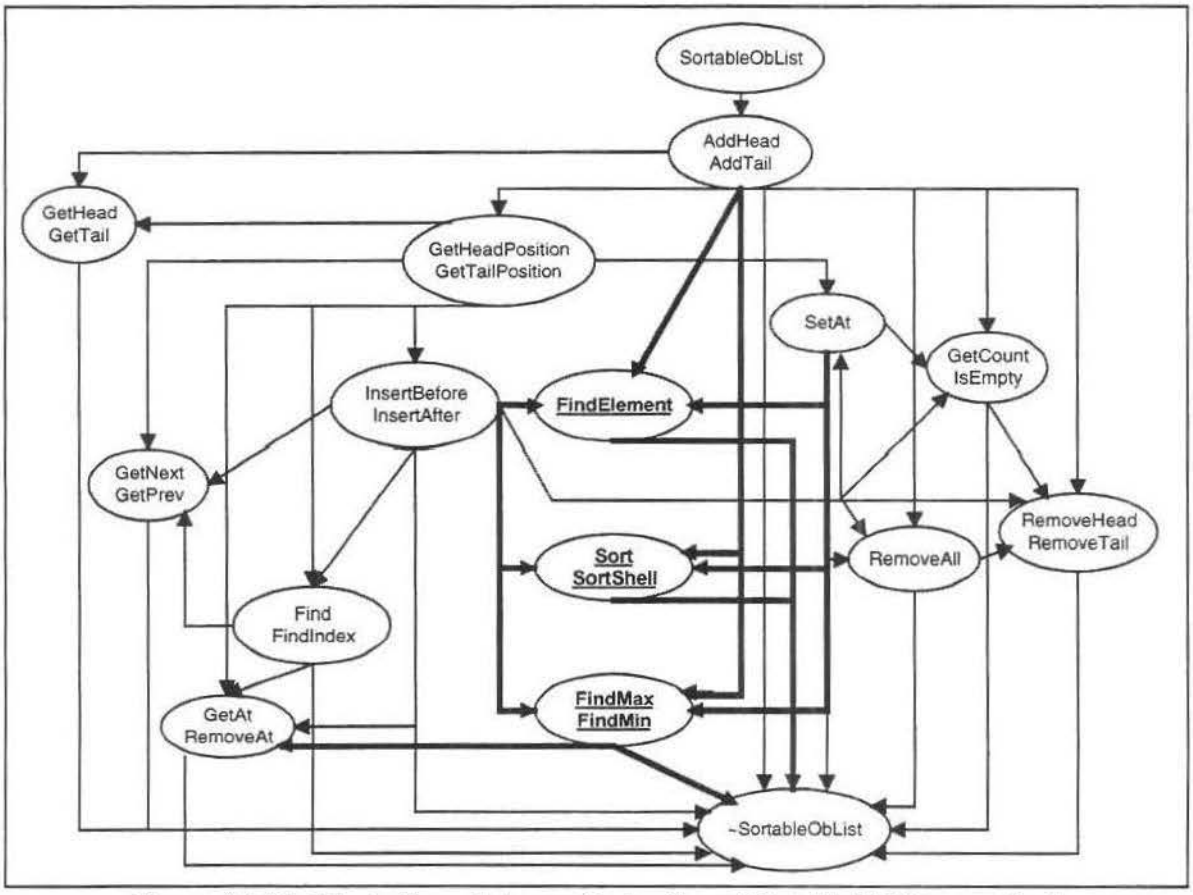

Figura 4.3 Modelo de fluxo de transação da classe CSortableObList, os métodos novos estão sublinhados, e as ligações novas estão realçadas.

4.4.5 Exemplo dos Testes Gerados

Como descrito anteriormente, o conjunto de requisitos de testes é gerado automaticamente pelo protótipo Concat. Cada requisito de teste exercita uma transação do modelo, ou seja, deve-se iniciar com a criação do objeto e finalizar com a sua destruição. O objetivo é cobrir todos os caminhos no modelo (todas as transações). Isto eventualmente é alcançado, pois o modelo não deve possuir ciclos. Um exemplo de um requisito de teste que exercita a transação mostrada na seção 4.4.4 é ilustrado na Figura 4.4.

```
Void Requisito_teste12_1(CObList* cst) {
  Char* met= new char[30];
  POSITION pos;
try {
  cst->Testa_Invariante(); //mecanismo BIT
  met = "AddHead(newelement)";
  cst->AddHead(newelement);
 cst->Testa_Invariante();  //mecanismo BIT
  met = "GetHeadPosition()";
  pos=cst->GetHeadPosition();
  cst->Testa_Invariante(); //mecanismo BIT
  met = "InsertBefore(pos,newelement)";
  cst->InsertBefore(pos,newelement);
  cst->Testa_Invariante(); //mecanismo BIT
  met = "Find(searchValue,pos)";
   cst->Find(searchValue,pos);
   cst->Testa_Invariante();  //mecanismo BIT
   met = "RemoveAt(pos)";
 cst->RemoveAt(pos);
   cst->Testa_Invariante();  //mecanismo BIT
   arq << "Requisito_teste12_1 OK!\n";
   arq.flush();
   objeto=cst;
   cst->Relator("Result.txt",objeto); //mecan. BIT
   arq << "\n";
   arq.close();
   delete cst;
  } .....
```

Figura 4.4 Exemplo de um requisito de teste

4.5 Avaliação Empírica

Esta seção descreve o experimento realizado para avaliar o método de teste baseado no Modelo de Fluxo de Transação, aplicado em conjunto com a técnica HIT.

4.5.1 Mutação de Interface

Para medir o potencial de detecção de falhas do conjunto de testes gerado com base no MFT, foi utilizada a técnica de análise de mutantes.

A função da Análise de Mutantes é avaliar a qualidade de um conjunto de teste baseado na sua habilidade em revelar falhas simples injetadas no programa sendo testado [Del97]. Esta análise segue o seguinte procedimento [Sou97]: (i) pequenas perturbações, obtidas através dos operadores de mutação definidos, são inseridas em um programa **P**, gerando um programa mutante **P'**; (ii) um conjunto de teste **T** é executado com **P'**; (iii) é verificado o comportamento de **P'**, e requisito seja diferente de P, então esse mutante é

dito "morto"; requisito seja equivalente, esse mutante se encontra vivo, ou porque **T** não é adequado para distinguir **P** de **P'**, ou ainda, porque **P'** possui comportamento semelhante ao programa **P**, requisito em que **P** e **P'** são ditos equivalentes. A aplicação de um conjunto de teste que resulta em apenas mutantes mortos e equivalentes, significa que o mesmo possui um bom potencial para a detecção de falhas.

Em particular, a Mutação de Interface [Del97] se preocupa em descobrir falhas de integração entre unidades, introduzindo assim erros diretamente nos pontos relacionados com a interação entre essas unidades. Assim, é definido um conjunto de operadores de mutação de interface, que buscam modelar falhas usualmente cometidas nas interações do programa. Este conjunto é composto de 33 operadores de mutação divididos em dois grupos, de acordo com a forma como as mutações são realizadas. O primeiro grupo realiza mutações dentro do corpo de uma unidade; e o segundo grupo realiza mutações onde uma unidade faz chamadas à outra unidade. Além disso, na definição dos operadores são usados os seguintes conjuntos: **P**– parâmetros formais utilizados na chamada da função; **G**– variáveis globais utilizadas na função chamada; **E**– variáveis globais não utilizadas na função chamada; **L**– variáveis declaradas na função chamada; **C**– constantes utilizadas na função chamada; **R**– constantes requeridas.

Os elementos pertencentes aos conjuntos P e G são chamados de *variáveis de interface*, pois são valores que fazem parte da interface da função. Os elementos dos demais conjuntos são chamados de *variáveis não de interface e constantes*.

Sabe-se que a utilização da análise de mutantes possui um alto custo computacional na execução e na sua análise, devido ao grande número de mutantes gerados. Desta forma, devido ao fator tempo optou-se por utilizar o conjunto de operadores essenciais de interface proposto por [Vin99], sendo este um subconjunto dos operadores de interface descritos acima. A Tabela 4.1 apresenta este conjunto essencial de interface.

O conjunto de operadores de interface foi desenvolvido para a linguagem C e os testes foram aplicados a programas em C++. Mas como o objetivo da estratégia proposta é exercitar as interações entre métodos de uma classe (interações intra-classe), e dado que não se dispõe ainda de operadores de mutação específicos para programas O-O, a mutação de interface foi considerada útil para avaliar o conjunto de testes gerado.

Grupo	num	Operador	Descrição
I	01	IndVarBitNeg	Acrescenta operador de negação de bit em variáveis não de interface
I	02	IndVarRepGlob	Troca variáveis não de interface por elemento de G
I	03	IndVarRepLoc	Troca variáveis não de interface por elemento de L
I	04	IndVarRepExt	Troca variáveis não de interface por elemento de E
I	05	IndVarRepReq	Troca variáveis não de interface por elemento de R
II	06	ArgAriNeg	Acrescenta operador de negação aritmética antes de argumento
I	07	CovAllNode	Cobertura de nós do grafo de fluxo de controle do programa
I	08	DirVarBitNeg	Acrescenta operador de negação de bit em variáveis de interface

Tabela 4.1 Conjunto de operadores essenciais de interface

4.5.2 Descrição do Experimento

Para os experimentos, foram utilizadas as duas classes exemplo, já descritas anteriormente. A Tabela 4.2 mostra informações gerais sobre as classes.

Descrição	CObList	CSortableObList
# métodos	21	6
# arcos	30	43
# nós	13	16
# requisitos de teste gerados	329	233

Tabela 4.2 Descrição dos dados das classes

Para evitar uma possível influência, o conjunto de requisitos de teste foi gerado antes do início da criação dos mutantes. A mesma pessoa que construiu o modelo de teste, criou os mutantes, o que poderia influenciar na escolha dos mutantes a serem criados. Entretanto, concluiu-se que o conhecimento dos requisitos de teste não teria grande impacto na inserção das falhas, já que esta foi baseada em um conjunto claro de regras. Além disso, a geração dos requisitos de testes, como descritos anteriormente, foi feita de maneira automática, utilizando o protótipo *Concat*.

Cada mutante foi criado como um programa separado. Cada programa foi compilado de forma a assegurar que seria compilado corretamente. Então o programa mutante foi executado com o conjunto de testes gerado. Se o programa abortava ou tinha um término anormal, a falha era considerada detectada. As saídas produzidas pelo programa mutante eram comparadas, ao término da execução, com as saídas do programa original,

utilizando-se um programa criado especialmente para esta finalidade. As saídas do programa original foram validados manualmente antes da avaliação empírica começar. A existência de alguma diferença entre as saídas foi considerada uma falha, e portanto o mutante considerado morto. Com o restante dos mutantes considerados vivos, foi feita uma análise para determinar se os programas eram equivalentes. Dessa forma, pode-se determinar a quantidade de mutantes mortos e equivalentes.

A investigação empírica do uso do MFT foi feita através de dois estudos. Nas próximas subseções são descritos cada estudo, e a discussão de seus resultados.

4.5.3 Estudo 1

O objetivo deste primeiro estudo foi avaliar a eficiência do conjunto de testes gerados na detecção de falhas existentes na própria classe. A classe CSortableObList foi utilizada para esse fim.

4.5.3.1 Descrição

Os operadores de mutação foram aplicados aos diversos métodos da classe CSortableObList, totalizando 700 mutantes gerados. Para cada operador essencial de interface foi determinado que seriam gerados no mínimo 25 mutantes por método. Em alguns requisitos não foi possível atingir essa quantidade, seja porque a interface do método tinha poucas variáveis, seja porque alguns operadores não eram aplicáveis. Por exemplo, o operador IndVarRepLoc não é aplicável requisito o método não tenha variáveis locais. Além disso, para três tipos de operadores não foram criados mutantes: CovAllNode , ArgAriNeg, e DirVarBitNeg, isto porque o operador CovAllNode é baseado no modelo de fluxo de controle para testes estruturais, que não foi usado neste trabalho. Já os operadores ArgAriNeg e DirVarBitNeg não foram muito utilizados, pois as variáveis eram na sua maioria de tipo não escalar, não possibilitando assim o uso de variáveis de negação.

4.5.3.2 Resultados e Discussão

A Tabela 4.3 mostra que dos 700 mutantes gerados, 652 foram detectados. Isto significa que o programa teve uma finalização anormal, ou gerou uma saída diferente do programa original, ou ainda foi sinalizada a violação de uma assertiva (pós-condição ou invariante de classe). O total de mutantes mortos representa 93,15% de mutantes gerados, o total de

mutantes equivalentes representa 2,72%, restando apenas 4,15% de mutantes vivos, como é mostrado na Tabela 4.4. Vale ressaltar que 59 dos mutantes foram mortos através da violação das assertivas, representando 8,43% do total de mutantes mortos. Desta forma, o escore de mutação foi de 0,9574 (total de mutantes mortos desconsiderando os mutantes equivalentes), mostrando que o conjunto de testes gerado pelo Modelo de Fluxo de Transação conseguiu detectar a maioria dos mutantes gerados.

Método	**Número do Tipo de Operador de Mutação**				
	01	**02**	**03**	**04**	**05**
Sort1	04	64	99	104	09
Sort2	07	25	25	25	25
ShellSort	24	25	25	25	28
Findmax	07	25	11	25	25
Findmin	07	25	11	25	25
Total	49	164	171	204	112

Tabela 4.3 Número de mutantes inseridos em cada método da classe CSortableObList

	Número de Falhas	**Porcentagem das Falhas**
Mortos	652	93,15%
Vivos	29	4,15%
Equivalentes	19	2,72%

Tabela 4.4 Número e porcentagem de falhas

A Tabela 4.5 apresenta a distribuição dos mutantes gerados por tipo de operadores de mutação.

Operador de Mutação	**Mortos**	**Vivos**	**Equivalentes**
01	42	7	0
02	152	9	3
03	168	3	0
04	197	6	1
05	93	4	15
Total	**652**	**29**	**19**

Tabela 4.5 Resultado por categoria

4.5.4 Estudo 2

O objetivo principal deste estudo foi verificar o potencial de detecção de falhas do conjunto de teste da CSortableObList, na existência de uma falha em sua classe base. O

motivo deste experimento foi analisar se a regra utilizada pela técnica HIT, de não gerar testes para caminhos que contêm apenas métodos herdados (ver seção 4.4.2), é válido em todas as circunstâncias. De acordo com esta técnica, em classes derivadas apenas características (métodos e atributos) novas, substituídas ou alteradas são testadas. Desta forma, operadores de mutação foram aplicados aos métodos da classe base CObList, e os testes realizados na classe derivada.

4.5.4.1 Descrição

Para evitar tendenciosidade, a escolha dos métodos levou em conta o seguinte aspecto: de acordo com a técnica HIT, os testes da classe base não são reaplicados aos métodos herdados sem modificação, ou aos que não interajam com métodos novos ou redefinidos. Portanto, ao se testar a classe derivada, não são gerados novos requisitos de testes para estes tipos de métodos. Desta maneira, três tipos de métodos foram escolhidos para se aplicar mutação: (a) um método herdado sem modificação, mas que interage com métodos novos ou modificados; (b) um método herdado sem modificação, mas que é chamado dentro de um método novo ou modificado; (c) um método herdado sem modificação, sem chamadas e interações com métodos novos ou modificados. O motivo desta escolha foi analisar o potencial de detecção de falhas existentes em cada um destes métodos, sendo estes mostrados na Tabela 4.6. Os operadores de interface foram aplicados aos três métodos; para cada operador foi determinado que seriam gerados uma média de 10 mutantes por método. Porém, em alguns requisitos não foi possível atingir essa quantidade, devido a circunstâncias similares às apresentadas no primeiro estudo.

método	Descrição	utilização
AddHead	Adiciona um elemento na primeira posição da lista	Chamado no requisito de teste da classe derivada
RemoveAt	Remove um elemento da lista dada uma posição	Chamado por algum método da classe derivada
RemoveHead	Remove o primeiro elemento da lista	não é chamado pela classe derivada ou usado pelo requisito de teste

Tabela 4.6 Descrição dos métodos da classe CObList escolhidos

4.5.4.2 Resultados e Discussão

A Tabela 4.7 descreve o total de mutantes gerados para os métodos escolhidos da classe CObList. 159 mutantes foram gerados, destes apenas os mutantes gerados para os dois primeiros métodos foram encontrados, como ilustrado na Tabela 4.8. Isto porque, o método RemoveHead não é testado no conjunto de teste da classe derivada. Uma vez que se trata de um método sem alterações e não se encontra em caminhos no MFT que foram criados ou alterados, portanto não são gerados requisitos de testes na classe derivada para testá-lo.

Método	**Número do Tipo de Operador de Mutação**				
	01	**02**	**03**	**04**	**05**
AddHead	2	21	3	13	3
RemoveAt	0	30	25	25	2
RemoveHead	3	15	5	8	4
Total	**5**	**66**	**33**	**46**	**9**

Tabela 4.7 Número de mutantes inseridos em cada método da classe CObList

Método	**Mutantes**	**Tipo de Operador de Mutação**				
		01	**02**	**03**	**04**	**05**
AddHead	**Mortos**	2	21	2	10	3
	Vivos	0	0	1	3	0
	Equivalentes	0	0	0	0	0
RemoveAt	**Mortos**	0	22	21	19	1
	Vivos	0	8	4	6	1
	Equivalentes	0	0	0	0	0
RemoveHead	**Mortos**	0	0	0	0	0
	Vivos	3	15	5	8	4
	Equivalentes	0	0	0	0	0

Tabela 4.8 Resultado por operador de mutação

Apesar do pequeno número de experimentos, os resultados desse estudo permitem mostrar a necessidade de se retestar os métodos da classe derivada reutilizando os testes gerados para a classe base. Com o uso da técnica HIT, os testes da classe base não são reaplicados aos métodos herdados sem modificação, ou que não interajam com métodos novos ou redefinidos. Desta forma, falhas podem ficar escondidas por exemplo, no requisito da classe base pertencer a uma biblioteca, esta ser substituída por uma nova

versão, e a nova versão ser utilizada sem reteste pela classe derivada. Por isso, é importante que os métodos herdados sejam retestados na classe derivada para evitar que modificações efetuadas nesses métodos afetem o comportamento da classe derivada.

4.6 Conclusão e Trabalhos Futuros

Este artigo apresenta um estudo empírico que investiga a eficácia na detecção de falhas de requisitos de testes construídos com base no Modelo de Fluxo de Transação. Este modelo foi utilizado no teste de classes, juntamente com a técnica Incremental Hierárquica. O objetivo principal foi verificar o potencial de detecção de falhas do conjunto de teste gerado.

Para isto, uma análise de mutação foi feita, utilizando operadores de Mutação de Interface. Dois estudos foram realizados; no primeiro estudo, foram criados 700 mutantes distribuídos entre os cinco métodos da classe escolhida, com o objetivo de verificar o potencial de detecção de falhas na classe derivada. Destes 700 mutantes, 652 foram mortos, 19 foram determinados equivalentes, restando somente 29 mutantes vivos, representando 4,15% dos mutantes. Estes resultados mostraram que 93,15% dos mutantes foram mortos, e que o modelo de fluxo de transação pode ser útil na detecção de falhas em sistemas O-O. Este estudo mostrou que a combinação do MFT com uma ferramenta automatizada para a geração dos requisitos de teste, possui um bom potencial na determinação de falhas.

No segundo estudo, tentou-se analisar o comportamento do conjunto de teste baseado na técnica HIT, com a ocorrência de falhas na classe base. O objetivo foi verificar a necessidade de se retestar métodos herdados sem modificações. Os resultados mostraram que o conjunto de teste criado para a classe derivada CSortableObList não é eficiente na detecção de falhas existentes em métodos herdados sem modificação, uma vez que, nenhum dos 35 mutantes gerados para o método RemoveHead foram encontrados. Note que os dois primeiros métodos escolhidos também são métodos herdados sem modificação, porém ou se encontram em caminhos do modelo que possuem alguma alteração, ou são chamados por métodos novos ou alterados. Neste requisito, dos 124 mutantes gerados 101 foram mortos, representando 81,45% dos mutantes gerados para estes dois métodos. Estes resultados mostraram a necessidade de se retestar os métodos

herdados da classe derivada, reutilizando os testes gerados para a classe base, para evitar que possíveis falhas se propaguem.

Esta avaliação examinou um conjunto restrito de classes, gerando manualmente um conjunto de mutantes. Mais avaliações devem ser realizadas para ratificar os resultados obtidos, utilizando um número maior de classes. Além disso, a utilização de uma ferramenta automatizada para a geração dos mutantes para programas em C++ mostra ser importante para a otimização do processo de geração de mutantes. Porém, os resultados se mostraram importantes para mostrar que o Modelo de Fluxo de Transação pode ser uma boa alternativa nos testes de classes de sistemas O-O.

Capítulo 5

O Uso de Métricas de Software para Analisar a Testabilidade e a Reusabilidade de um Sistema Orientado a Objetos

Neste artigo, é apresentado uma análise feita para a escolha do subconjunto de classes a se tornarem autotestáveis do Sistema de Telemetria e Telecomando (TMTC). Esta análise foi baseada nos seguintes critérios: criticalidade, reusabilidade e testabilidade. Com exceção do primeiro critério, que foi avaliado empiricamente através de consultas aos desenvolvedores, os outros foram obtidos através de métricas de software.

Este artigo foi submetido ao XIII SBES 99 - Simpósio Brasileiro de Engenharia de Software, evento realizado de 13 – 15 de outubro de 1999, em Florianópolis SC.

O Uso de Métricas de Software para Analisar a Testabilidade e a Reusabilidade de um Sistema Orientado a Objetos

Rosileny Lie Yanagawa
lie@dcc.unicamp.br

Eliane Martins
Eliane@dcc.unicamp.br

Instituto de Computação – IC
Universidade Estadual de Campinas – UNICAMP
UNICAMP CP 6176 Cep 13081-970
Campinas-SP, Brazil
Fax (019) 788-5847

Abstract

Object-oriented technology is being increasingly used for real world applications in recent years, mainly because it eases the production of reusable software components. Reuse allows costs and efforts of systems development to be reduced. However, more tests are needed to make sure the reusable component is reliable.

Testing object-oriented software has many differences if compared to testing in traditional software. Difficulties are increased because of the new characteristics introduced by this approach: encapsulation, inheritance, and polymorphism. The consequences could be systems with low testability, resulting in more difficulties to find faults and more efforts to produce tests that can meet reliability goals.

In short, we have studied the use of built-in-self testing concepts and built-in functions, mainly used in VLSI circuits, in an object-oriented system. Our objective is to examine how self-testing concepts contribute for testing and testability of object-oriented systems. We used software metrics to choose where these concepts would be inserted into the application. These metrics are applied to a Telemetry[4] and Telecomand [5]System developed by the National Space Research Institute (INPE). This paper summarizes the results obtained by performing testability and reusability metrics analysis.

Sumário

A reutilização de componentes é apontada como umas das principais justificativas para a utilização da orientação a objetos no desenvolvimento de sistemas de software. Entretanto, essa reutilização requer uma alta confiabilidade dos componentes, exigindo a realização de um grande número de testes que possam garantir a qualidade do produto final. Desta forma, testes devem ser aplicados a cada vez que um componente é reusado em um novo contexto.

Testes em sistemas orientados a objetos são dificultados devido ao conjunto de características introduzido pela abordagem, tais como: o encapsulamento, a herança e o polimorfismo. A consequência disto é a redução da testabilidade desses sistemas, o que significa um aumento na dificuldade de encontrar falhas e um esforço maior para produzir testes que garantam a produção de classes confiáveis. Com isso, a realização dos testes resulta no aumento dos custos, que são na maioria das vezes limitados.

Em resumo, este trabalho tem por objetivo a melhoria da testabilidade de uma aplicação OO, utilizando os conceitos de autoteste e de mecanismos embutidos de testes usados em circuitos VLSI. Para determinar onde inserir essas característica foi feita uma análise da aplicação com base nos seguintes critérios: criticalidade, reusabilidade e testabilidade. Tanto a reusabilidade quanto a testabilidade foram avaliadas através de métricas; os outros foram determinados empiricamente, através de consultas aos desenvolvedores. A aplicação alvo é o sistema de Telemetria e Telecomando do Instituto de Pesquisas Espaciais (INPE). No texto são mostrados os resultados obtidos na análise dessa aplicação.

---

[4] telemetria são informações da espaçonave codificada e transmitida à Terra
[5]telecomando são ordens ou comandos codificados e transmitidos da Terra para a espaconave.

5.1 Introdução

A construção de componentes reutilizáveis tem sido indicada como uma das principais razões para a utilização, cada vez maior, da programação orientada a objetos. Entretanto, esta reutilização exige dos componentes um alto índice de confiabilidade, resultando na necessidade de um grande número de testes. Por outro lado, testes são sempre apontados como consumidores de tempo e de custo no desenvolvimento de software. Segundo Voas [Voa95a], testes de software consumem no mínimo 50% dos esforços de desenvolvimento, extrapolando às vezes os índices de 90% no requisito de sistemas críticos.

As dificuldades de teste são ainda maiores quando se trata de sistemas orientados a objetos (OO). Isto se deve, principalmente, ao conjunto de características introduzido pela abordagem, tais como: o encapsulamento, a herança e o polimorfismo. A consequência disto pode ser a redução da testabilidade desses sistemas, o que significa um aumento na dificuldade de encontrar falhas e um esforço maior para produzir testes que garantam a produção de classes confiáveis.

O uso de soluções que permitam melhorar a testabilidade de sistemas OO se faz portanto necessário. Em [Bin94] é proposto o uso do projeto visando testabilidade (DFT, do inglês "*Design for Testability*") e do conceito de autoteste, usados em hardware, para a melhoria da testabilidade de sistemas orientados a objetos. Além da capacidade de teste embutido ou BIT, do inglês *"Built-in Test"* para facilitar o acesso a componentes internos de um objeto, Binder também propõe a introdução da capacidade de geração, execução e análise automática dos testes, tendo-se assim a aplicação do conceito de autoteste às classes, denominado em inglês *"Built-in-Self Test"* ou BIST.

Com o objetivo de avaliar a aplicabilidade do conceito de autoteste no desenvolvimento e nos testes de sistemas orientados a objetos, o mesmo será utilizado nos testes de classes de um sistema de Telemetria e Telecomando do Instituto Nacional de Pesquisas Espaciais (INPE). Entretanto, devido ao fator tempo que é normalmente restrito na realização de testes, não é possível a realização dos testes de todas as classes da aplicação. Assim, fez-se necessário a utilização de critérios que possibilitassem a determinação das classes a se tornarem autotestáveis.

Neste trabalho foi realizada uma análise da aplicação baseada nos seguintes critérios: criticalidade, reusabilidade e testabilidade. Com exceção do primeiro critério, que foi avaliado empiricamente através de consultas aos desenvolvedores, os outros foram obtidos através de métricas de software. Assim, neste texto são apresentados as análises e resultados obtidos sobre a aplicação, com base nos critérios citados.

As seções seguintes estão organizadas da seguinte forma; a Seção 5.2 apresenta fundamentos importantes relacionados a testabilidade e a reusabilidade de sistemas orientados a objetos; a Seção 5.3 descreve as métricas e os critérios utilizados; a Seção 5.4 apresenta a descrição da aplicação alvo; a Seção 5.5 mostra os resultados obtidos através das métricas; a Seção 5.6 apresenta uma avaliação dos resultados obtidos; a Seção 5.7 descreve algumas recomendações aos desenvolvedores; por fim, a Seção 5.8 apresenta as conclusões e as extensões do trabalho.

5.2 Visão Geral

As seções seguintes apresentam uma visão geral dos conceitos de testabilidade e reusabilidade em sistemas orientados a objetos. O objetivo é simplesmente fornecer uma base para o entendimento do que será discutido a seguir. As referências [Bin94,Voa95b, Bie96] podem ser consultadas para uma apresentação mais detalhada destes conceitos.

5.2.1 Conceitos de Testabilidade

Segundo o padrão IEEE 610.12, a testabilidade é definida como:

3. em que medida um sistema (ou componente) facilita o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos;
4. em que medida um requisito é representado de forma a permitir o estabelecimento de critérios de testes e a realização dos testes que vão determinar se esses critérios foram atendidos.

Em outras palavras, a testabilidade de um software pode ser definida como a medida de quão fácil é satisfazer a critérios de testes pré-estabelecidos. O grau de testabilidade de um componente ou sistema tem uma influência direta nos esforços gastos na fase de testes.

Assim, quanto maior a testabilidade de um componente ou sistema mais fácil será alcançar os objetivos estabelecidos.

Outra definição para testabilidade vem sendo proposta por Voas et al. [Voa92, Voa95b, Voa98], como sendo a possibilidade de ocorrer um defeito[6] requisito existam falhas, ou seja, um componente com baixa testabilidade tende a produzir saídas corretas para a maioria das entradas que executam uma falha.

Pode-se citar dois aspectos importantes a fim de se obter um software com boa testabilidade: controlabilidade e observabilidade. A observabilidade refere-se à facilidade em se determinar o quanto as entradas fornecidas afetam as saídas obtidas. A controlabilidade refere-se à facilidade de se produzir a saída desejada a partir da entrada fornecida. Assim, para se testar um componente deve ser possível controlar (controlabilidade) suas entradas e estado interno, bem como observar (observabilidade) a sua saída. Em geral é difícil conseguir uma boa controlabilidade e observabilidade, em especial se o componente está embutido em outros componentes. Nesse aspecto os sistemas orientados a objetos, devido a suas características intrínsecas, apresentam alguns obstáculos à testabilidade, conforme será visto a seguir.

5.2.2 Impacto da Orientação a Objetos sobre a Testabilidade

Os conceitos de OO dados nesta Seção são baseados em [Buz98]. Os termos classe, objeto e atributos, sendo amplamente conhecidos, não serão definidos aqui. Um objeto encapsula dados (i.e, uma coleção de atributos) e os algoritmos que manipulam esses dados; esses algoritmos são chamados de *operações*. As operações são implementadas como *métodos*[7]. Objetos interagem através do envio de *mensagens*; a recepção de mensagem faz com que seja executado um método correspondente da interface pública do objeto recipiente. O envio de mensagem corresponde portanto à invocação de uma operação declarada na interface pública de um objeto. A interface pública de um objeto compreende o conjunto de operações que podem ser chamadas por outros objetos.

---

[6] Considera-se que ocorra um defeito quando o sistema produz uma saída diferente da saída esperada

[7] Apesar de ter sido utilizado neste trabalho a linguagem C++, preferiu-se utilizar nesse texto o termo método em vez de função-membro, como eles são chamados nessa linguagem. As funções usadas fora do escopo de uma classe são chamadas de funções globais.

A seguir será visto o impacto sobre a testabilidade de mecanismos tais como: encapsulamento, herança, polimorfismo e ligação dinâmica, e relacionamentos. As referências [Smi90, Per90, RSTC95, Kun95, Bin94, Bar98] foram usadas como base.

i. Encapsulamento

O encapsulamento descreve o empacotamento de uma coleção de itens. A classe é, usualmente, o nível de encapsulamento em abordagens OO. O encapsulamento permite que detalhes internos da implementação sejam escondidos do mundo externo (ocultação da informação). As diferentes metodologias e linguagens oferecem diferentes opções de visibilidade das características de uma classe (atributos e operações). Em C++, por exemplo, tem-se características *públicas* (visíveis por todos os objetos), *privadas* (visíveis internamente ao objeto, somente) e *protegidas* (visíveis internamente ao objeto ou aos objetos de classes derivadas).

Em relação aos testes, algumas questões são introduzidas pelo encapsulamento, tais como:

- dificuldade em se testar os atributos e métodos que estão escondidos sem a quebra do encapsulemento, dificultando o controle e observação do estado interno do objeto;
- aumento nas chances de ocorrerem falhas de interface, devido aos sistemas OO serem, em geral, compostos por muitos componentes pequenos, altamente coesos, e que trocam mensagens entre si.

ii. Herança

Herança é um mecanismo que permite a uma classe (classe *derivada* ou *subclasse*) herdar características (métodos e atributos) de outra classe (classe *base* ou *superclasse*). Diz-se que há *herança múltipla* quando uma classe derivada herda característica de mais de uma classe base (estas constituem então suas classes *ancestrais*).

A herança é um mecanismo muito importante para a reusabilidade, mas apresenta uma série de dificuldades para os testes:

- a implementação do mecanismo de herança varia de uma linguagem para outra, portanto a estratégia a ser utilizada para os testes é altamente influenciada pela linguagem;

- reuso implica em re-teste; mesmo que as características herdadas permaneçam inalteradas na subclasse, elas devem ser testadas no novo contexto. A herança de comportamento[8] é melhor nesse requisito, pois permite que as entradas de testes das partes que não foram modificadas possam ser reutilizadas;
- a complexidade de uma classe vai aumentando quanto mais baixo ela estiver na hierarquia de herança e o uso de herança múltipla aumenta ainda mais essa complexidade. O aumento de complexidade implica em que o esforço necessário para a preparação dos testes seja maior. Em especial para determinar que entradas de testes podem ser reaplicadas, quais devem ser modificadas. Obviamente que o uso de herança de implementação é um fator de complexidade a mais, podendo inclusive impedir o reuso de testes;
- as modificações em uma ou mais ancestrais podem causar efeitos indesejáveis nas classes derivadas, tendo como implicação a necessidade de retestá-las.

iii. Polimorfismo e ligação dinâmica

Polimorfismo significa "várias formas para o mesmo nome", e quando usado no contexto de orientação a objetos significa que diferentes tipos de objetos podem responder à mesma mensagem de maneiras diferentes. Alguns conceitos relacionados ao polimorfismo são relevantes para as atividades de testes, tais como: redefinição, sobrecarga e parametrização. A redefinição permite que um método ou atributo da superclasse seja redefinido na subclasse. A sobrecarga permite associar um método a um objeto em tempo de execução. Por fim, a parametrização permite o uso de classes genéricas ou classes parametrizadas, as quais servem de moldes para outras classes.

A ligação dinâmica (do inglês,"*dynamic*" ou "*late binding*") pode ser definida como sendo a associação, em tempo de execução, entre uma operação e a sua implementação.

O uso de polimorfismo e ligação dinâmica acarreta nas seguintes dificuldades para os testes:

- a geração de testes baseados na implementação se torna mais complexa, pois o código a ser executado é mais dependente das condições de execução do que nas linguagens

[8] Herança de comportamento garante que a subclasse tenha o mesmo comportamento da subclasse

imperativas. Portanto, uma análise estática do código para determinar as entradas de teste que satisfaçam a um determinado critério não é suficiente;

- uso de herança de implementação[9] pode causar a ocorrência de muitos erros de execução, pois a classe derivada não tem o mesmo comportamento de suas ancestrais. A dificuldade está em selecionar entradas de teste que permitam que esses erros sejam detectados;
- a determinação de todos os usos polimórficos de um objeto ou característica é impossível.

iv. Relacionamentos

Objetos podem se relacionar uns com os outros de diferentes formas [Sie96, cap.2]: um objeto pode conter outro objeto; objetos podem enviar mensagens para outros objetos; objetos podem ser executados concorrentemente. A variedade e complexidade dos relacionamentos entre objetos causam o chamado problema da interdependência complexa [Kun95], que acarreta as seguintes dificuldades para os testes:

- é difícil entender uma classe, pois ela depende de muitas outras;
- é difícil saber por onde começar os testes, pois não há uma hierarquia do tipo coordenador-subordinado, como nas linguagens imperativas;
- torna-se muito complicado construir "stubs";
- é difícil avaliar o impacto de modificações sobre o sistema.

Dadas as vantagens oferecidas pela orientação a objetos para a melhoria da qualidade de software, a melhoria da testabilidade é extremamente importante para garantia da confiabilidade e manutenibilidade de sistemas desenvolvidos segundo esse paradigma. Na Seção a seguir será visto como técnicas usadas no hardware podem ser úteis nesse requisito.

5.2.3 Autoteste em Sistemas OO

Binder [Bin94] propõe a aplicação da noção de projeto visando a testabilidade (DFT) no desenvolvimento de sistemas orientados a objetos. A testabilidade deve ser levada em conta

---

[9] herança é usada como uma técnica de implementação, não garantindo que a subclasse tenha o mesmo comportamento da superclasse [Buz98]

o longo de todo o processo de desenvolvimento [Bin94]. Nesse estudo o enfoque está nas modificações a serem introduzidas no código para se conseguir alta testabilidade. O DFT é normalmente usado em hardware para melhorar a testabilidade de circuitos integrados (CI). A dificuldade do teste de CIs está, por um lado, na aplicação de padrões de teste a pontos internos do circuito e na observabilidade dos valores lógicos resultantes em pontos do circuito aos quais se tenha acesso. Por outro lado, tem-se a dificuldade também na geração de padrões de teste. Com o uso de DFT, circuitos adicionais são acrescentados, ainda em fase de projeto, com o intuito de facilitar o acesso aos componentes internos de um circuito. Para simplificar a geração de padrões de teste são também incorporadas estruturas especiais para permitir que os padrões de teste sejam gerados e avaliados internamente. Dessa forma, elimina-se a etapa de geração externa de padrões de teste, e mais ainda, elimina-se a necessidade de um testador externo, o que representa também um fator importante na redução do custo final do produto [Cor91]. Este é o conceito de **autoteste,** denominado em inglês *"Built-in-Self Test"* ou BIST.

Com base nos conceitos de DFT usados em hardware, Binder propôs a construção de classes autotestáveis. Para se ter BIST em sistemas OO, os seguintes componentes extra são necessários, mostrados na Figura 5.1 [Toy98]:

- a *especificação de testes*, que é o modelo de base usado na geração dos testes;
- *os mecanismos embutidos de teste* ou BIT (em inglês, Built-In Tests), usados no controle e observação do estado interno, bem como na análise dos resultados;
- *um driver,* que gera os testes a partir da especificação embutida na classe, executa e avalia os testes, utilizando os procedimentos de testes embutidos.

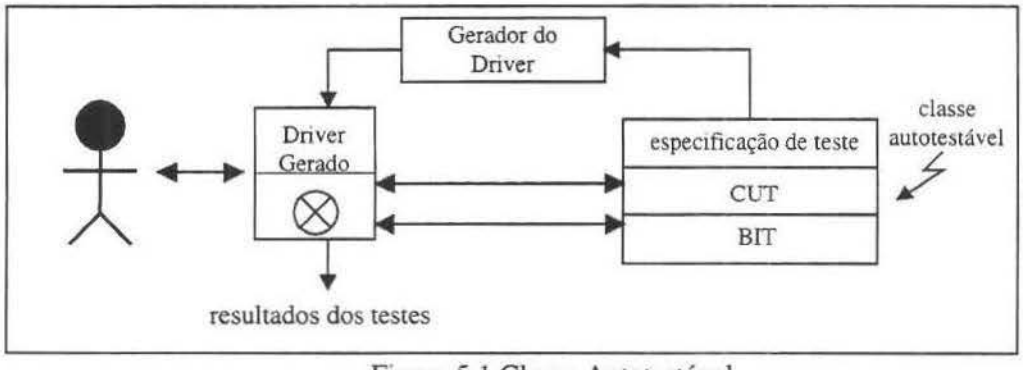

Figura 5.1 Classe Autotestável

O uso de autoteste é muito útil tanto para os testes aplicados durante o desenvolvimento quanto na manutenção. No entanto, o seu uso requer um certo esforço: para produzir a especificação de teste, instrumentar a classe, e gerar o *driver*. Por essa razão nem sempre é viável que todas as classes do sistema sejam autotestáveis. Existe então a necessidade de selecionar as classes para as quais a introdução do conceito de autoteste seja realmente útil, ou seja, os benefícios obtidos na realização de testes subsequentes compensem os custos para a obtenção da classe autotestável. Em outros termos, o autoteste pode ser interessante para classes reusáveis, uma vez que é necessário testá-las a cada reuso em um novo contexto.

No item a seguir são apresentados alguns aspectos sobre a reusabilidade. Em seguida serão vistos os critérios usados nesse trabalho para a escolha das classes nas quais serão utilizadas mecanismos de DFT (BIT ou BIST).

5.2.4 Reusabilidade em Sistemas OO

Pesquisadores têm apontado a geração de componentes de software reutilizáveis como o maior benefício do paradigma orientado a objetos [Bie96, Smi92]. A maior vantagem desta reutilização é dita como sendo a redução de custos e esforços na produção de sistemas.

Segundo Bieman [Bie96] o reuso em sistemas orientados a objetos pode ser visto segundo várias classificações: *reuso público/privado; reuso por instanciação/por parametrização/por herança; e reuso direto/indireto.* O *reuso público* é definido como sendo o reuso de software construído externamente (de outra aplicação); já o *reuso privado* é definido como sendo o reuso de software dentro da própria aplicação. Além disso, existe o *reuso por instanciação* (em inglês, *"verbattim"*), que é feito através de uma instanciação e uso de classes definidas previamente, onde não há modificações na classe reusada; o *reuso por parametrização* (em inglês, *"generic"*), que significa o reuso de classes genéricas; o *reuso por herança* (em inglês, *"leveraged"*), que é feito através do mecanismo de herança, de forma que a subclasse pode modificar características, além de acrescentar as suas próprias. Por fim, tem-se o *reuso direto* que é o reuso sem entidades intermediárias; e o *reuso indireto* que é o reuso através de entidades intermediárias, entre o *cliente* e o *fornecedor*. O *fornecedor* é definido como sendo a classe que realiza algum

serviço para uma outra classe; O *cliente* é definido como sendo a classe que utiliza este serviço.

Dadas estas definições é possível definir a reusabilidade existente na aplicação a partir das métricas estudadas, sendo descritas a seguir.

5.3 Medindo Testabilidade e Reusabilidade em Sistemas OO

Nesta Seção são descritos alguns estudos existente que utilizam métricas de software OO Além disso, as métricas de software disponíveis são apresentadas, e como estas foram utilizadas neste trabalho.

5.3.1 Estudos Existentes

Segundo [Bas96], teste em grandes sistemas é um exemplo de uma atividade consumidora de tempo e de recursos. A aplicação dos mesmos esforços de testes em todas as partes do sistema tem-se tornado inviável. Portanto, existe a necessidade de identificar os componentes mais propensos a falhas, onde os testes devem estar concentrados. A utilização de métricas de software tem sido apontada como boa indicadora destes componentes. Entre as propostas encontradas na área de testes OO, que utilizam métricas como indicadores de qualidade, pode-se citar as propostas apresentadas por [Voa91, Voa97, Voa95a, Bas96, Bie95a, Bie95b].

Os trabalhos apresentados por Voas [Voa91, Voa97, Voa95a] apresentam formas de medir a testabilidade e a reusabilidade de sistemas, através da *análise de sensibilidade*. No primeiro artigo [Voa91] define como sendo o nível de sensibilidade a falhas, ou seja, a probabilidade de que ocorra um defeito requisito haja falhas no sistema. Para determinar o grau de sensibilidade de um programa são realizadas mutações do programa e de seus dados. Quando o programa mutante é executado, comparam-se os seus dados de saída com os dados de saída do programa original. Quanto maior o número de mutantes com saídas diferentes, maior o grau de sensibilidade do programa. Desta forma, a baixa sensibilidade tende a mostrar maior potencial para esconder falhas e, portanto, indicar que mais testes deverão ser realizados.

O trabalho [Voa95a] realiza medições para determinar o grau de confiabilidade de componentes reutilizáveis. Esta análise é aplicada nas classes de um sistema de simulação da NASA, com o objetivo de indicar se os testes originais foram suficientes para garantir a aplicabilidade das classes, de forma confiável, em outros contextos. Neste estudo parte-se do pressuposto de que a necessidade de testes subsequentes em componentes reusados ocasiona a perda parcial dos benefícios alcançados pelo reuso.

Já em [Voa97] é descrito um estudo com o objetivo de inserir assertivas em partes do sistema mais propensas a esconder falhas. Assertivas são definidas como sendo mecanismos que auxiliam a assegurar que um programa satisfaz certas restrições semânticas, aumentando a observabilidade do sistema. Neste trabalho é descrita a utilização da técnica em uma simulação de um sistema ATM (do inglês, "*Automated Teller Machine*"). A *análise de sensibilidade* indicou as regiões do código com baixa testabilidade. Foram introduzidas assertivas nestas partes e realizada novamente a *análise de sensibilidade*. Os resultados indicaram que as assertivas causaram um aumento da testabilidade de cada uma destas partes.

Em [Bas96] é realizada uma investigação empírica do conjunto de métricas de projeto OO introduzidas em [Chi94], com o objetivo de observá-las como indicadoras de classes propensas a falhas. Para isto, foram coletadas métricas de 180 classes de sistemas em C++. Com o uso de análises estatísticas procurou-se avaliar a relação entre as métricas e as classes mais propensas a falhas.

Já o trabalho proposto por [Bie95a] realiza um estudo quantitativo para avaliar o uso da herança em sistemas OO. Neste estudo foram utilizadas métricas de reusabilidade, tais como: o número de classes em cada nível na hierarquia de herança, número (exato, máximo e médio) de filhos, ancestrais, e profundidade de herança. Para a análise foi estudada a utilização da herança em 19 sistemas de software em C++, contendo um total de 2.744 classes. Os resultados mostraram que apesar da herança ser bastante utilizada em alguns tipos de aplicações (como por exemplo, interfaces gráficas), ela foi pouco encontrada na maioria das aplicações estudadas, mostrando ser menos utilizada do que o esperado.

Por fim, em [Bie95b], são definidas e aplicadas métricas de coesão e reuso em um sistema orientado a objetos desenvolvido na Universidade de *Stanford* em C++. Neste

estudo foi realizada uma avaliação do relacionamento entre a coesão e o reuso das classes do sistema. Para isto, duas novas métricas sensíveis a pequenas modificações em relação a coesão da classe foram definidas, são elas: TCC (do inglês, *"Tight class cohesion")* que indica o número de métodos diretamente interligados; e LCC (do inglês, *"Loose class cohesion")* que indica o número relativo de métodos direta e indiretamente conectados. Ambas as métricas indicam o grau de conectividade entre os métodos visíveis em uma classe. Os resultados revelam classes com alto grau de coesão, mas constata que as classes mais reutilizadas via herança possuíam baixa coesão. A Tabela 5.1 apresenta as diferenças existentes em cada estudo descrito acima.

Trabalho	Objetivo da medição	Métrica utilizada	Como foi feita a medição
[Voa91]	Determinar o grau de testabilidade de um sistema	Grau de sensibilidade a falhas	Análise de sensibilidade (teste de mutação)
[Voa95a]	Determinar o grau de confiabilidade de um componente reutilizável	Grau de sensibilidade a falhas	Análise de sensibilidade (teste de mutação)
[Voa97]	Inserir assertivas em partes do sistema mais propensos a falhas	Grau de sensibilidade a falhas	Análise de sensibilidade (teste de mutação)
[Bas96]	Validar empiricamente métricas como indicadores de qualidade	Métricas CK (WMC, NOC, DIT, CBO, RFC, LCOM)	Análise estatística (regressão logística)
[Bie95a]	Avaliar o uso de herança em um conjunto de sistemas desenvolvidos em C++	Número (exato, médio e máximo) de profundidade da herança, filhos, e ancestrais; número de classes em cada nível de herança;	Análise estática (utilização da ferramenta Jasmin)
[Bie95b]	Analisar a relação da coesão e herança existentes nas classes medidas.	Métricas Bieman para a coesão (TCC,LCC) Métricas CK para o reuso (NOC)	Análise estática (utilização da ferramenta GEN++)

Tabela 5.1 Comparação dos estudos existentes

5.3.2 Medidas Escolhidas

O objetivo deste trabalho foi o de levantar as possíveis classes a se tornarem autotestáveis, utilizando para isto métricas de software. Uma análise estática do código foi realizada usando-se a CCCC, desenvolvida como tese de mestrado e que se encontra disponível na Internet [CCC98]. Dentre as métricas de software OO estão disponíveis na ferramenta

CCCC quatro das seis métricas propostas por *Chidamber* e *Kemerer* [Chi94], sendo usadas nos estudos:

- *Weighted Methods per Class* (WMC): a WMC indica a complexidade individual de uma classe. De acordo com [Chi94], todos os métodos igualmente complexos, portanto WMC indica o número total de métodos definidos em uma classe.
- *Depth of Inheritance Tree of a Class* (DIT): DIT é definida como sendo a profundidade máxima de cada classe no grafo de herança.
- *Number of Children of a Class* (NOC): NOC indica o número de filhos de uma classe.
- *Coupling Between Object classes* (CBO): CBO indica o número de classes a que uma dada classe esteja acoplada. Uma classe está acoplada a outras quando utiliza funções ou dados membros (atributos) de outra classe.

Além destas métricas OO, são também disponíveis as seguintes métricas de procedimentos:

- *Lines of Code* (LOC): LOC indica o número de linhas de código, com exceção das linhas brancas e de comentários;
- McCabe's Cyclomatic Complexity (MVG): MVG indica o número de decisões contidas em cada função/método do programa.

5.3.3 Utilização das Métricas

Com o objetivo de determinar as classes da aplicação às quais devem ser acrescentados os mecanismos de DFT, foram usados os seguintes critérios:

a) criticalidade: a classe é responsável por uma parte essencial para o funcionamento correto do sistema
b) reusabilidade: a classe é reusada por um ou mais componentes do sistema (reuso privado, ver 5.2.4).
c) testabilidade: a classe é difícil de ser testada, ou seja, não é fácil satisfazer a critérios de testes pré-estabelecidos.

O critério (a) foi avaliado empiricamente, através de consultas aos desenvolvedores. Os critérios (b) e (c) foram avaliados através de métricas descritas no item precedente. As

relações entre essas métricas e esses critérios são apresentadas na Tabela 5.2, com base nos trabalhos [Bin94, Bas96, Bie96, Chi94].

	Testabilidade	**Reusabilidade**
WMC	Quanto maior o número de métodos; mais complexa a classe; mais propensa a falhas e mais difícil de testar; maior número de testes necessários.	Quanto maior o número de métodos; maior tendência para a classe ser específica, menor chance de reuso; menor reusabilidade
DIT	Quanto maior o nível de profundidade na árvore de herança; mais propensa a falhas, pois herda uma grande quantidade de definições de seus ancestrais; maior número de testes necessários.	Quanto maior o nível de profundidade na árvore de herança; mais especificas, mais difíceis de serem reusadas; menor reusabilidade.
NOC	Quanto maior o número de filhos; mais difícil de ser modificada, pois pode afetar seus filhos; maior exigência de confiabilidade; maior número de testes necessários.	Quanto maior o número de filhos, maior número de classes que reusam seus componentes (variáveis e métodos) através da herança, maior reusabilidade.
CBO	Quanto maior o acoplamento; mais propensa a falhas, pois depende de métodos e objetos definidos em outras classes; maior número de "stubs" e de testes necessários	Quanto maior o grau de dependência, mais classes dependem dela; maior chance para o reuso; maior reusabilidade
# de servidores	Quanto maior o número de servidores; maior dependência de outras classes; maior número de "stubs" e de testes necessários.	Quanto maior o número de servidores; maior a dependência de outras classes; maior número de classes prestam serviços; menor a possibilidade de serem reusadas; menor reusabilidade.
# de clientes	Quanto maior o número de clientes; prestam muito serviços; maior funcionalidade, maior número de testes necessários, porque sua confiabilidade é importante.	Quanto maior o número de clientes; maior o número de classes que utilizam seus serviços; maior reusabilidade
LOC[10]	Quanto maior o número de linhas de código; maior funcionalidade, mais complexas e mais difícil de testar.	Quanto maior o número de linhas de código; mais funcionalidade; pode restringir o reuso; menor reusabilidade
MVG	Quanto maior a complexidade ciclomática; maior o número de decisões (*if e else)* e *loops (while e do)*; maior complexidade; maior dificuldade nos testes.	Quanto maior o número de complexidade ciclomática; maior a funcionalidade; pode restringir o reuso; menor reusabilidade.

Tabela 5.2 Relação entre os critérios e as métricas OO

[10] LOC e MVG são medidas para os métodos, sendo discutíveis como métricas de qualidade para classes.

5.4 Descrição da Aplicação

O sistema de Telemetria e Telecomando (TMTC) [Amb96] está sendo desenvolvido pela divisão de Sistema de Solo do Instituto Nacional de Pesquisas Espaciais (INPE). Este sistema tem o objetivo de suprir os requisitos do programa CBERS (em inglês: *China-Brazil Earth Resources Satellite*). O TMTC é constituído por um conjunto de subsistemas: de telemetria, de telecomando e outros subsistemas de suporte para a interface com o ambiente externo.

O subsistema de telemetria recebe, processa, armazena e exibe todos os dados recebidos do satélite em uma estação terrena e transmite-os para o centro de controle de satélite através de uma rede de comunicação de dados. Os dados de telemetria são exibidos em tempo real ou ainda armazenados em arquivos. O subsistema de telecomando trata da edição, gerência e transmissão de telecomandos do centro de controle de satélites para a estação terrena, para daí serem transmitidos ao satélite.

Diferentemente dos demais subsistemas, o subsistema que engloba a telemetria e o telecomando não representa apenas uma coleção de classes independentes, já que as classes estão fortemente interligadas. Além disso, este subsistema é responsável pelo domínio da aplicação, assim os testes estão direcionados para esta parte do sistema. Mais especificamente, os testes devem se concentrar no subsistema de telemetria, uma vez que na realização deste estudo o subsistema de telecomando ainda estava em fase de implementação.

O sistema TMTC foi desenvolvido para ambiente Microsoft Windows e a implementação está sendo feita utilizando-se o Visual C++ 4.0. O sistema possui um total de 183 classes divididas entre os diversos subsistemas. Particularmente, o subsistema de telemetria possui 31 classes e o subsistema de telecomando possui 19 classes.

5.5 Resultados Coletados

As métricas foram coletadas em todas as classes do subsistema de telemetria da aplicação TMTC, num total de 31 classes. A Tabela 5.3 apresenta os valores das métricas obtidas e a Tabela 5.4 apresenta as estatísticas das 31 classes estudadas.

Classe	WMC	DIT	NOC	CBO	LOC	MVG	# servidores	# clientes
1. CArrValue	5	1	1	1	18	2	1	0
2. CTmAnlg	26	1	1	3	166	32	2	1
3. CTmAnlgStatistics	11	2	0	4	205	46	4	0
4. CTmBarCh	11	0	0	1	444	59	1	0
5. CTmBilvl	13	1	0	1	101	23	1	0
6. CTmCalib	7	0	0	3	59	13	0	3
7. CTmCbersAocs1	5	2	0	1	121	22	1	0
8. CTmCbersAocs2	7	2	0	1	133	31	1	0
9. CTmCbersHouseKeep	7	2	0	1	79	24	1	0
10. CTmCbersNormal	7	2	0	1	126	28	1	0
11. CTmCbersSegment	5	1	4	6	133	31	2	4
12. CTmCList	5	0	0	2	63	15	1	1
13. CTmComServer	6	0	0	1	113	21	1	0
14. CTmCount	11	1	0	1	185	34	1	0
15. CTmDerived	18	1	0	2	202	35	2	0
16. CTmDesc	36	0	10	21	227	57	2	19
17. CTmDigital	9	1	0	1	88	18	1	0
18. CTmDiscr	15	1	0	1	163	38	1	0
19. CTmFrame	4	0	0	4	99	20	3	1
20. CTmList	13	0	0	5	131	18	1	4
21. CTmMeq	11	0	0	2	238	55	2	0
22. CTmOutRg	9	1	0	3	96	12	1	2
23. CTmProcess	46	0	0	12	716	163	7	5
24. CTmProtocol	4	0	2	2	33	6	1	1
25. CTmReportTime	8	0	0	2	225	75	2	0
26. CTmRTProcess	8	1	0	2	100	14	2	0
27. CTmTimer	3	1	0	1	67	14	1	0
28. CTmTimeSat	7	0	0	1	76	16	1	0
29. CTmTimeSdid	8	1	0	1	17	2	1	0
30. CTmValue	7	1	0	3	26	8	0	3
31. CValue	5	0	2	1	13	1	1	0

Tabela 5.3 Valores das métricas coletadas.

Classe	WMC	DIT	NOC	CBO	LOC	MVG	# servidores	# clientes
Máximo	46	2	10	21	716	163	7	19
Mínimo	3	0	0	0	13	1	0	0
Média	10,87	0,74	0,64	2,93	144	30,1	1,51	1,42

Tabela 5.4 Estatísticas das 31 classes estudadas.

5.6 Análise dos Resultados

A Figura 5.2 apresenta a distribuição das métricas DIT e NOC, sendo as classes representadas por números de acordo com a ordem mostrada na Tabela 5.3. Estes resultados indicaram baixa profundidade quanto a hierarquia de herança, existindo apenas

dois níveis de profundidade (DIT). As classes com maior profundidade são: CTmAnglStatistics, CTmCbersAocs1, CTmCbersAocs2, CTmCbersNormal e CTmCbersHouseKeep. Além disso, as classes possuíam, em geral, poucos filhos (NOC), com exceção da classe CTmDesc.

A Figura 5.3 apresenta o grau de dependência das classes estudadas (CBO). Com este gráfico foi possível observar que todas as classes do subsistema estão interligadas, não existindo desta forma classes independentes. Pode-se destacar as seguintes classes com maiores graus de dependência: CTmDesc, CTmProcess, CTmCbersSegment, CTmList, CTmFrame.

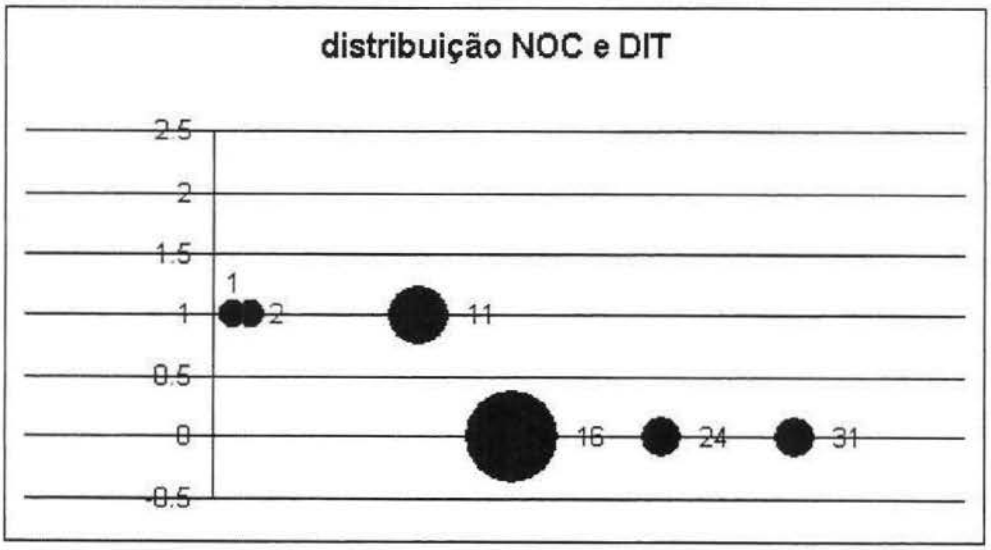

Figura 5.2 Distribuição das classes segundo DIT e NOC

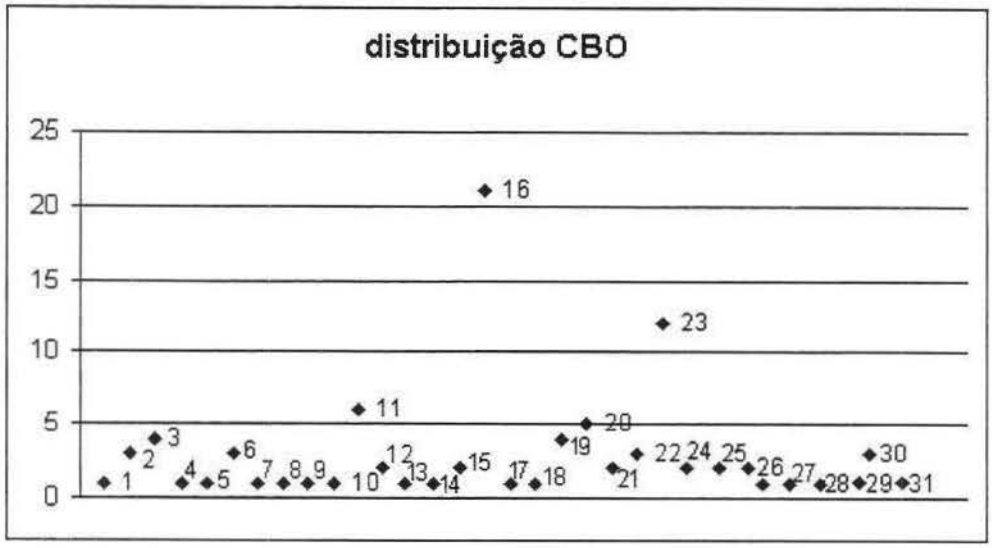

Figura 5.3 Distribuição das classes de acordo com o CBO

Na Figura 5.4 são apresentados os resultados das métricas segundo o número de clientes. Classes que se destacaram com o número de clientes são: CTmDesc, CTmProcess, CTmCbersSegment e CTmList.

A Figura 5.5 apresenta a distribuição das métricas de acordo com o WMC, sendo possível observar classes que se destacaram quanto ao número de métodos, tais como: CTmProcess, CTmDesc, CTmAngl, e CTmDerived.

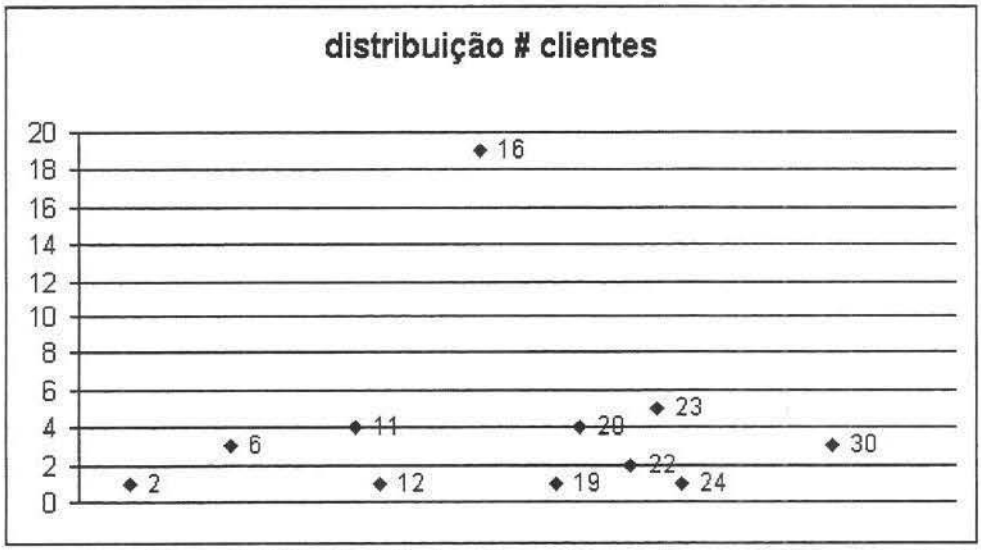

Figura 5.4 Distribuição das classes de acordo com o número de clientes

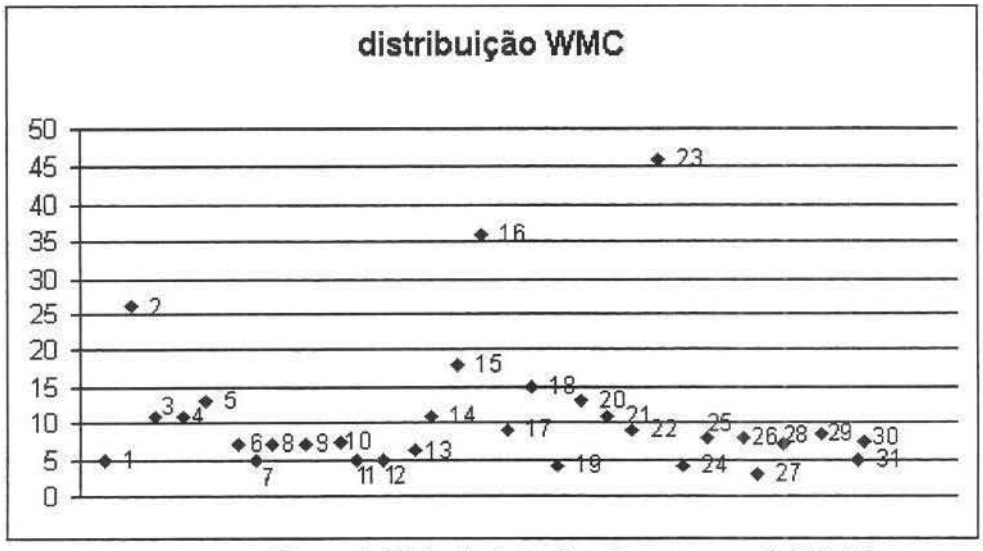

Figura 5.5 Distribuição das classes segundo WMC

Em relação as medidas do LOC, foi possível observar classes que se destacaram quanto ao número de linhas de código (média 144 – Tabela 5.4), tais como: CTmProcess, CTmBarch, CTmMeq, CTmDesc, CTmReportTime, CTmAnglStatistics e CTmDerived. De acordo com o MVG, pode-se verificar a existência de classes valores maiores, bem acima da média (30,1 – Tabela 5.4), são elas: CTmProcess, CTmReportTime, CTmBarCh, CTmDesc e CTmMeq.

De acordo com os resultados das métricas, foi possível determinar as classes com menor testabilidade e as classes com alta reusabilidade. A Tabela 5.5 indica, conforme a Tabela 5.3, as classes em que os mecanismos DFT são mais recomendados.

Classe	Autoteste	Somente mecanismo BIT	Motivo
CTmAngl		X	Valor significativo [11]WMC
CTmAnglStatistics		X	Nível de [12]DIT
CTmBilvl		X	Valor significativo WMC
CTmCbersAocs1		X	Nível de DIT
CTmCbersAoc2		X	Nível de DIT
CTmCberHouseKeep		X	Nível de DIT
CTmCberNormal		X	Nível de DIT
CTmCbersSegment	X		Valor significativo NOC Valor significativo CBO Valor significativo de clientes
CTmDerived		X	Valor significativo WMC
CTmDesc	X		Valor significativo WMC Valor significativo NOC Valor significativo CBO Valor significativo de clientes
CTmList		X	Valor significativo WMC Valor significativo CBO Valor significativo de clientes
CTmProcess	X		Valor significativo WMC Valor significativo CBO Valor significativo de clientes
CTmProtocol		X	Valor significativo NOC

Tabela 5.5 Classes classificadas de acordo com a testabilidade e a reusabilidade

Segundo o critério de criticalidade, tem-se as seguintes classes consideradas mais críticas pelos desenvolvedores do TMTC: CTmDesc e CTmProcess. Por serem classes bases de generalização do processamento das mensagens de telemetria recebidas das

[11] Considera-se como significativo, nesse texto, um valor acima da média mostrada na Tabela 5.4
[12] Considera-se o nível da hierarquia de herança significativo como sendo maior ou igual a 2.

estações terrenas, que se comunicam com o satélite. Além disso, são classes que possuem grande quantidade de relacionamentos e de métodos.

5.7 Recomendações aos Desenvolvedores

Com base nos resultados das métricas, foi possível tirar algumas conclusões sobre o desenvolvimento e implementação da aplicação TMTC. Sendo assim, algumas recomendações são sugeridas aos desenvolvedores, com o objetivo de possibilitar um aumento de testabilidade em desenvolvimentos futuros:

1. o *fato de não usar atributos públicos:* a não utilização de atributos públicos é uma boa estratégia de desenvolvimento, pois diminui o acoplamento entre os objetos;
2. *o desenvolvimento de classes com alto WMC:* classes com WMC alto devem ser revistas (em especial a classe CTmDesc, pois possui muitos filhos) pois classes com WMC alto tendem a ser mais complexas e difíceis de serrem testadas.;
3. *o desenvolvimento de classes com alto NOC:* a existência de classes com alto NOC, mostra que o reuso (a maior vantagem da programação OO) foi bastante empregado. Entretanto, na fase de testes estas classes merecem uma maior atenção por parte dos desenvolvedores, uma vez que possuem muitos filhos que utilizam e modificam suas definições, possibilitando a propagação de falhas pelo sistema. Um exemplo, a classe CTMDesc que possui 10 filhos, em conjunto com WMC e LOC altos, a tendência é que seus filhos sejam complexos, devendo ser testados cuidadosamente;
4. *o desenvolvimento de classes com alto grau de acoplamento:* classes com alto grau de acoplamento devem ser revistas, pois estes índices indicam que classes estão muito interligadas, já que cada classe utiliza serviços de outras, dificultando a realização de testes e facilitando a ocorrência de falhas.

5.8 Conclusões e Extensões

Este artigo apresentou o uso de métricas de software na realização de uma análise da reusabilidade e testabilidade de uma aplicação real, com o objetivo principal de auxiliar nos testes de classes dessa aplicação. Neste estudo foi possível mostrar como a utilização de métricas pode ser útil na determinação de classes que necessitavam de mais testes. As

principais contribuições foram: (1) o auxílio na escolha de classes a se tornarem autotestáveis; (2) o auxílio na escolha de classes a utilizarem os mecanismos BIT.

Dando continuidade a esse trabalho serão realizados os testes das classes escolhidas. Esta fase engloba as seguintes atividades: (1) a determinação da ordem dos testes. A ordem dos testes determina uma sequência de testes para as classes de acordo com os relacionamentos (herança, agregação e associação) existentes entre elas, com o objetivo de otimizar os testes e evitar a utilização de "stubs"; (2) a elaboração dos modelos de testes para as classes escolhidas; (3) a construção das especificações de testes, com base nos modelos de testes; (4) a introdução das assertivas e dos métodos relatores (métodos BIT); (5) a realização dos testes e análise dos resultados.

Capítulo 6

O Uso de Classes Autotestáveis em uma Aplicação Aeroespacial

Este capítulo apresenta o uso dos conceitos de autoteste no conjunto de classes escolhidas de um sistema real.

A aplicação alvo é o sistema de Telemetria e Telecomando desenvolvido pela Divisão de Sistemas de Solo do Instituto Nacional de Pesquisas Espaciais (INPE). Apesar do interesse de se testar todas as classes do TMTC, o fator tempo não possibilitou a realização dos testes de todas as classes da aplicação. Assim, os testes se concentraram nas classes escolhidas através dos critérios estabelecidos no Capítulo 05.

Este experimento teve como objetivo principal verificar a aplicabilidade da proposta em um ambiente real de desenvolvimento. Apesar de encontrar falhas ser o principal objetivo dos testes, nossa maior preocupação foi verificar as dificuldades e as vantagens existentes em se aplicar a metodologia, uma vez que o sistema já tinha sido previamente testado.

Assim, o capítulo está organizado da seguinte forma: a Seção 6.1 descreve a estrutura do sistema TMTC; a Seção 6.2 apresenta a estratégia utilizada para os nossos testes; a Seção 6.3 apresenta os passos realizados para a introdução do autoteste; a Seção 6.4 apresenta como se deu a geração dos testes; a Seção 6.5 apresenta como se deu a execução dos testes; a Seção 6.6. e Seção 6.7 apresentam as dificuldades e resultados encontrados.

6.1 Descrição do Sistema

Como descrito no capítulo anterior, o sistema de Telemetria e Telecomando (TMTC) vem sendo desenvolvido pela divisão de Sistema de Solo do Instituto Nacional de Pesquisas Espaciais (INPE). Este sistema tem o objetivo de suprir os requisitos do programa CBERS

(em inglês: *China-Brazil Earth Resources Satellite*), e já vem sendo utilizado para a comunicação do satélite CBERS com a estação de Solo.

A arquitetura do sistema é composta por cinco componentes, ilustrado na Figura 6.1. Os componentes são:

- **DBI (Data Base Interface):** é responsável pela interface entre a aplicação e o banco de dados;
- **ESI (External System Interface):** é responsável pela interface entre a aplicação e outros sistemas e dispositivos;
- **SI (Support Interface):** fornece a interface entre a aplicação e as funções do sistema operacional e outros serviços gerais.
- **HCI (Human Computer Interface):** fornece uma interface amigável entre o usuário e a aplicação.
- **PD (Problem Domain):** está diretamente relacionado com o domínio da aplicação; este componente é subdividido entre o subsistema de telemetria e o de telecomando.

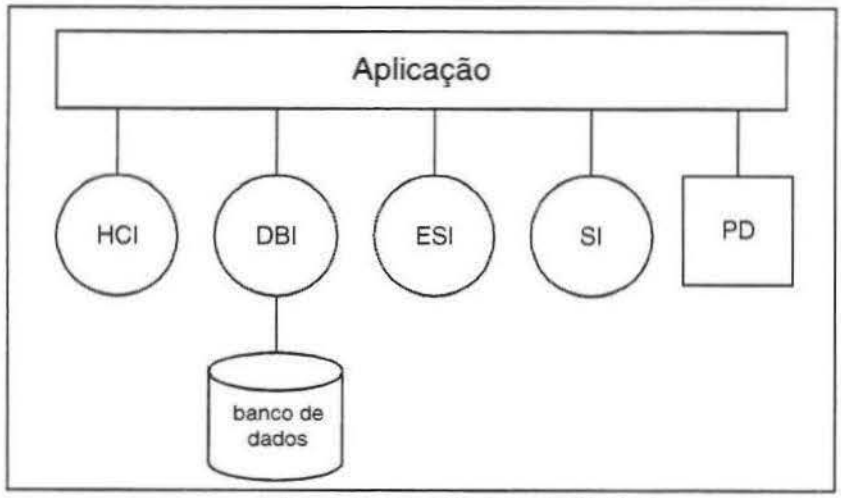

Figura 6.1 Arquitetura do TMTC

O subsistema de telemetria recebe, processa, armazena e exibe todos os dados recebidos do satélite em uma estação terrena e transmite-os para o centro de controle de satélite através de uma rede de comunicação de dados. Os dados de telemetria são exibidos em tempo real ou ainda armazenados em arquivos. O subsistema de telecomando trata da edição, gerência e transmissão de telecomandos do centro de controle de satélites para a estação terrena, para posteriormente serem transmitidos ao satélite.

O sistema TMTC foi desenvolvido para ambiente Microsoft Windows e a implementação está sendo feita utilizando o Visual C++ 4.0. O sistema possui um total de 183 classes divididas entre os diversos subsistemas. Particularmente, o subsistema de telemetria possui 31 classes e o subsistema de telecomando possui 19 classes.

Os testes se concentraram no componente relacionado ao domínio do problema, pois, diferentemente dos demais subsistemas, este componente além de tratar o problema da aplicação, não representa apenas uma coleção de classes independentes, já que as classes estavam fortemente interligadas. Mais especificamente, o entendimento da aplicação se concentrou nas classes do subsistema de telemetria, pois o subsistema de telecomando ainda estava em fase de implementação.

A equipe de desenvolvimento, responsável pelo sistema TMTC, realizou os testes baseados na abordagem hierárquica para testes OO, proposta por Siegel [Sie96]. A abordagem divide o sistema em diferentes níveis de hierarquia, sendo eles: objetos, classes, componentes de base, subsistemas e sistema. Desta forma, alguns dos dados de testes realizados pelos desenvolvedores serviram como base para a construção da estratégia de teste apresentada a seguir. Entretanto, os desenvolvedores não mantiveram alguns dos testes feitos, principalmente em relação aos testes de unidades. Isto poderia ter melhorado o entendimento e a construção do nosso plano de teste e principalmente, uma comparação entre os conjuntos de testes gerados.

Na Seção seguinte é descrita a estratégia de teste usada para os testes das classes de telemetria escolhidas.

6.2 Estratégia Utilizada para os Testes

A estratégia utilizada para a introdução do autoteste levou em conta os seguintes aspectos:

- **Escolha das classes a serem testadas:**

Uma análise da aplicação foi realizada para determinar quais classes se tornariam autotestáveis. A escolha se baseou nos critérios: criticalidade, reusabilidade e testabilidade (detalhes no Capítulo 05). As seguintes classes foram escolhidas: CTmAnlg, CTmBilvl, CTmCbersAocs1, CTmCbersAocs2, CTmCbersHouseKeep, CTmCbersNormal, CTmCbersSegment, CTmDerived, CTmDesc, CTmList, CTmProcess, CTmProtocol.

Algumas destas classes foram escolhidas apenas para a introdução da capacidade BIT (introdução de assertivas e métodos relatores). Entretanto, para se verificar o uso destes mecanismos, seria necessário a existência prévia de testes, como por exemplo os que foram realizados pelos desenvolvedores. Como não foi possível o aproveitamento destes testes, optou-se então pela introdução do autoteste em todas as classes escolhidas, ou seja, além da capacidade BIT, criar também a especificação de teste e o driver específico (que contém a sequência de teste executável) para cada classe.

- **Ordem dos Testes:**

A ordem de realização dos testes define a sequência em que as classes devem ser testadas de acordo com os relacionamentos existentes entre elas, tais como: herança, agregação e associação. Esta ordem facilita o entendimento e a execução dos testes, além de possibilitar a redução do número de "*stubs*". Com base nos dados obtidos através das métricas aplicadas às classes e das informações coletadas dos artigos de Kung et. al. [Kun95] e Thévenod-Fosse & Waeselynck [The97], foi possível construir manualmente uma ordem de teste para as classes do subconjunto de Telemetria, como ilustrado na Figura 6.2.[13] Os retângulos grifados representam as classes escolhidas para os testes do subsistema; os arcos são direcionados e representam a dependência existente entre as classes, estes partem da classe prestadora de serviços (origem) para a classe dependente (destino); os números nos retângulos representam uma ordem de teste sugerida com base nesta dependência.

- **Escolha de Classes Adicionais:**

Durante a preparação dos testes e construção da ordem de teste, percebeu-se que algumas classes, apesar de serem simples, eram utilizadas pelas demais classes escolhidas. Neste requisito, existiam algumas opções que poderiam ser utilizadas: (1) a criação de "stubs"; (2) o uso das classes sem a realização de testes, partindo do pressuposto que já haviam sido testadas; ou (3) a inserção do autoteste às classes, sendo esta a opção escolhida. Portanto, as seguintes classes foram adicionadas ao conjunto de classes a se tornarem autotestáveis: CTmValue, CValue, e CarrValue, que não foram previamente selecionadas segundo critérios descritos no Capítulo 05.

---

[13] A ordem de teste foi obtida manualmente, entretanto recomenda-se o uso de ferramentas para a obtenção de informações mais precisas sobre a dependência entre as classes de um sistema

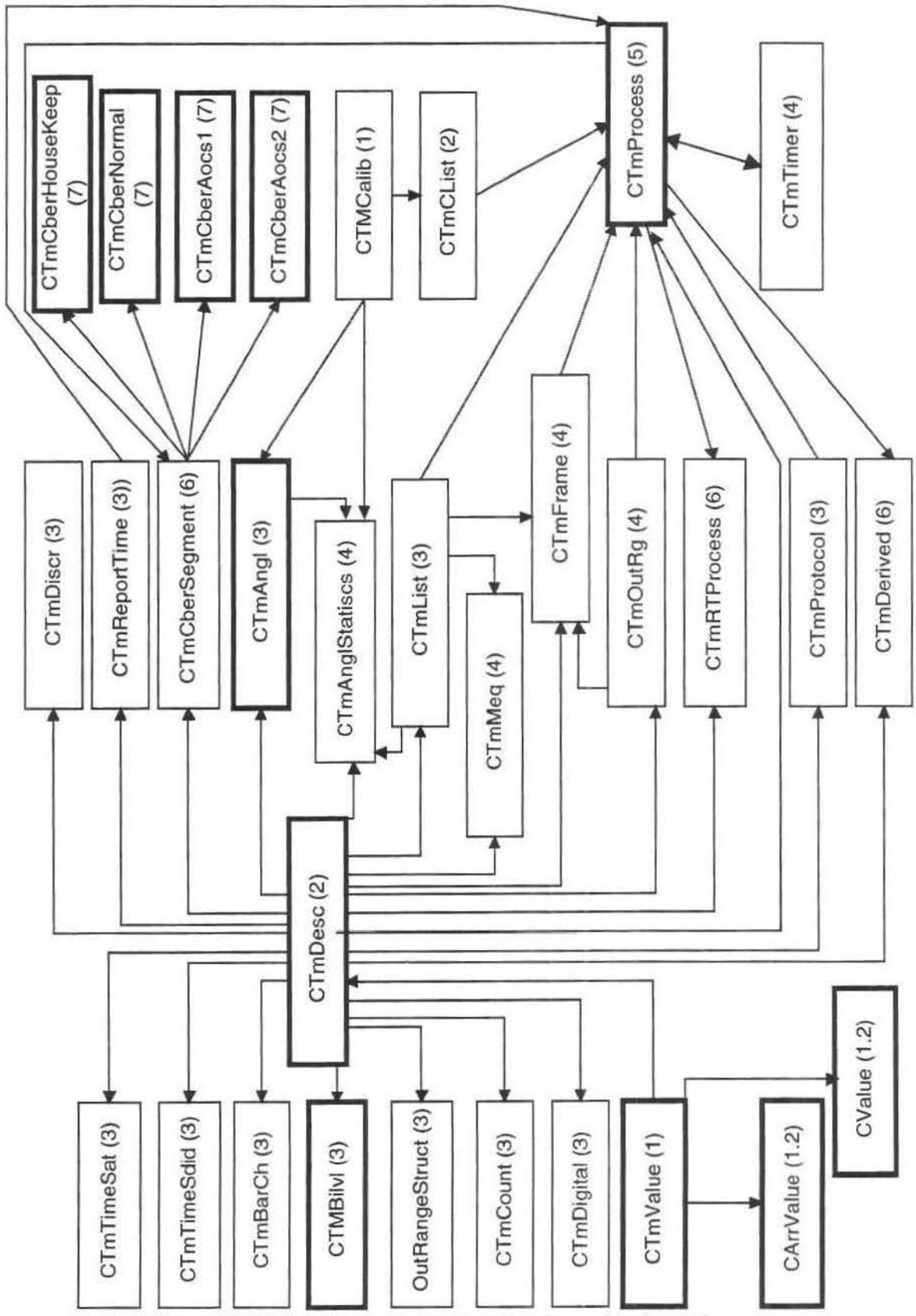

Figura 6.2 Ordem para os testes do subconjunto de telemetria

- **Automação dos Testes:**

O protótipo *Concat* foi utilizado para a geração dos requisitos de testes. O objetivo do uso do protótipo foi automatizar o processo de geração dos requisitos de testes, reduzindo o esforço para geração dos testes cada vez que uma classe é reusada.

6.3 Passos para a Construção de Classes Autotestáveis

Como ilustrado no Capítulo 03, a metodologia de teste adotada engloba um conjunto de passos para a introdução do autoteste nas classes. Estes passos foram seguidos para a introdução do autoteste nas classes de telemetria, descritos a seguir.

1. **Construção do modelo de teste para a classe sob teste:**

Um grafo é construído para cada classe, representando o seu modelo de teste. Este pode ser definido como uma abstração da classe e serve de base para a geração dos requisitos de teste. No requisito, a Concat utiliza o modelo de fluxo de transação - MFT (descrito no Capítulo 03), portanto um modelo MFT foi construído para cada classe.

A ordem de construção dos modelos de teste foi estabelecida de acordo com a ordem de teste estabelecida. A Tabela 6.1 descreve características de cada modelo construído.

Classe	Descrição	# métodos	# nós	#arcos
CtmValue	Classe virtual que armazena o valor atual de uma telemetria	07	04	05
Cvalue	Classe que armazena o valor atual de uma telemetria (em apenas 1 byte)	07	04	05
CarrValue	Classe armazena o valor atual de uma telemetria e o número de bytes necessários para armazenar este valor	07	04	05
CtmDesc	Classe base que descreve o que é comum a todas as telemetrias	40	32	40
CtmAnlg	Classe responsável pelo processamento das telemetrias analógicas	50	39	48
CtmBilvl	Classe responsável pelo processamento das telemetrias que possuem dois níveis (1 ou 0).	46	37	46

Classe	Descrição	# métodos	# nós	#arcos
CtmProcess	Classe virtual responsável pelo processamento de mensagens de telemetria.	41	13	30
CtmCbersSegment	Classe responsável pelo processamento dos frames de telemetria dos segmentos chamados dados armazenados normais (*"stored data normal"*)	42	14	35
CtmCberHouseKeep	Classe responsável pelo processamento dos frames de telemetria dos segmentos chamados *"house-keep"*	42	14	35
CTmCbersAocs1	Classe responsável pelo processamento dos frames dos segmentos chamados dados armazenados Aocs1 (*"stored data Aocs1"*)	42	14	35
CTmCbersAocs2	Classe responsável pelo processamento dos frames dos segmentos chamados dados armazenados Aocs2 (*"stored data Aocs2)"*	42	14	35

Tabela 6.1 Descrição do conjunto de classes escolhidas

Após a construção dos modelos de teste para classe, uma avaliação foi feita pelos desenvolvedores para verificar se os modelos refletiam as funções de cada classe e se estavam de acordo com a sua especificação. A ajuda dos desenvolvedores e a implementação foram essenciais para o entendimento das classes, uma vez que a documentação existente não descrevia as funções e as restrições das classes de forma mais detalhada e atualizada.

2. **Construção de uma especificação de teste que descreve o modelo de teste construído:**

A especificação de teste foi construída a partir dos modelos obtidos no passo 1. Ela descreve o modelo na forma de fatos. Para cada nó, informações são fornecidas para auxiliar a geração das mensagens aceitas pela classe sob teste, a geração dos dados de teste e a reutilização de requisitos de teste. O formato desta especificação pode ser visto no Capítulo 03.

3. **Associação da especificação de teste à classe sob teste**

Esta associação é feita nomeando-se o arquivo contendo a especificação de teste de acordo com um padrão: nome da classe e a extensão "mt" representando o modelo de teste. Um relacionamento entre a classe e sua especificação deve ser criado e mantido.

4. **Instrumentação da classe**

Neste passo, a capacidade de teste embutido é inserida na classe sob teste: métodos de teste embutido (métodos relatores) e assertivas (pré e pós condições).

A ajuda dos desenvolvedores foi importante para localizar as posições adequadas para as assertivas.

5. **Redefinição dos métodos de testes embutidos**

Neste último passo, a redefinição dos métodos de teste embutido é realizada. Estes métodos precisam ser redefinidos para fornecer a invariante da classe e relatar os valores dos atributos da classe. Para cada classe é implementado um método relator adequado, capaz de obter as saídas de cada método executado.

Mais detalhes sobre os passos realizados para a construção das classes autotestáveis podem ser vistos no Capítulo 03.

6.4 Geração dos Requisitos de Testes

A *Concat* gera um conjunto de requisitos de testes com base nos modelos de teste construídos, sendo que um requisito de teste é composto por um ou mais métodos que exercitam uma transação do modelo. A *Concat* parametriza os métodos através da escolha *aleatória* de valores do domínio de entrada. Na parametrização dos testes são fornecidos dados de entradas para os parâmetros de cada método que compõe um requisito de teste. Entretanto, a *Concat* não gera dados não escalares (vetores, objetos), sendo portanto necessário que o testador forneça manualmente os dados de entrada destes parâmetros antes da execução propriamente dos testes.

Para aumentar a eficiência dos requisitos de teste, já que a escolha aleatória de dados é pouco eficaz, optou-se por utilizar *teste de dados*, mais particularmente, testes de *partições de equivalência* e de *valores limites*. Esta categoria de teste tem por objetivo a busca de falhas a partir da manipulação dos dados. Para isto, os dados de teste são escolhidos de acordo com a características dos domínios de entrada e de saída do software. No requisito dos *testes de partições de equivalência*, o domínio de entrada do software ou componente pode ser dividido em dois subdomínios: um para as entradas válidas e outro para as entradas inválidas. Já os *testes dos valores limites* parte do pressuposto que os

erros ocorrem justamente nos limites dos dois subdomínios (entradas válidas e inválidas), levando em conta também o domínio de saída na seleção dos dados. Portanto, estes testes buscam selecionar os valores limites de cada entrada. Neste trabalho, consideramos apenas os limites de entradas dos dados.

Para a escolha dos conjuntos de entrada, os dados contidos no banco de dados do sistema foram também utilizados. A partir destas informações e do auxílio dos desenvolvedores, foi possível identificar as partições válidas e inválidas para cada classe. Além disso, com o objetivo de possibilitar reutilizações futuras, foi criado um arquivo de histórico contendo o conjunto de testes gerado para cada classe, facilitando assim testes posteriores.

6.5 Execução e Análise dos Testes

O objetivo desta fase foi executar a sequência teste para cada classe tornada autotestável e analisar os resultados obtidos. A execução das classes seguiu a sequência estabelecida na *ordem de teste* descrita na Seção 6.2. Entretanto, apesar da ordem de teste falicitar o entendimento e possibilitar diminuir o número de *stubs* necessários, esta não englobava as dependências das classes escolhidas com as dos outros subsistemas (por exemplo, de classes do subsistema de banco de dados). Estas dependências não foram consideradas, pois era inviável construir manualmente um modelo que englobasse todas as dependências do sistema. Entretanto, percebeu-se que algumas das classes escolhidas possuíam grande dependência com classes de outros subsistemas, tornando as classes complexas e forçando a existência de *stubs*. Esta dependência dificultou o entendimento e análise dos resultados obtidos com os testes.

Devido a estas circunstâncias, apesar de possuírem os mecanismos de autoteste, não houve a execução e análise dos testes. Em particular, as classes relacionadas ao processamento das telemetrias derivadas da CTmProcess, pois utilizavam um grande número de serviços de outras classes, principalmente relacionadas às classes de banco de dados, sendo inviável e complexo a construção de *stubs*.

A Tabela 6.2 mostra as classes efetivamente testadas e o número de testes gerados para cada uma delas. A quarta coluna descreve a quantidade de conjuntos de dados de entrada

construídos utilizando os testes de *partições de equivalência* e de *valores limites* descritos na Seção 6.4.

Classe	# requisitos de teste gerados pela *Concat*	# requisitos de teste reusados da classe base	# de conjuntos de dados de entrada
CTmDesc	11	0	07
CTmBilvl	4	11	07
CTmAnlg	4	11	08
CValue	11	11	07
CArrValue	11	11	07
CTmValue	11	0	07

Tabela 6.2 Requisitos de testes gerado para cada classe

Vale ressaltar que as classes CTmDesc e CTmValue são classes abstratas, não podendo assim ser diretamente testadas. Assim, os requisitos de testes gerados para estas classes foram instanciados com objetos de suas classes derivadas. Os requisitos de testes gerados para estas duas classes foram portanto reusadas nos testes das classes derivadas.

6.6 Dificuldades Encontradas

Algumas dificuldades foram encontradas na realização dos testes, descritas a seguir:

- a ausência de uma documentação apropriada para o entendimento das classes, com a capacidade de rastreamento, possibilitando encontrar um dado componente de software que implementa parte da especificação;
- a introdução de assertivas foi problemática, devido a dificuldade no entendimento da funcionalidade das classes. Isto porque o sistema não era trivial e não tratava de dados simples. Além disso, um estudo do código foi necessário para o entendimento das classes. A existência de uma documentação que descrevesse as condições válidas de entrada e saída da classe poderia ter facilitado a tarefa.
- o alto grau de interdependência com classes de outros subsistemas dificultou a realização dos testes, pois foi necessário entender a sequência de execução dos métodos, além de obrigar a criação de *stubs*. A necessidade de uma ferramenta que determinasse a ordem dos testes se mostrou essencial para a melhoria dos testes.

6.7 Resultados Obtidos

Apesar deste sistema já estar sendo utilizado, e já ter sido previamente testado, algumas observações podem ser feitas em relação aos resultados obtidos. Os testes de partições de equivalência mostraram a falta de robustez dos métodos que tratam as entradas vindas de outros subsistemas, pois mostraram que valores inválidos não foram tratados e testados pelos desenvolvedores. A permanência das assertivas pode ser útil para detectar erros durante a operação ou quando a classe for reusada, evitando que o sistema utilize dados inválidos.

Nenhuma falha grave foi detectada pelos testes, isto talvez porque o sistema já foi previamente testado e vem sendo utilizado no projeto CBERS. Os testes conseguiram mostrar melhorias que poderiam ser introduzidas, como por exemplo: o tratamento de valores inválidos e a documentação detalhada do sistema.

Além disso, estes testes poderão ser reutilizados quando as classes forem modificadas ou reusadas. Alguns fatores positivos já podem ser apontados com o reuso dos testes feitos:

- as assertivas já estão inseridas no sistemas e podem servir de oráculo parcial nos testes;
- a especificação dos testes que está associada à classe pode ser utilizada para o próprio entendimento das funcionalidades da classe;
- o modelo de teste pode ser reutilizado sofrendo apenas alterações necessárias para refletir as modificações feitas na classe;
- os requisitos de testes construídos poderão ser reutilizados nos testes de classes reusadas sem modificações;
- a existência de classes abstratas, que não possibilita o teste de forma direta, foi contornada: (i) pelo fato de se ter usado testes de caixa preta, que permite gerar testes para classes abstratas; e (ii) o uso da abordagem HIT, que permite que estes testes sejam reusados nos testes das subclasses concretas;
- a existência de informações escondidas (encapsulamento) em conjunto com a herança que dificulta observar as saídas esperadas, o que seria necessário em alguns requisitos a quebra do encapsulamento, foi evitado através do uso dos mecanismos

embutidos, que dá visibilidade ao estado interno, reduzindo a dificuldade criada pelo encapsulamento.

Capítulo 7

Conclusão

Este trabalho apresenta uma avaliação empírica da abordagem proposta para a construção e uso de classes autotestáveis. Nessa abordagem as classes são acrescidas de mecanismos BIT (em inglês, "*Built-in Test*") e de uma especificação de teste para permitir que testes sejam gerados e avaliados internamente. Além do conceito do autoteste, a Técnica Incremental Hierárquica (HIT) [Har92] também foi usada com o objetivo de permitir que os testes de uma superclasse possam ser reusados nos testes de uma subclasse.

Como contribuição desse trabalho pode-se citar:

- Uma avaliação da metodologia para a construção e uso de classes autotestáveis, usando como exemplo classes da biblioteca *MFC*. Estas classes são compostas de mecanismos BIT (assertivas e métodos relatores) e uma especificação de teste, que permite a geração, execução e avaliação semi-automática dos testes. Além disso, o uso da técnica incremental hierárquica na geração dos testes, que permite o reuso de testes das classes, nos testes das subclasses. Por fim, o uso do modelo de fluxo de transação para a construção da especificação de teste das classes.
- A realização de um estudo empírico para investigar a eficácia na detecção de falhas de requisitos de testes construídos com base na metodologia do trabalho, através de uma análise de mutação, usando operadores de *Mutação de Interface*. Dois estudos foram realizados, dos quais foram obtidos os seguintes resultados: (a) a estratégia adotada para a geração dos requisitos de teste possui um bom potencial na determinação de falhas, isto porque obteve um escore de mutação de 0,9574. (b) a necessidade de se retestar a classe dos métodos da classe derivada, reutilizando os testes gerados para a classe base, para evitar que possíveis falhas se propaguem, já que se tentou analisar o comportamento do conjunto de teste baseado na técnica HIT, com a ocorrência de falhas na classe base. Isto porque todos os mutantes gerados para um método herdado sem modificação e que não é chamado por nenhum outro método não foram detectados. Mais avaliações devem ser feitas para confirmar os

resultados obtidos, usando um número maior de classes. Além disso, percebeu-se a importância do uso de uma ferramenta automatizada para a geração dos mutantes para programas em C++. Entretanto, os resultados deste estudo são importantes para mostrar que os requisitos de testes gerados pela metodologia criada foram eficazes na detecção de falhas em sistemas OO.

- O uso de métricas de software na realização de uma análise da reusabilidade e testabilidade de uma aplicação real, com o objetivo principal de auxiliar nos testes de classes dessa aplicação. Neste estudo foi possível mostrar como a utilização de métricas pode ser útil na determinação de classes que necessitavam de mais testes. As principais contribuições foram: (1) o auxílio na escolha de classes a se tornarem autotestáveis; (2) o auxílio na escolha de classes a utilizarem os mecanismos BIT. O uso das métricas de software incentivou o grupo de desenvolvimento a utilizar estas medidas nos outros módulos do sistema.
- A apresentação da experiência obtida com os testes das classes de uma Aplicação Aeroespacial. Apesar de nenhuma falha grave ter sido detectada pelos testes, isto talvez porque o sistema já foi previamente testado e vem sendo utilizado no projeto CBERS. Os testes mostraram algumas melhorias que poderiam ser introduzidas, como por exemplo: o tratamento de valores inválidos e a documentação detalhada do sistema. Além disso, o uso dos mecanismos de testes embutidos, que permite a inserção da especificação de teste, métodos relatores, e assertivas dentro das classes, poderá facilitar o reuso dos testes nas novas versões já previstas para o sistema.

Trabalhos Futuros

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

- O uso da estratégia de teste apresentada na dissertação nos testes de componentes de software, podendo englobar teste de integração entre classes relacionadas;
- A realização de uma avaliação da estratégia para testes de regressão, usando uma ferramenta que possa avaliar a cobertura de código através da aplicação de um subconjunto de requisitos de teste gerado;
- A construção de uma ferramenta completa baseada na estratégia de teste proposta, englobando a construção de um modelo de fluxo de transação completo; a definição

de métodos herdados, novos e redefinidos; a criação do histórico de teste para as classes; e o auxílio na construção de stubs. Neste sentido, a ferramenta seria composta de uma interface, um gerador de teste, um histórico de teste e um oráculo.

Referências Bibliográficas:

[Amb96] A.M. Ambrósio, L.S.C. Gonçalves; M.G.V. Ferreira, and P.E. Cardoso. "Brazilian Experiences in Upgrading a Satellite Control System". In: SPACEEOPS96, 1996.

[Bar98] S . Barbey, D. Buchs, and C. Péraire. "Issues and Theory for Unit Testing of Object-Oriented Software", 1998. In: http://www.di.epfl.ch.

[Bas96] V. Basili, L. Briand, and W.L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators". In: *IEEE Transaction on Software Engineering;* vol 22, no. 10, pg 751-761, October 1996.

[Bei90] B. Beizer. "Software Testing Techniques"; International Thomson Computer, 2ª edition, 1990.

[Bie95a] J.M. Bieman and J.X. Zhao. "Reuse Through Inheritance: A Quantitative Study of C++ Software". In: *Proc. ACM Symposium on Software Reusability*, April 1995.

[Bie95b] J.M. Bieman and B. Kang. "Cohesion and Reuse in an Object-Oriented System". In: *Proc. ACM Symposium on Software Reusability,* Seattle USA, pg 259-262, 1995.

[Bie96] J.M. Bieman. "Metric Development for Object-Oriented Software". In: *Proc. Software Mearurement,* International Thompson Computer Press, pg 75-92, 1996.

[Bin94] R.V. Binder. "Design for Testability with Object-Oriented Systems". In: *Communications of the ACM*, v. 37 (9), pg 87-101, September 1994.

[Bin96] R. V. Binder. "The FREE Approach to Object-Oriented Testing: an Overview",1996. In : www.rbsc.com/pages/FREE.html.

[Buz98] L.E. Buzato and C.M.F. Rubira. "Construção de Sistemas Orientados a Objetos Confiáveis". In:11ª Escola de Computação. Rio de Janeiro, 1998.

[CCC98] CCCC – *"C and C++ Code Counter"*, 1998. tool obtained in: http://www.fste.ac.cowan.edu.au/~tlittlef/cccc_ug.htm.

[Chi94] S.R. Chidamber and C.F. Kemerer. "A Metric Colletion and Evaluation with a Software Process". In: *IEEE Trans. Software Eng.*, vol. 20, no.6, pg 476-493, June 1994.

[Cor91] M.L. Cortes and J.M.F. Neto. "Introdução aos Teste de Circuitos Digitais". In: *V EBAI*, 1991.

[Del97] M.E. Delamaro and J.C. Maldonado. "Teste de Integração: Projeto de Operadores para o Critério Mutação de Interface". In: *XI Brazilian Symposium on Software Engineering*; Fortaleza CE, pg 413-428; October 1997.

[Har92] M. J. Harrold and J. D. McGregor. "Incremental Testing of Object-Oriented Class Strutures". ACM, 1992.

[Hof89] D. Hoffman. "Hardware Testing and Software ICS", in *Proceedings of Pacific NW Software Quality Conference*; Portland- Oregon, September 1989.

[Hof95] D. Hoffman and P. Strooper. "The Testgraph Methodology: Automated Testing of Collection Classes", Joop, November 1995.

[Kun95] D. Kung et al.; "Developing an Object-Oriented Software Testing and Maintenance Environment". In: *Communications of the* ACM, v. 38, n°. 10, p. 75-87, October 1995.

[Mye79] G.J. Myers, "The Art of Software Testing". John Wiley & Sons, New York, 1979.

[Per90] D.E. Perry and G.E. Kaiser. "Adequate Testing and Object-Oriented Programming". In: *Journal of Object-Oriented Programming*, pg13-19, Jan-February 1990.

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

[RSTC95] Reliable Software Technology Corporation. "Testability of Object-Oriented Systems". Technical Report - Information Technology Laboratory – NIST GCR 95-675, June 1995.

[Sha00] J. Shafer. "Improving Software Testability"; 2000 ; In: http://www.data-dimensions.com/testersnet/docs/testability.htm.

[Sie96] S. Siegel. *"Object Oriented Software Testing: a Hierarchical Approach"*. John Wiley & Sons, 1ª edition, New York, 1996.

[Smi90] M. D. Smith and D.J. Robson "Object-Oriented Programming – the Problems of Validation". In: *Intern. Conference on Software Maintenance*, EUA, pg 272-281, 1990.

[Smi92] M.D. Smith and D.J. Robson. "A Framework for testing object-oriented programs". In: JOOP; pg 45-53, June 1992.

[Sou97] S.R.S. Souza; J.C. Maldonado; S.R. Vergílio; "Análise de Mutantes e Potenciais-Usos: Uma Avaliação Empírica"; Workshop do Projeto Validação e Teste de Sistemas de Operação"; Águas de Lindóia-SP; p. 85-94; January 1997.

[The97] P. Thévenod-Fosse and H. Waeselynck; "Towards a Statistical Approach to Testing Object-Oriented Programs". In: LAAS Report 96481; Centre National de La Recherche Scientifique, January 1997.

[Toy98] C. M. Toyota and 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, June1998.

[Tra00] Y.L.Traon; D.Deveaux; J. Jézéquel; "Self-Testable Component: from pragmatic tests to Design-for-Testability Methodology". In hhtp://www.irisa.fr/testobjects/self-test; 2000.

[Vin99] A.M.R.Vincenzi, J.C. Maldonado, E.F. Barbosa, and Delamaro, M.E.; "Operadores Essenciais de Interface: Um Estudo de Requisito"; In: *XIII Brazilian Symposium on Software Engineering"*; Florianópolis-SC; p. 373-391; October 1999.

[Voa91] J. Voas, L. Morell, and K. Miller. "Predicting where faults can hide from testing". In: *IEEE Software*, pg 41-48, March 1991.

[Voa92] J.M. Voas. "PIE:A Dynamic Failure - Based Technique". In: *IEEE Transactions on Software Engineering*, vol.18, pg 717-727 August 1992.

[Voa95a] J. Voas et al. "Software Testability: An Experiment in Measuring Simulation Reusability". In: ACM *Symposium on Software Reusability,* pg 247-255, April 1995.

[Voa95b] J.M. Voas ad K.W. Miller. "Software Testability: The New Verification". In: *IEEE Software*, pg 17-28, May 1995.

[Voa97] J.M. Voas. "Software Testability Mearurement for Assertion Placement and Fault Localization". 1997, In: http://www.rstcorp.com.

[Voa98] J. Voas, M. Schmid, and M. Schatz. "A Testability-Based Assertion Placement Tool for Object-Oriented Software" Technical Report - Information Technology Laboratory – NIST CGR 98-735, January 1998.

[Wan99] Y. Wang; G. King; and H. Wickburg. "A Method for Built-in Tests in Component-based Software Maintenance". IEEE, pg 186-189; 1999.

[Yan00] R.L.Yanagawa; E.Martins. "Avaliação Empírica da Eficácia dos Testes baseados no Modelo de Fluxo de Transação em Sistemas Orientados a Objetos". In: *XI CITS:QS – Conferência Internacional de Tecnologia de Software: Qualidade de Software*; Curitiba; June 2000.